Communications-Oriented Development of Component-Based Vehicular Distributed Real-Time Embedded Systems

Saad Mubeen[a,b], Jukka Mäki-Turja[a,b], Mikael Sjödin[a]

Contact: saad.mubeen@mdh.se, +46 21 10 31 91

[a] *Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Västerås, Sweden*

[b] *Arcticus Systems AB, Järfälla, Sweden*

Abstract

We propose a novel model- and component-based technique to support communications-oriented development of software for vehicular distributed real-time embedded systems. The proposed technique supports modeling of legacy nodes and communication protocols by encapsulating and abstracting the internal implementation details and protocols. It also allows modeling and performing timing analysis of the applications that contain network traffic originating from outside of the system such as vehicle-to-vehicle, vehicle-to-infrastructure, and cloud-based applications. Furthermore, we present a method to extract end-to-end timing models to support end-to-end timing analysis. We also discuss and solve the issues involved during the extraction of these models. As a proof of concept, we implement our technique in the Rubus Component Model which is used for the development of software for vehicular embedded systems by several international companies. We also conduct an application-case study to validate our approach.

Keywords:
distributed real-time embedded systems, vehicular software systems, component-based software engineering, real-time systems, timing model, model- and component-based development.

1. Introduction

In most of the model- and component-based software development strategies for automotive and other vehicular applications, models of the behavior of each on-board function are developed and successively refined to reach the implementation of each node or Electronic Control Unit (ECU). In this refinement process, the communications needed for each node are derived and a message set for each on-board network is defined. Moreover, timing parameters and requirements for each message are established. The majority of existing model- and component-based development approaches for vehicular distributed real-time embedded systems[1] allow for structural and functional modeling. They do not support *execution modeling* [1] which is concerned with the modeling of run-time properties and/or requirements (e.g., end-to-end deadlines and jitter) of software functions. The modeling of the systems should extend down to the execution level to allow precise control of resource utilization and that timing requirements are not violated when the system is executed. However, providing such modeling support is very challenging because the functionality in the systems can be realized with more than one execution model, e.g., separate execution models for the nodes and networks. Today, one of the main challenges during the development of the systems in the industry is to model and express timing related information and perform timing analysis [2].

One way to deal with these challenges is to use a component technology that allows model- and component-based development of the systems with the support for modeling, analyzing, predicting and modifying the execution behavior. Such a component technology should complement structural and functional modeling with the modeling of execution requirements at an abstraction level close to the functional specification while abstracting the implementation details. The component technology should support the expression of timing related information and facilitate the identification of timing errors during the development by rendering the modeled application for end-to-end timing analysis with ease and unambiguity.

However, building such a component technology raises many challenges. One of the main reasons behind these challenges is that the development process for these systems in academia and industry may be very different from

[1]Throughout the paper, we use the terms *system* or *application* to refer to component-based vehicular distributed real-time embedded system or application

each other. In academia, the development process often starts with discussions about models and functions. The models are assumed to be platform independent. Further, it is assumed that the models and functions will be deployed on specific platforms at a later stage. However, this way of development for the systems is often not practiced in the industry, especially in the automotive or vehicle domain. The traditional process for the development of these systems in the industry starts with designing the bus (or network) communication. The infrastructure for the system to be developed is already known. In the early stage of industrial development process, usually the focus is on finding the answers to the questions as follows. How many busses will there be in the system? Which nodes will be connected to which bus? How many messages will be there in the system? Which messages will be sent by each node? After finding the answers to these questions, the focus is shifted towards the development of functions. Thus, communications-oriented development process is used.

An important class of emerging distributed applications is novel functionality in road vehicles. These applications realize novel services based on vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communications. Both V2V and V2I are expected to support novel applications for road-safety, traffic efficiency, and driver/passenger comfort and entertainment. Already today, there are examples of traffic related cloud-services, e.g., community map and turn-by-turn navigation such as Waze[2] and traffic-congestion information by Google. However, to be successfully adopted, the development of these new applications needs to be integrated in contemporary workflow for development of vehicular functions. In most of the model- and component-based software development strategies, there is often non-existing, or limited, support to model network traffic originating from outside the vehicle. That is, traffic from V2V, V2I, and other, e.g., cloud-based applications are not naturally modeled and analyzed in existing approaches.

1.1. Goals and paper contributions

In order to provide a model- and component-based approach to support communications-oriented development of vehicular distributed real-time embedded systems, we target the following challenges in this paper[3].

[2]http://www.waze.com

[3]This work is an extension of our previous work [3].

1. Modeling of legacy network communication.
   (a) Use of legacy (previously developed) nodes.
   (b) Development of new nodes that are deployed in legacy systems that use predefined communication rules.
   (c) Adaptation[4] of a node when communication rules change without affecting its internal component design.
2. Extraction of end-to-end timing models from these systems.
3. Modeling and timing analysis of the applications that contain network traffic originating from outside of the system.

In order to provide proof of concept, we realize this technique in the existing industrial model the Rubus Component Model (RCM) [4]. We also conduct the automotive-application case study to validate our approach.

1.2. Paper layout

The rest of the paper is organized as follows. Section 2 discusses the background and related work. In Section 3, we discuss the research problem in detail. In Section 4, we introduce a new approach for modeling legacy network communication. In Section 5, we discuss a method to extract end-to-end timing models. In Section 6, we present a case study. Section 7 concludes the paper and discusses the future work.

2. Background and related work

2.1. The Rubus concept

Rubus is a collection of methods and tools for model- and component-based development of dependable embedded real-time systems. Rubus is developed by Arcticus Systems[5] in close collaboration with several academic and industrial partners. Rubus is today mainly used for development of control functionality in vehicles by several international companies, e.g., BAE Systems Hägglunds[6], Volvo Construction Equipment[7], Knorr-bremse[8], and

[4]We assume the adaptation (redeploying or upgrading) of a node is done offline. Dynamic adaptation and reconfiguration of the system is not within the scope of our current work.

[5]http://www.arcticus-systems.com

[6]http://www.baesystems.com/hagglunds

[7]http://www.volvoce.com

[8]http://www.knorr-bremse.com

Mecel[9]. The Rubus concept is based around RCM and its development environment Rubus-ICE (Integrated Component development Environment) [5], which includes modeling tools, code generators, analysis tools and run-time infrastructure. The overall goal of Rubus is to be aggressively resource efficient and to provide means for developing predictable, timing analyzable and synthesizable control functions in resource-constrained embedded systems.

The Rubus concept jointly considers the following viewpoints during the development. These viewpoints are also shown in Figure 1.

1. The viewpoint of the developer/designer[10].
2. The viewpoint of the analysis framework.
3. The viewpoint of the run-time system.

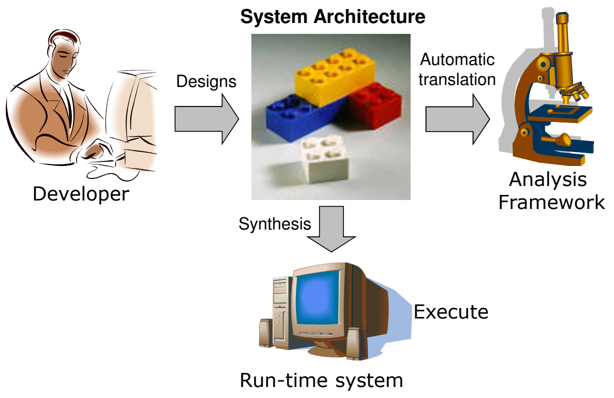


Figure 1: Three main viewpoints jointly considered in Rubus during the development.

In the viewpoint of the developer, the software architecture of the application is modeled in terms of software components and their interactions. This viewpoint consists of tools that handle software complexity, support appropriate level of expressiveness, and provide abstraction mechanisms that hide low-level details (such as source code).

In the viewpoint of the analysis framework, the software architecture is formal enough to render itself to automated analysis (e.g., response-time analysis). The analysis framework has the knowledge of the component architecture as well as the constraints and services provided by the run-time

[9]http://www.mecel.se

[10]Developer refers to the application developer. We overload the terms "developer", "designer" and "user" throughout the paper.

system. This viewpoint comprises of tasks (which are run-time entities), their activations, their interactions, and analysis models of tasks and messages. This viewpoint hides the complexity from the developer by providing automated analysis tools that extract the analysis models from the software architecture.

In the viewpoint of the run-time system, the synthesis takes (as input) the architecture design and possibly some artifacts (such as priorities for a task model) produced by the analysis framework, and maps it to the run-time system. The synthesis tools use the task model attributes and the component architecture, that is both syntactically and semantically correct, to generate code for the run-time system. This viewpoint provides sufficient run-time services to the components of the application while keeping a small footprint for the run-time system. With this view, the entire component framework is provided at development time, but only the parts that are used are mapped down to the actual run-time system.

2.1.1. The Rubus Component Model (RCM)

The purpose of the component model is to express the infrastructure for software functions, i.e., the interaction between the software functions in terms of data and control flow. The control flow is expressed by triggering objects such as internal periodic clocks, interrupts, internal and external events. One important principle in RCM is to separate functional code and infrastructure implementing the execution model. The infrastructure is synthesized from the model.

In RCM, the basic component is called a Software Circuit (SWC). It is the lowest-level hierarchical element in RCM and its purpose is to encapsulate basic functions. The SWCs interact with each other through the use of trigger and data ports. Trigger and data correspond to trigger flow and data flow respectively. An SWC can be seen as a type, or a class, that can be instantiated an arbitrary number of times. By separating functional code from the infrastructure, RCM facilitates analysis and reuse of components in different contexts (an SWC has no knowledge how it connects to other components). Furthermore, the component model has a possibility to encapsulate SWCs into software assemblies enabling the designer to construct the system at different hierarchical levels.

The execution semantics of software components (functions) is simply:

1. upon triggering, read data on data in-ports;

2. execute the function;
3. write data on data out-ports;
4. activate the output trigger.

The software architecture of an example system modeled with RCM is depicted in Figure 2. The example shows how components interact with external events and actuators with regard to both data and triggering.

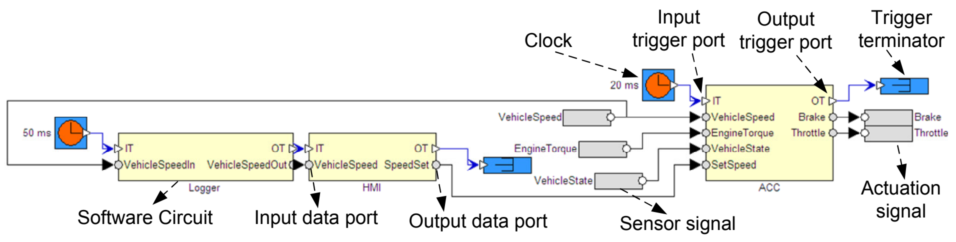


Figure 2: Example of the architecture of a system modeled in RCM.

2.1.2. The Rubus code generator and run-time system

Using the resulting software architecture of connected SWCs, the run-time system maps SWCs to run-time entities; tasks. Each external event trigger defines a task and SWCs connected through the chain of triggered SWCs (triggering chain) are allocated to the corresponding task. All clock triggered "chains" are allocated to an automatically generated static schedule that fulfills the precedence order and other temporal requirements.

Within trigger chains, inter-SWC communication is aggressively optimized to use the most efficient means of communication possible for each communication link. For example, there is no use of semaphores in point-to-point communications within a trigger chain. Another example is sharing of memory buffers between ports when there are no overlapping activation periods. This means that a buffer can be shared between two ports belonging to different SWCs if it can be guaranteed that these ports will never use the buffer space at the same time. This is true in the case of a trigger chain because a task early in the chain can never be active at the same time as a task late in the chain (considering the deadlines of tasks are smaller than their respective periods).

Allocation of SWCs to tasks and construction of schedule can be submitted to different optimization criterion to minimize, e.g., response times for different types of tasks, or memory usage. The run-time system executes all tasks on a shared stack, thus eliminating the need for static allocation of stack memory to each individual task.

2.1.3. The Rubus analysis framework

The Rubus model allows expressing real-time requirements and properties at the architectural level. For example, it is possible to declare real-time requirements from a generated event and an arbitrary output trigger along the trigger chain. For this purpose, the designer has to express real-time properties of SWCs such as Worst Case Execution Times (WCETs). The scheduler will take these real-time constraints into consideration when producing a schedule. For event-triggered tasks, response-time calculations are performed and compared to the requirements. The timing analysis supported by the model includes tighter response-time analysis of tasks with offsets [6], response-time analysis of Controller Area Network (CAN) [7, 8, 9], and distributed end-to-end response time and delay analysis [10].

2.2. Related work

There are many modeling technologies that support component-based development of distributed systems, e.g., Distributed Component Object Model (DCOM) [11], Common Object Request Broker Architecture (CORBA) [12] and Enterprise JavaBeans (EJB) [13]. These models in their original form are not suitable for the development of resource-constrained distributed embedded systems with real-time requirements because they require excessive amount of computing resources, have large memory footprint and have inadequate support for modeling real-time communication. We focus on the component technologies that are targeted towards the vehicular domain.

2.2.1. AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) [14] is an industrial initiative to provide standardized software architecture for the development of software in the automotive domain. In AUTOSAR, the application software is defined in terms of Software Components (SWCs). The distribution of SWCs, their virtual integration and communication at design time is handled by the Virtual Function Bus (VFB). Furthermore, VFB hides the low-level implementation and communication details at the design time. We list some of the differences between AUTOSAR and RCM as follows.

- When AUTOSAR was being developed, there was no focus placed on its ability to specify and handle timing-related information such as real-time requirements and properties. On the other hand, these requirements and capabilities were taken into account right from the beginning during the development of RCM.
- AUTOSAR describes embedded software development at a higher level of abstraction compared to RCM. A Software Circuit in RCM more resembles to a runnable entity (schedulable element) compared to AUTOSAR SWC.
- Unlike AUTOSAR, RCM clearly distinguishes between the control flow and the data flow among SWCs within a node.
- In RCM, special network interface components are used if SWCs require inter-ECU communication; otherwise, SWCs communicate via data and trigger ports. On the other hand, AUTOSAR does not differentiate between intra- and inter-node communication at modeling level. There are no special components in AUTOSAR for modeling inter-node communication.
- AUTOSAR hides the modeling of the execution environment. On the other hand, RCM explicitly allows the modeling of execution requirements, e.g., jitter and deadlines, at an abstraction level close to the functional specification while abstracting the implementation details.

Despite these differences, there are some similarities between AUTOSAR and RCM, e.g., the sender receiver communication mechanism in AUTOSAR is very similar to the pipe-and-filter communication mechanism for components interconnection in RCM. In conclusion, AUTOSAR is more focussed on the functional and structural abstractions, hiding the implementation details about execution and communication. Whereas, RCM is all about modeling, analysis and synthesis of the execution environment of software functions. AUTOSAR hides the details that RCM highlights.

2.2.2. TIMMO, TIMMO-2-USE, TADL and TADL2

TIMing MOdel (TIMMO) [2] is a large EU research project with both academic and industrial partners. It is more academic driven. It is an initiative to provide AUTOSAR with a timing model. The timing extensions proposed in this project are included in the version 4.0 of AUTOSAR specification [15]. TIMMO describes a predictable methodology and a language, Timing Augmented Description Language (TADL) [16], to express timing requirements and constraints during all design phases in the development of automotive embedded systems. TADL is inspired by Modeling and Analysis of Real Time and Embedded systems (MARTE) [17] which is a UML profile for model-driven development of real-time and embedded systems. TIMMO development methodology makes use of structural modeling provided by EAST-ADL [18] which is a domain specific architecture description language targeted towards the automotive domain. TIMMO methodology and its model structure abstract the modeling of communication at implementation level of EAST-ADL where AUTOSAR is used. Hence, the modeling of intra- and inter-node communication mechanisms are the same as that of AUTOSAR. Both TIMMO methodology and TADL have been evaluated on prototype validators. To the best of our knowledge there is no concrete industrial implementation of the results of TIMMO project.

TIMMO-2-USE [19], another large EU research project, is a followup on TIMMO project. In this project, TADL2 language has been introduced which includes a major redefinition of TADL. TADL2 supports the AUTOSAR extensions regarding timing model. Apart from the redefinition of TADL, this project provides new algorithms, tools, and a methodology to model advanced timing information at different levels of abstraction. The use cases and validators indicate that the project results are in compliance with the AUTOSAR-based tool chain [15]. Since this project is recently finished, it may take some time for its results to become mature and find their way in the industrial use. Arcticus Systems has been involved in TIMMO-2-USE project as one of the industrial partners.

In conclusion, TIMMO methodology and TADL focus on expressing timing information. They are initiatives to annotate AUTOSAR with a timing model. This will be hard to accomplish all the way since AUTOSAR aims at hiding implementation details of execution environment and communication through the VFB. At the modeling level, there is no information in AUTOSAR to express low-level details, e.g., linking information. These details are necessary to extract the timing model from the architecture. There is no focus in this initiative on how to extract this information from the model or perform timing analysis or synthesize the run-time framework. In our view, timing model means extracting enough information to be able to perform certain kind of timing analysis, e.g., end-to-end response-time analysis.

2.2.3. ProCom

ProCom [20], developed as part of a research project at Mälardalen University, is a two-layered component model for the development of distributed embedded systems. At the upper layer, called ProSys, it models a system with concurrent subsystems that communicate with each other by means of asynchronous messages. At the lower layer, called ProSave, a subsystem is internally modeled in terms of functional components which are implemented as a piece of code, e.g., a C function. ProCom is inspired by RCM and there are a number of similarities between the ProSave modeling layer and RCM:

- components in both ProSave and RCM are passive,
- both models clearly separate data flow from control flow among their components,
- both models use pipe-and-filter style of communication mechanism for components interconnection.

However, ProCom does not differentiate between intra- and inter-node communication which is unlike RCM. ProCom hides communication details, whereas RCM lifts them up to the modeling level. It will be very hard in ProCom to extract the timing model and perform end-to-end timing analysis at the level where it is done in RCM.

2.2.4. COMDES-II

COMDES-II [21], developed at the University of South Denmark, provides a component-based framework for the development of distributed embedded control systems. It models the architecture of a system at two levels. At upper level, an application is modeled as a network of actors that are active components. Actors communicate with each other by sending labeled messages. At the lower level, the functionality of an actor is modeled in terms of Function Blocks which are passive components similar to the SWCs in RCM. Unlike RCM, COMDES-II employs signal-based communication for both intra- and inter-node interactions. COMDES-II does not include explicit components to model network communication. Despite few differences, there are a number of similarities between RCM and COMDES-II. However, it will be very hard in COMDES-II to extract the timing model and perform end-to-end timing analysis at the level where it is done in RCM.

2.2.5. Middleware-based approaches

Object Management Group defined middleware technologies such as Real-Time CORBA, minimum CORBA and CORBA lightweight services for the development of real-time and distributed embedded systems [22]. The run-time framework of Real-time CORBA is heavyweight. On the other hand, RCM has a small run-time footprint, i.e., timing and memory overhead. In RCM, we do the timing analysis and actually synthesize the application as run-time and communication platform efficient as possible. We believe, due to high resource requirements at run-time, Real-time CORBA is not efficiently usable in the type of applications that RCM focuses on.

There are other middleware solutions such as iLand project [23] in which a middleware-based framework is introduced to support predictable and time-bounded reconfiguration at run-time for the service-oriented distributed real-time systems. The methodology in this work supports composition of distributed applications based on the concept of services while taking into account real-time properties and requirements. This work focuses on service-oriented development, whereas our approach is based on component-based development. The framework in [23] relies on run-time mechanisms such as dual-band priority assignment [24] for dynamic resource management and reconfiguration. In [25], a component-based modeling approach is introduced that enables dynamic replacement of components at run-time while preserving the temporal properties of the system. On the other hand, we do not consider dynamic reconfiguration and replacement of components in our work. Most of these techniques have been validated for soft real-time systems in the multimedia-applications domain. However, our approach mostly focuses on hard real-time distributed embedded systems in the vehicular domain.

3. Problem statement

To provide a model- and component-based approach to support communications-oriented development of the systems, we target the following issues.

3.1. Modeling of legacy network communication

In an ideal scenario, it should be possible to automatically generate the communication from the design model for each distributed real-time application. However, this is often not the practice in the industry because of presence of legacy communications and legacy systems. These systems have their own predefined rules for communication. In order to support the modeling of these systems, the implementation details must be abstracted; the communication protocols must be encapsulated and abstracted; and adaptation of a node must be supported when communication rules change (e.g., due to re-deployment in a new system or due to upgrades in the communication system) without affecting its internal component design. This problem can be formulated as: *how to model legacy network communication and allow the use of legacy nodes to support the communications-oriented development processes for component-based distributed real-time embedded systems?*

3.2. Issues concerning the extraction of end-to-end timing model

In order to ensure that the system will behave in a timely manner during its execution, we need to analyze tasks, messages and event chains in distributed transactions and predict the end-to-end delays. For this purpose, the end-to-end timing model should be unambiguously extracted from the modeled application. Moreover, the distributed transactions in the applications should be unambiguously identified, extracted and linked. The distributed transactions may consist of trigger chains, data chains or a combination of both. The first SWC in a trigger chain is triggered independently, while the rest of the SWCs are triggered by their respective predecessors as shown in Figure 3 (a). Whereas, each SWC in a data chain is triggered independently as shown in Figure 3 (b). A mixed chain is a combination of both trigger and data chains as shown in Figure 3 (c). The end-to-end timing model should include linking and mapping information of all these chains. The model should also identify the type of each chain because different timing constraints are specified on different types of chains [26]. Furthermore, data chains require different end-to-end timing analysis compared to trigger chains [10].

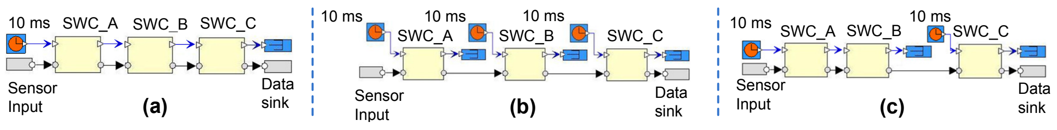


Figure 3: Example of (a) Trigger chain (b) Data chain (c) Mixed Chain.

The linking and mapping problem is common in all types of chains. For simplicity, we consider the system which is modeled with only trigger chains as shown in Figure 4. There are two nodes in the system with three SWCs in node A and four SWCs in node B. SWCs communicate with each other by using both inter- and intra-node communication. The intra-node communication takes place via connectors. Whereas, the inter-node communication takes place via a real-time network to which the nodes are connected. One trigger chain that is activated by a clock consists of four SWCs namely SWC1, SWC2, SWC4 and SWC5. We regard this chain as distributed trigger chain because it is distributed over more than one node. It is identified with the solid-line arrow in Figure 4. In this chain, a clock triggers SWC1 which in turn triggers SWC2. SWC2 then sends a signal to the network. This signal is transmitted over the network in a message (frame[11]) and is received by SWC4 at the receiver node. SWC4 processes it and sends it to SWC5. The time elapsed between the event trigger at input of the task corresponding to SWC1 and production of the response of the task corresponding to SWC5 is referred to as the holistic or end-to-end response time of the distributed chain and is identified in Figure 4. The second distributed trigger chain that is activated by an external event consists of three SWCs namely SWC3, SWC6 and SWC7. It is identified by the dashed-line arrow in Figure 4.

There may not be direct triggering connections between any two neighboring SWCs in the chain which is distributed over more than one node, e.g., SWC2 and SWC4 in Figure 4. In this case, SWC2 communicates with SWC4 by sending signals over the network. Here, the problem is that when a trigger signal is produced by SWC2, it may not be sent straightaway as a message to the network. A message may combine several signals, and hence, there may be some waiting time for the signal to be sent to the network. The message may be sent periodically or sporadically or by any other rule defined by the underlying network protocol. When these trigger chains are modeled using the component-based approach, it is not straightforward to link them to extract the end-to-end timing model. For example, if a message is received at node B then the following information should be available to correctly link the received message in the chain: the ID of the sender node; the ID of the task corresponding to SWC that generated this message; the ID of the destination node; and the ID(s) of the task(s) corresponding to SWC(s) that should receive this message. In order to get a bounded end-to-end delay, a more important question is when and who triggers the destination SWC when a message is received at the destination node.

[11]We use the terms message and frame interchangeably because we only consider messages that fit into one frame.

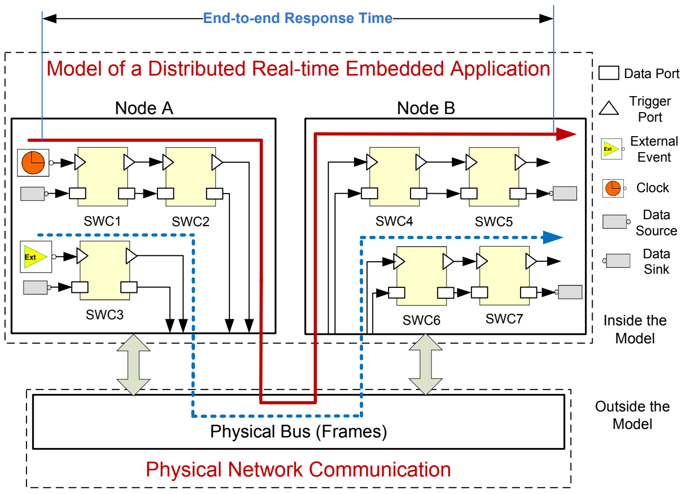


Figure 4: Example of distributed trigger chains.

The existing modeling components in RCM do not provide enough support to link and extract the corresponding timing information of distributed chains. Therefore, special objects in the component technology are needed to provide the linking information of distributed chains to extract end-to-end timing information. Further, there is a need to model mapping between signals and messages and vice versa. SWCs inside a node communicate via signals, whereas they communicate via messages if located on different nodes. Moreover, there is a need to model exit and entry points for RCM models. An exit point is where a message (data) leaves the model and is transmitted according to the protocol-specific rules of the network. Similarly, an entry point is where a message enters the model from the model of the network or any other model. The reason for the need to model exit and entry points for RCM models is to get the bounded delays for distributed chains. The model of entry and exit points will support the use of nodes developed using RCM with the nodes developed by other component technologies.

The problem discussed in this subsection can be formulated as: *how to extract end-to-end timing models from component-based distributed real-time embedded systems that are built using the communications-oriented development processes?* We believe that the issues discussed in this subsection may occur during the development of any other component model for distributed real-time embedded systems that uses the pipe-and-filter communication mechanism for components interconnection, e.g., ProCom [20] and

COMDES [21]. The problem of linking distributed chains may also exist in any type of "inter-model signaling", where a signal leaves one model (e.g., a node, or a core, or a process) and appears again in some other model.

3.3. Modeling and timing analysis of "outside traffic"

The problem arises when the requirements dictate the modeling and end-to-end timing analysis of the system at a stage where the models of some ECUs may not be available. However, the signals and messages which these missing ECUs are supposed to send and receive have been decided. In such a system, the network is assumed to contain "outside traffic", i.e., the messages whose sender nodes are not developed yet. This "outside traffic" could come from external services (e.g., V2V, V2I, and other cloud-based applications), from legacy nodes that lack proper behavior models, or from crude preliminary models of nodes that have not been completely modeled yet. Regardless of source, it is important to be able to analyze end-to-end timing behavior of vehicle internal functions, while taking into account the outside generated traffic. Similarly, the available ECUs may send messages via network to the nodes that will be available at a later stage. Some reasons behind these requirements are to support design space exploration, allow fine tuning of the system with respect to real-time requirements and detection of timing errors.

There exist timing dependencies among messages and their sender and receiver tasks. A message inherits some timing properties from its sender task, e.g., transmission type and period. If the sender task is triggered periodically then the message it sends is also periodic. Further, the message inherits period from its sender task. Similarly, if the sender task is activated sporadically then the corresponding message is sporadic and the message inherits *inhibit time* from the sender task. The inhibit time is the minimum amount of time that should elapse between two consecutive transmissions of a sporadic message. In the case of mixed transmission mode, the message inherits both period and inhibit time from its sender [9]. When the systems are analyzed, each message is assumed to inherit the release jitter from its sender (attribute inheritance [27]). For the messages whose sender tasks are unknown (because the sender ECUs are not available yet or the network traffic is generated from outside of the model), these properties must be extracted in the timing model. Otherwise, the end-to-end timing analysis of these systems cannot be performed. The problem discussed in this subsection can be formulated as: *how to model and timing analyze the applications that contain network traffic originating from outside of the system?*

4. Modeling of legacy network communication

We introduce a new modeling entity the Network Specification to represent the model of communication in a physical network. In order to abstract the implementation of communications in a node, we propose two special-purpose modeling entities namely Out and In Software Circuits for each frame that a node sends to and receives from the network respectively.

4.1. Network Specification (NS)

It is the model representation of a physical network. There is one NS for each network protocol. It consists of two parts, one is independent of the underlying communication protocol while the other is protocol dependent. The protocol-independent part defines messages and the data-elements mapped to them. A message is an entity that is used to send information from one node to another via network. Moreover, the protocol-independent part of the NS describes message properties such as a message ID, a unique sender node ID, a list of receiver nodes IDs and an ordered set of signals included in the message. For example, a signal in RCM has a name, data type, resolution and real-time properties. The protocol-independent part of NS also contains the list of nodes in the system.

The protocol-dependent part of the NS is uniquely defined for each protocol, e.g., it will be different for different high-level protocols for Controller Area Network (CAN) [28] such as CANopen [29], Hägglunds Controller Area Network (HCAN) [30] and CAN for Military Land Systems domain (MilCAN) [31]. It defines the behavior semantics of each message according to the network communication protocol. It contains complete information of all frames which are sent to and received from the network. Moreover, it describes the frame properties. A frame is a formatted sequence of bits that is actually transmitted over the network. In RCM, a frame is a collection of RCM signals.

The frame properties described by the protocol-dependent part of the NS (e.g., for the CANopen protocol) include an identifier (a reference to the corresponding message in the protocol-independent part), a priority, a transmission type (e.g., different types of message transmission in the CANopen protocol), a sender node ID, a list of receiver nodes IDs, whether a frame is an IN frame or an OUT frame, a period (period with which a message is sent in the case of periodic transmission), an inhibit time (minimum time between successive transmission of a message in the case of one of the asynchronous transmission types in CANopen), SYNC period (time between SYNC messages sent by the CANopen SYNC master), and real-time requirements (e.g., message deadline). Moreover, it also specifies the bus speed. The transmission type of a frame can be periodic, sporadic or mixed (transmitted periodically as well as sporadically) [9].

In RCM, the components inside a node communicate with each other via data and control signals. However, if a component on one node communicates with a component on another node via a network then the signals are packed into the frames. The frames are then transmitted over the network. Here, some questions arise concerning the communication in the network. How are signals mapped to messages? How are the signals packed into the frames? How are the signals encoded into the frames at the sender node? How are the signals decoded from the frames and sent to the respective SWCs at the receiver node? How many signals are there in each frame? All rules concerning the answers to these questions are specified in the Signal Mapping. The Signal Mapping is a unique object for each protocol for network communication and is an integral part of the protocol-dependent part of the NS. The Signal Mapping also describes the length of each signal in a frame, the type of signal encoding in a frame (e.g., signed or unsigned 2's complement), and maximum age of a signal guaranteed by the sender.

4.2. Out Software Circuit (OSWC)

It is the model representation of signals in an outgoing message to the network. Basically, it is a Software Circuit which denotes the data that leaves the model. There is one OSWC in a node for every outgoing frame on the network. Each OSWC describes the signals that can be sent in a particular frame. A frame contains zero or more signals. The OSWC has only one trigger in-port and at least one data in-port. Each data in-port is associated with one signal in the NS. Therefore, the number of data in-ports may vary depending upon the number of signals packed in the frame. The OSWC has no data and trigger out-ports. It uses protocol-specific rules, specified in the protocol-specific part of the NS, while encoding data and mapping signals to a frame. In this way, it provides a clear abstraction to the SWCs that send signals to one of its data in-ports. Thus, SWCs are kept unaware of the protocol-specific details such as signal-to-frame mapping, data type encoding and transmission patterns of frames. The conceptual model of the OSWC is illustrated in Figure 5 (a), whereas its RCM model is shown in Figure 5 (b).

4.3. In Software Circuit (ISWC)

It is the model representation of signals in an incoming message from the network. Basically, it is a Software Circuit which denotes the data that enters the model. There is one ISWC component in a node for every frame received from the network. It describes all the signals that are contained in a received frame that is associated to it. The ISWC component has one trigger out-port that produces a trigger signal every time the component is executed. There is at least one data out-port in the ISWC. Each data out-port is associated with one signal in the NS. Therefore, the number of data out-ports may vary depending upon the number of signals contained in the received frame. There are no data in-ports in the ISWC. It has one trigger in-port which is triggered every time a frame arrives from the network. When a frame arrives at a node, the physical network drivers and protocol-specific implementation of the ISWC extract the signals (zero or more signals per frame) and encode their data in the RCM data type. When the signal(s) is delivered, the data is placed on the data port which is connected to the data in-port of the destination SWC (the linking and mapping information is provided in the NS), and the corresponding trigger port is triggered. Figures 5 (a) and 5 (a) graphically illustrate the conceptual and RCM model of the ISWC respectively. It should be noted that the developer can specify timing parameters such as execution times for the OSWC and ISWC.

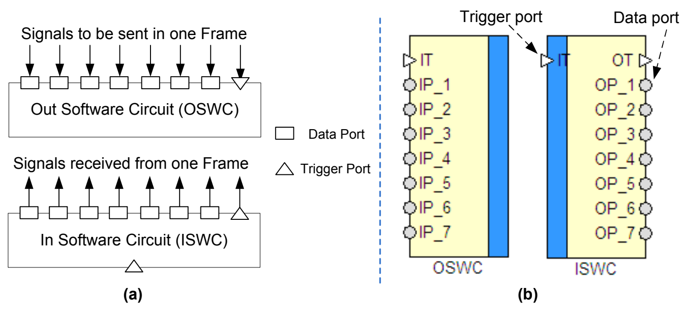
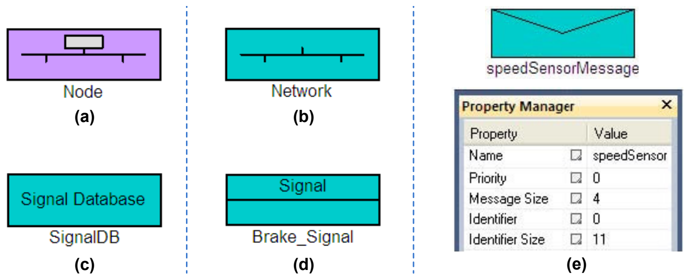


Figure 5: OSWC and ISWC components (a) conceptual models (b) models in RCM.

The models of a node, a network, a signal database and a signal in RCM are shown in Figures 6 (a), (b), (c) and (d) respectively. The signal database object corresponds to the Signal Mapping which is part of the NS (as discussed in Section 4.1). It contains all the signals that are sent over the network. Each signal in the signal database is linked to one or more messages. The model of a message along with the list of user-defined properties is shown in Figure 6 (e). The developer specifies only the name, priority, data size, value and type of identifier (in the case of CAN) of the message. The message automatically inherits jitter, transmission type and period or inhibit time or both from the sender OSWC.



Property	Value
Name	speedSensor
Priority	0
Message Size	4
Identifier	0
Identifier Size	11

Figure 6: RCM models of (a) node, (b) network, (c) signal database, (d) signal, (e) message and its attributes.

Consider an example of a node in a distributed real-time embedded application modeled with the introduced objects as shown in Figure 7. Let the node be connected to the CAN network. The upper half of Figure 7 represents the model of a node, whereas the lower half represents the physical communication including the CAN controller and network. There are two grey boxes outside the model called CAN SEND and CAN RECEIVE that are placed just below the sets of OSWCs and ISWCs respectively. These grey boxes are specific for each network protocol. The frames that leave the model (sent to CAN SEND) are denoted by S (Send), e.g., $S1$, $S2$ and $S3$. Similarly all the frames that enter the model (received from CAN RECEIVE) are denoted by R (Receive), e.g., $R1$ and $R2$. All signals that are sent in the frame $S1$ are provided at the data in-ports of OSWC1. These signals are mapped and encoded into $S1$ by OSWC1 according to the protocol-specific information available in the NS. Once the frame is ready, it leaves the model as it is sent to the grey box CAN SEND. In this example, this grey box represents a CAN controller in the node which is responsible for physical transmission of this frame over the network according to the arbitration and communication rules specified by the CAN protocol.

When a frame arrives at the receiving node, it is transferred by the network drivers to the CAN RECEIVE grey box which is responsible for raising an interrupt request and passing the frame to the corresponding ISWC. In this case, the *TrigInterrupt* object in RCM corresponding to the interrupt is connected to the *in-port* of the ISWC. If CAN drivers use polling-based

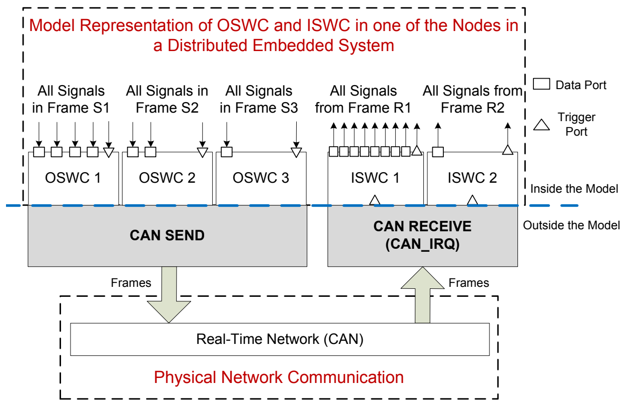


Figure 7: Model of a node using OSWCs and ISWCs for network communication.

processing instead of interrupts then the *in-port* of the ISWC is connected to the clock, where clock period is set equal to the polling period. Upon receiving the frame, the ISWC decodes it, extracts signals from it, places the data on the corresponding data port (connected to the data *in-port* of the destination SWC), and triggers the corresponding trigger port by using the linking information available in the NS. It should be noted that when the OSWC is triggered, it executes the required functionality (e.g., mapping of signals to a message) and then the data (message) is transferred from the RCM model of node to the network controller or another model of communication network. Therefore, OSWC also represents the model of an exit point for RCM models. Similarly, ISWC component represents the model of an entry point for RCM models. Using our new approach, nodes can be developed without explicit knowledge about the communication configuration.

4.4. Automatic generation of the OSWC and ISWC components

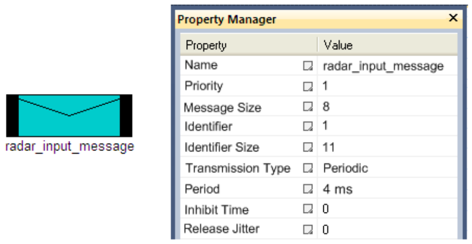
Both OSWC and ISWC can be automatically generated from the NS by a Network Configuration Tool. The input to this tool is the protocol-specific information about the network communication and the linking information of tasks in all distributed chains (i.e., trigger, data and mixed) present in the application. This information is provided from the configuration files that correspond to the NS. The output of this tool is a set of automatically generated OSWCs and ISWCs for each node in the network. This tool also carries out mapping from the NS to the OSWC and ISWC and vice versa. One of the main purpose of our modeling technique was to use legacy components, however, it can also be used the other way around. That is, the protocol-independent part of the NS can be automatically generated from the models of the OSWC and ISWC from an existing system.

4.5. Support for modeling "outside traffic"

A solution to the problem (discussed in Section 3.3) could be using the model of a dummy sender node in place of the ECU that is not available but decisions on the messages it sends are already taken. The only purpose of this node is to encode and pack signals into messages and send them to the network. Similarly, a dummy receiver node can be used to receive messages in place of a missing ECU. Such a solution can be realized in RCM with the models of sender and receiver nodes that contain one OSWC and one ISWC for every message they send and receive respectively. However, this solution is impractical as it adds design complexity to the system. Moreover, it adds an extra modeling and testing overhead on the developer because there are several modeling and specification steps involved when a node is modeled.

The problem can be solved in a better way by introducing a special type of message called stand-alone message. This message supports the modeling of "outside traffic". It does not bear any association with the OSWC component. This means, it does not have any sender task inside the model of the system. However, there is an option for the user to associate this message to any number of ISWCs, i.e., it can be received by any number of tasks. Apart from name, priority, data size, value and type of identifier (user-defined properties for a regular message); it is also possible for the user to specify transmission type and corresponding timing parameters (period, inhibit time or both) for this message. The transmission type of a message is a very important parameter because the network timing analysis is dependent upon it especially in the case of CAN and its high-level protocols. For example, if there are only periodic and sporadic messages in the system then one type of timing analysis is used such as [7]. On the other hand, if there is at least one mixed message in the system then a different timing analysis (i.e., response-time analysis for mixed messages) is used [9, 32, 33].

The user can also specify release jitter for the stand-alone message. The release jitter may either be equal to the difference between the estimated worst- and best-case response times of the sender (belonging to the node that is not available) or zero if these response times cannot be estimated at this stage. The extra user-defined information in the case of stand-alone messages is vital for the network timing analysis, and hence, for the end-to-end timing analysis. The standalone message introduced in Rubus-ICE along with the list of its user-defined properties is shown in Figure 8. The dark vertical stripes on both of its sides differentiate it from the regular message in RCM. This message is treated differently from the regular message at the attribute inheritance step by the holistic timing analysis algorithm [27, 10].



Property	Value
Name	radar_input_message
Priority	1
Message Size	8
Identifier	1
Identifier Size	11
Transmission Type	Periodic
Period	4 ms
Inhibit Time	0
Release Jitter	0

Figure 8: Model of a stand-alone message with the list of user-defined properties.

It should be noted that the list of user-defined properties of a stand-alone message (see Figure 8) is more general and includes user-defined properties of a regular message (see Figure 6 (e)). One may think of using the user-defined properties in Figures 8 consistently for all types of messages. However, this is not practical mainly because of two reasons. First, the timing information extracted from the modeled application may be redundant. That is, the transmission type and corresponding period and inhibit time will be extracted from the user-defined input as well as from the sender task. This redundancy may result in the extraction of ambiguous end-to-end timing model. Information duplication can lead to inconsistency in the model. Second, it will add extra complexity and burden on the developer to specify too much information during the modeling. Our intension is to extract unambiguous end-to-end timing information and keep things as simple as possible for the developer.

5. Extraction of end-to-end timing models

In order to ensure all timing requirements are met, the modeled application should render itself to the end-to-end timing analysis. For this purpose, the end-to-end timing model of the application should be available.

5.1. End-to-end timing model

This model consists of timing properties, requirements and dependencies concerning all tasks, messages, task chains and distributed transactions in the system under analysis. It consists of the following sub models:

1. System timing model
   (a) Node timing model
   (b) Network timing model
2. System linking model

5.1.1. System timing model

This model is composed of node and network timing models.

1) Node timing model. This model contains node-level timing information. We consider the model that is based on the transactional task model (i.e, tasks with offsets) introduced by [34] and later on, extended by many researchers, e.g., [35, 36]. A node, Γ, consists of a set of k transactions $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_i is activated by mutually independent events, i.e., the phasing between the events is arbitrary. The activating events can be a periodic sequence of events with a period T_i. In case of sporadic events, T_i denotes the minimum inter-arrival time between two consecutive events.

There are $|\Gamma_i|$ tasks in a transaction Γ_i. Each task in Γ_i may not be activated until a certain time, called an *offset*, elapses after the arrival of the external event. By task activation we mean that the task is released for execution. A task is denoted by τ_{ij}. The first subscript, i, specifies the transaction to which this task belongs and the second subscript, j, denotes the index of the task within the transaction. A task, τ_{ij}, is defined by the following attributes.

- C_{ij} denotes the worst-case execution time of the task.
- O_{ij} denotes the offset of the task.
- D_{ij} specifies the optional deadline of the task.
- J_{ij} denotes the maximum release jitter.
- B_{ij} represents the maximum blocking time which is the maximum time the task has to wait for a resource that is locked by a lower priority task. In order to obtain the blocking time for a task, a resource sharing protocol, e.g., Stack Resource Policy (SRP) [37] or Priority Ceiling Protocol (PCP) [38], that bounds the blocking time must be used.

- P_{ij} denotes the priority of the task.
- R_{ij} denotes the worst-case response time of the task.

In this model, there are no restrictions placed on offset, deadline or jitter, i.e., they can each be either smaller or greater than the period.

2) Network timing model. This model contains network-level timing information of the system. A network consists of a number of nodes that are connected through a real-time network. Currently, RCM supports CAN and its higher-level protocols such as CANopen, MilCAN and HCAN. However, it can be easily extended to support other protocols such as Flexray [39]. In this model, each message m has the following attributes.

- ID_m denotes a unique identifier.
- $FRAME\_TYPE$ specifies whether the frame is a Standard or an Extended CAN frame.
- $TRANSMISSION\_TYPE$ specifies whether the message is periodic or sporadic or mixed (both periodic and sporadic).
- P_m denotes unique priority.
- C_m specifies the transmission time.
- J_m denotes the release jitter. Usually, it is inherited from the task that queues m.
- s_m denotes the data payload in each message. It ranges from 0 to 8 bytes in a CAN message.
- T_m specifies the period of a message in the case of periodic transmission. For a sporadic message, $MINT_m$ is used which refers to the minimum time that should elapse between the transmission of any two messages. For a mixed message, both T_m and $MINT_m$ are specified.
- B_m denotes the blocking time of the message. It refers to the maximum amount of time during which this message can be blocked by the lower priority messages.
- R_m denotes the worst-case response time.

5.1.2. System linking model

In distributed embedded systems, there exist chains of components (tasks) that may be distributed over more than one node. A task chain consists of a number of tasks that are in a sequence and have one common ancestor. A task in a chain may receive trigger, data or both from its predecessor. Two neighboring tasks in a distributed transaction may reside on two different nodes, while the nodes communicate with each other via network. When there are chains in the system, the end-to-end timing model should not only contain timing related information but also the linking information among all tasks and messages within each distributed chain. All mapping and linking information of distributed chains is extracted into the system linking model.

5.2. Method to unambiguously extract and link distributed chains

We provide a method to identify, extract and link distributed chains from the RCM models of component-based distributed real-time systems.

5.2.1. Extraction of unambiguous timing information

The end-to-end timing information that is extracted from all tasks, messages and distributed chains can be divided into two categories. The first category corresponds to the timing information that is provided by the user in the modeled application, e.g., most of the task and message attributes discussed in Section 5.1.1. Whereas, the second category corresponds to the timing information which is not directly provided by the user but has to be extracted from the modeled application. For example, release jitter for a message. It is inherited as the difference between the worst- and best-case response times of the sending task. Similarly, message transmission type, message period and inhibit times are often not specified by the user, rather they are inherited from the sender tasks. Hence, these parameters must be extracted from the modeled application and added in the timing model. We assign period or inhibit time to the message which is equal to the period or inhibit time of its sender task. If the sender task is activated by a clock, we assign periodic transmission type to the message. Similarly, if the sender task is activated by a sporadic event then we assign sporadic transmission type to the message. However, if the sender task is triggered by both a clock and a sporadic event then transmission type of the message is considered as mixed. These assignments are important because a message is analyzed differently based on its transmission type.

5.2.2. Identification of trigger, data and mixed distributed chains

In order to unambiguously identify each individual chain, we attach *trigger dependency* attribute with each task. This attribute is part of the data structure of tasks in the timing model. Basically, it extracts the triggering information for the corresponding task. If a task is triggered by an independent source, such as a clock, then this attribute is set to "independent". On the other hand, if the task is triggered by another task then this parameter is set to "dependent". A precedence constraint is also specified on this task in the case of dependent triggering.

An iterative method determines whether the triggering of every two neighboring tasks in a chain is dependent or independent of each other by testing the value of corresponding *trigger dependency* attributes. If this attribute for all tasks (except the first) is "dependent", the chain is identified as a trigger chain. On the other hand, if this attribute for each and every task in the chain has "independent" value, the chain is identified as a data chain. However, if this attributed has "independent" value for some tasks in the chain while "dependent" value for the rest, the chain is regarded as a mixed chain. This method is applied to all chains in the system.

5.2.3. Linking of distributed trigger, data and mixed chains

The method for linking distributed chains is built upon the modeling approach that we discussed in the previous section. This method treats all types of chains in a similar fashion. The linking information for all distributed chains in the modeled application is provided in the NS. We assign references to trigger *in-ports* of OSWCs and the trigger *out-ports* of ISWCs along the same distributed chain. These references are contained in a reference array. There is one reference array for each distributed chain. The ordering of references within the array corresponds to the ordering of the components (OSWC/ISWC) along the trigger chain. That is, the first reference in the array corresponds to the trigger port of the first component in the chain, and so on. The reference arrays corresponding to all distributed chains in the system are specified in the NS. Consider the example shown in Figure 9. There are three SWCs in each node. The nodes are connected to the CAN network. There are two trigger chains in the system that are distributed over two nodes. These chains are identified as TC_1 and TC_2 as shown below.

- $TC_1 : SWC1 \rightarrow SWC2 \rightarrow OSWC\_A1 \rightarrow ISWC\_B1 \rightarrow SWC4 \rightarrow SWC5$.

- $TC_2 : SWC6 \rightarrow OSWC\_B1 \rightarrow ISWC\_A1 \rightarrow SWC3$.

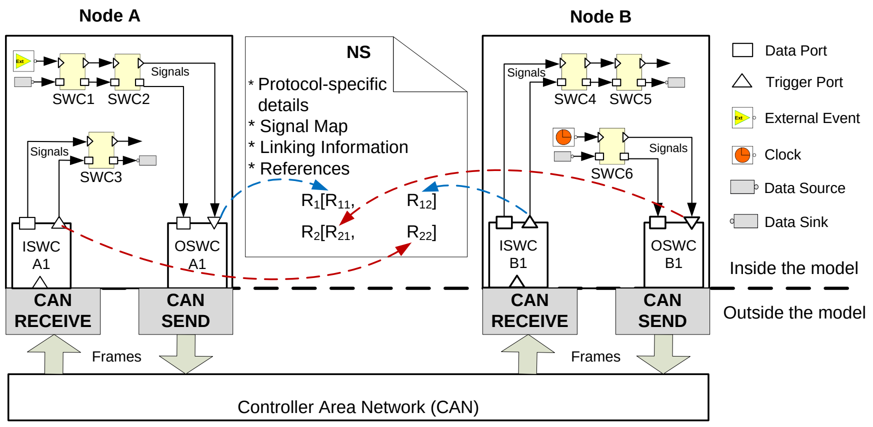


Figure 9: Demonstration of linking distributed chains.

The trigger chain TC_1 is triggered by an external event, whereas TC_2 is triggered by a clock. There is a reference array R_1 that contains references to all the OSWC and ISWC components in TC_1. R_{11} refers to the trigger *in-port* of OSWC A1 in Node A, whereas R_{12} refers to the trigger *out-port* of ISWC B1 in node B. Similarly, a reference array R_2 is stored in the NS that contains references to all OSWCs and ISWCs in TC_2. R_{21} refers to the trigger *in-port* of OSWC B1 in Node B, whereas R_{22} refers to the trigger *out-port* of ISWC A1 in node A. In this way, all the neighboring components located in different nodes within a distributed trigger chain can be linked.

5.3. Extraction of end-to-end timing model in Rubus-ICE

In Rubus-ICE, the application is modeled in the Rubus Designer tool. It is then compiled to the Intermediate Compiled Component Model (ICCM). Apart from the compiled component model, the ICCM file also includes timing and linking information of the modeled system. The timing model that is implemented in the Rubus Analysis Framework, extracts the required timing and linking information from the ICCM file[12] as shown in Figure 10. From the extracted model, the Rubus Analysis Framework performs the end-to-end timing analysis and then provides the results, i.e., response times of

[12]in XML format

individual tasks, response times of network messages, end-to-end response times and delays of distributed chains, network utilization, etc., back to the Rubus-ICE tool suite.

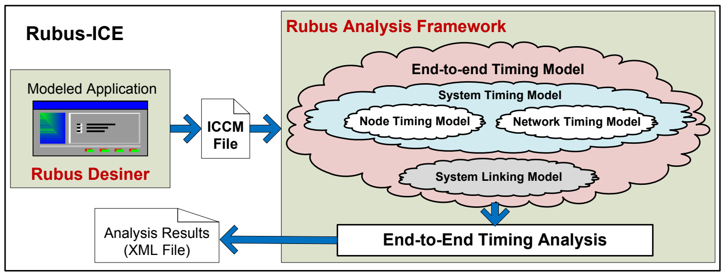


Figure 10: Extraction of the end-to-end timing model in Rubus-ICE tool-suite.

6. Automotive application case study

We provide a proof of concept for the modeling technique and timing model extraction method that we implemented in Rubus-ICE by conducting an automotive-application case study. We model the next-generation Adaptive Cruise Control system with RCM and analyze it with the Holistic Response Time Analysis (HRTA) plug-in in Rubus-ICE.

6.1. Next-generation adaptive cruise control system

The Adaptive Cruise Control (ACC) system is an automotive feature that allows a vehicle to automatically adapt itself to the traffic environment to maintain a steady speed to the value that is preset by the driver. Often, it uses a radar to create a feedback of distance to, and velocity of, the preceding vehicle. It also communicates (cooperates) with the surrounding vehicles. Moreover, it receives traffic related cloud-services such as community map and turn-by-turn navigation services from outside of the vehicle. Based on the feedback, it either reduces the vehicle speed to keep a safe distance and time gap from the preceding vehicle or accelerates the vehicle to match the preset speed specified by the driver. The ACC system may be modeled with four nodes namely Cruise Control (CC), Engine Control (EC), Brake Control (BC) and User Interface (UI) [40]. Figure 11 shows the block diagram of the ACC system. The nodes communicate with each other via CAN network.

Assume that the models of EC and BC nodes are available while the models of CC and UI nodes will be available at a later stage. However, the

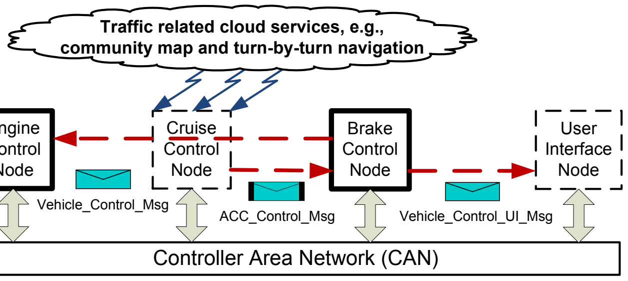


Figure 11: Block diagram of Adaptive Cruise Control System.

decisions about network communication have been made. There is one stand-alone message "ACC_Control_Msg" in the system that is assumed to be sent by the CC node (not available yet). This message is received by the BC node as shown by the dashed-line arrow in Figure 11. The BC node sends two messages over the network. First message "Vehicle_Control_UI_Msg" is sent to the UI node (not available yet). Whereas, the second message "Vehicle_Control_Msg" is sent to the EC node. It should be noted that the dashed-line arrows represent virtual communication while the actual message transmission takes place through the CAN network.

The UI node reads driver inputs and shows status messages and warnings on the display screen. The inputs are acquired by means of switches and buttons mounted on the steering wheel. These include Cruise Switch input that corresponds to ON/OFF, Standby and Resume states for ACC; Set Speed input (desired cruising speed set by the driver) and desired clearing distance from the preceding vehicle. This node receives linear and angular speed, status of manual brake sensor, and status messages and warnings to be displayed on the screen from the BC node via the CAN network.

The CC node analyzes the state of the cruise control switch. If the switch is in the ON state then the cruise control functionality is activated. It reads input from a proximity sensor (e.g., radar) and processes it to determine the presence of the vehicle in front of it. It also receives V2V communication and navigation information from outside of the vehicle as shown in Figure 11. Moreover, it processes the radar signals along with the other information, such as vehicle speed, to determine its distance to the preceding vehicle. It sends a CAN message to the BC node. The message carries the control information that is used to adjust the speed of the vehicle with respect to the cruising speed or clearing distance from the preceding vehicle.

The EC node is responsible for controlling vehicle speed by adjusting the engine throttle. It reads sensor input and determines engine torque. It receives a CAN message (from the BC node) that includes information regarding vehicle speed and status of the manual brake sensor. Based on the received information, it determines whether to increase or decrease engine throttle. It then sends new throttle position to the actuators that control the engine throttle.

The BC node receives signals from the break sensors. It also receives the status from the linear and angular speed sensors which are connected to the wheels. It receives a CAN message that includes control information processed by the CC node. Based on this feedback, it computes new vehicle speed. It produces control signals and sends them to the brake actuator and brake light controller. It also sends CAN messages to the EC and UI nodes that carry information regarding status of manual brake, vehicle speed and angular speed.

6.2. Modeling of the ACC system in Rubus-ICE

The RCM model of ACC system is shown in Figure 12. Since, CC and UI nodes are not available at this stage, there are models of only CC and EC nodes in the application. The model of CAN bus is also shown. The selected speed of CAN bus is 500 kbps. The standard CAN frame format is selected.

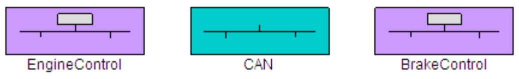
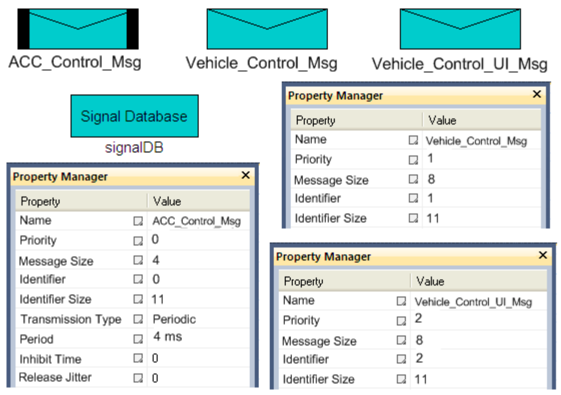


Figure 12: Adaptive Cruise Control System modeled with RCM.

There are three CAN messages in the system ACC_control_Msg, Vehicle_Control_Msg and Vehicle_Con-trol_UI_Msg as shown in Figure 13. ACC_control_Msg is the only stand-alone message. The senders and receivers of all messages are shown in Figure 11. A signal database is also shown in Figure 13. It corresponds to the NS (see Section 4) and contains all the signals that are sent over the network. Each signal in the signal database is linked to one or more messages. The user-defined properties of all messages are also visible in Figure 13.

The internal architecture of the BC node is shown in Figure 14. It is modeled with five SWCs (SpeedSensorInput, ManualBrakeSensorInput, RMPSensorInput, SetBrakeSignal_SWC and SetBrakeLightSignal_SWC), one



Property Manager

Property	Value
Name	ACC_Control_Msg
Priority	0
Message Size	4
Identifier	0
Identifier Size	11
Transmission Type	Periodic
Period	4 ms
Inhibit Time	0
Release Jitter	0

Property Manager

Property	Value
Name	Vehicle_Control_Msg
Priority	1
Message Size	8
Identifier	1
Identifier Size	11

Property Manager

Property	Value
Name	Vehicle_Control_UI_Msg
Priority	2
Message Size	8
Identifier	2
Identifier Size	11

Figure 13: CAN messages and signal database modeled with RCM.

ISWC component (ACC_control_Msg_ISWC), two OSWC components (Vehicle_Control_Msg_OSWC and Vehicle_Control_UI_Msg_OSWC) and one assembly (Brake_Control). An assembly in RCM is a container for various software items. The Brake_Control assembly is further modeled with two SWCs BrakeInputInfoProcessing and BrakeController as shown in Figure 15. Each component is named according to its functional behavior, e.g., the ACC_control_Msg_ISWC component is responsible for receiving ACC_control _Msg to the network.

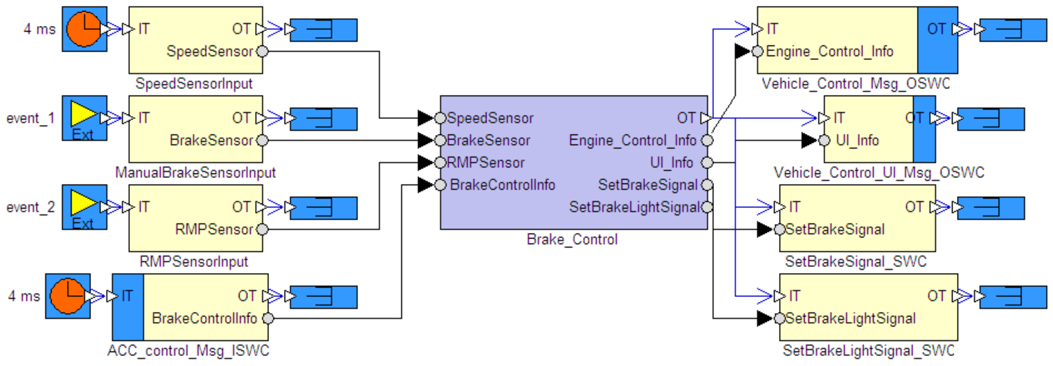


Figure 14: RCM model of the Brake Control node.

The internal architecture of EC node is shown in Figure 16. It is modeled with two SWCs (EngineTorqueInput and SetThrottlePosition), one ISWC component (Vehicle_Con-trol_Msg_ISWC) and one assembly (Engine_Control) as shown in Figure 17. The Engine_Control assembly is further modeled with

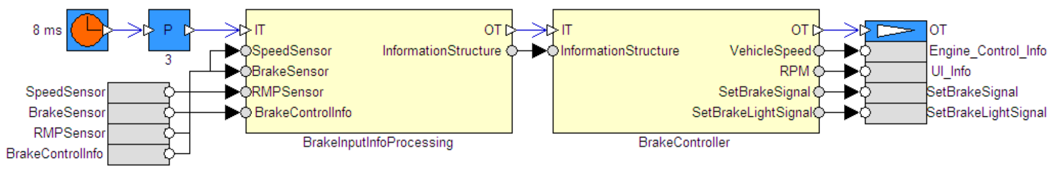


Figure 15: Internal model of Brake_Control assembly in RCM.

two SWCs EngineInputInformationProcessing and ThrottleControl.

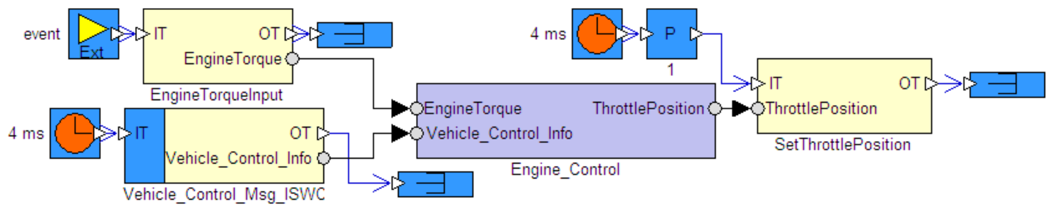


Figure 16: RCM model of the Engine Control node.

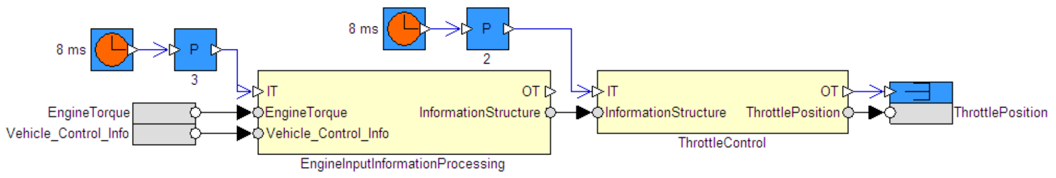


Figure 17: Internal model of Engine_Control assembly in RCM.

6.3. Holistic response-time analysis of the ACC system

The HRTA plug-in in Rubus-ICE calculates the response times of all messages and tasks as well as end-to-end or holistic response times of Distributed Transactions (DTs). We refer the reader to [41] for the details about the holistic response-time analysis. We focus on the analysis of the following DTs.

1. DT_1: ACC_control_Msg → ACC_control_Msg_ISWC → BrakeInputInfoProcessing → BrakeController → SetBrakeSignal_SWC
2. DT_2: ACC_control_Msg → ACC_control_Msg_ISWC → BrakeInputInfoProcessing → BrakeController → SetBrakeLightSignal_SWC
3. DT_3: SpeedSensorInput → BrakeInputInfoProcessing → BrakeController → Vehicle_Control_UI_Msg_OSWC → Vehicle_Control_UI_Msg

Table 1: Calculated holistic response times of distributed transactions under analysis.

Distributed Transaction	DT_1	DT_2	DT_3	DT_4
Holistic Response Time (μs)	520	555	1250	830

4. DT_4: SpeedSensorInput $\rightarrow$ BrakeInputInfoProcessing $\rightarrow$ BrakeController $\rightarrow$ Vehicle_Control_Msg_OSWC $\rightarrow$ Vehicle_Control_Msg $\rightarrow$ Vehicle_Control_Msg_ISWC $\rightarrow$ EngineInputInformationProcessing $\rightarrow$ ThrottleControl $\rightarrow$ SetThrottlePosition

Both distributed transactions DT_1 and DT_2 are initiated by the stand-alone message ACC_control_Msg. They terminate by producing the control signals for brake actuators and brake light controllers. DT_3 starts with the speed sensor input in the BC node and terminates by sending the message destined for the UI node. Finally, DT_4 initiates with the speed sensor input in the BC node and terminates by producing a control signal for engine throttle controller in the EC node. The worst-case execution times of all SWCs are selected in the range of $(20 - 200)\mu s$. The holistic response times of these distributed transactions are shown in Table 1. In order to interpret the calculated results, consider the holistic response time of DT_4 in Table 1. It indicates that the maximum time required from sensing the variation in the vehicle speed to controlling the engine throttle actuator is $830\mu s$. The holistic response times of other DTs can be interpreted in a similar fashion.

7. Conclusion and future work

We introduced a new technique to provide a model- and component-based support for communications-oriented development of vehicular distributed real-time embedded systems. The proposed approach allows modeling of legacy network communication and abstracts the implementation and configuration of communications in the component-based systems. It explicitly enables the communication capabilities of a node, but hides the implementation or protocol details. Moreover, it allows model- and component-based development of new nodes that are deployed in legacy systems that use predefined communication rules. The proposed approach also enables adaptation of a node when communication rules change without affecting its internal architecture. In order to support end-to-end timing analysis, we presented a method to extract end-to-end timing models from the systems that are developed using the proposed approach. In this context, we discussed and resolved various issues. Our technique also supports modeling and timing analysis of distributed applications that contain network traffic originating from outside of the system, e.g., cloud-based applications. As a proof of concept, we implemented this technique in the existing industrial tool Rubus-ICE, and validated it by modeling and analyzing the automotive application-case study. We believe, this technique may be suitable for several other model- and component-based development technologies that use a pipe-and-filter style for component interconnection, e.g., ProCom and COMDES. Moreover, it can be used for any type of “inter-model signaling”, where a signal leaves one model (e.g., a node, or a core, or a process) and appears again in some other model. We believe, the tools implementing our technique may prove helpful for the software development organizations in the vehicular domain to decrease the costs for software development, configuration and testing.

An interesting future research direction is to bridge the semantic gap between functional models (expressed in standard languages as EAST-ADL and/or proprietary languages such as Simulink or Statemate) and execution models (expressed in proprietary languages like RCM). It would also be interesting and useful to facilitate the exchange of timing analysis models and tools between RCM and several other component models and tools.

Acknowledgement

This work is supported by the Swedish Knowledge Foundation (KKS) within the projects FEMMVA and EEMDEF, the Swedish Research Council (VR) within project TiPCES, and the Strategic Research Foundation (SSF) with the centre PROGRESS. The authors would like to thank the industrial partners Arcticus Systems, BAE Systems Hägglunds and Volvo Construction Equipment (VCE), Sweden.

References

[1] J. Mäki-Turja, K. Hänninen, M. Nolin, Towards efficient development of embedded real-time systems, the component based approach, in: International Conference on Embedded Systems & Applications (ESA), 2006.

[2] TIMMO Methodology, Version 2, Deliverable 7, Oct. 2009.

[3] S. Mubeen, M. Sjödin, J. Mäki-Turja, Supporting early modeling and end-to-end timing analysis of vehicular distributed real-time applications, in: Real-Time and Distributed Computing in Emerging Applications (REACTION) Workshop located at 33rd IEEE Real-Time Systems Symposium (RTSS), 2012.

[4] K. Hänninen et.al., The Rubus Component Model for Resource Constrained Real-Time Systems, in: 3rd IEEE International Symposium on Industrial Embedded Systems, 2008.

[5] Rubus-ICE: Integrated component Development Environment, 2013. http://www.arcticus-systems.com.

[6] J. Mäki-Turja, , M. Nolin, Tighter response-times for tasks with offsets, in: Real-time and Embedded Computing Systems and Applications Conference (RTCSA), 2004.

[7] K. Tindell, H. Hansson, A. Wellings, Analysing real-time communications: controller area network (CAN), in: Real-Time Systems Symposium (RTSS) 1994, pp. 259 –263.

[8] R. Davis, A. Burns, R. Bril, J. Lukkien, Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised, Real-Time Systems 35 (2007) 239–272.

[9] S. Mubeen, J. Mäki-Turja, M. Sjödin, Extending schedulability analysis of controller area network (CAN) for mixed (periodic/sporadic) messages, in: 16th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), 2011.

[10] S. Mubeen, J. Mäki-Turja, M. Sjödin, Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study, in: Computer Science and Information Systems, vol. 10, no. 1, pp 453-482, January 2013. ISSN: 1361-1384.

[11] Microsoft, Distributed Component Object Model (DCOM), http://msdn.microsoft.com/en-us/library/Aa286561.

[12] OMG, Common Object Request Broker Architecture (CORBA), Version 3.1, 2008, OMG Group, 2008.

[13] L. DeMichiel, Sun Microsystems, Enterprise JavaBeans Specification, Version 2.1, Sun Microsystems (2002).

[14] AUTOSAR Techincal Overview, Version 2.2.2. AUTOSAR – AUTomotive Open System ARchitecture, Release 3.1, The AUTOSAR Consortium, Aug., 2008, http://autosar.org.

[15] Mastering Timing Information for Advanced Automotive Systems Engineering. In the TIMMO-2-USE Brochure, 2012. Available at: http://www.timmo-2-use.org/pdf/T2UBrochure.pdf.

[16] TADL: Timing Augmented Description Language, Version 2, Deliverable 6, October 2009, The TIMMO Consortium.

[17] The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems, 2010, OMG Group, 2010.

[18] EAST-ADL Domain Model Specification, Deliverable D4.1.1, 2010, http://www.atesst.org/home/liblocal/docs/ATESST2_D4.1.1_EAST-ADL2-Specification_2010-06-02.pdf.

[19] TIMMO-2-USE, http://www.timmo-2-use.org/.

[20] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, I. Crnkovic, A Component Model for Control-Intensive Distributed Embedded Systems, in: 11th International Symposium on Component Based Software Engineering (CBSE), 2008, Springer, 2008, pp. 310–317.

[21] X. Ke, K. Sierszecki, C. Angelov, COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems, in: 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2007, pp. 199 –208.

[22] Catalog of Specialized CORBA Specifications. OMG Group, http://www.omg.org/technology/documents/.

[23] M. Garcia Valls, I. Lopez, L. Villar, iland: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems, IEEE Transactions on Industrial Informatics 9 (2013) 228–236.

[24] M. Garcia Valls, A. Alonso, J. Antonio de la Puente, A dual-band priority assignment algorithm for dynamic qos resource management, Future Generation Computer Systems 28 (2012) 902–912.

[25] J. C. Romero, M. Garcia Valls, Scheduling component replacement for timely execution in dynamic systems, Software- Practice and Experience (2013).

[26] N. Feiertag, K. Richter, J. Nordlander, J. Jonsson, A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics, in: Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS), 2008.

[27] K. Tindell, J. Clark, Holistic schedulability analysis for distributed hard real-time systems, Microprocess. Microprogram. 40 (1994) 117–134.

[28] Road Vehicles interchange of digital information controller area network (CAN) for high-speed communication, ISO Standard-11898, Nov. 1993.

[29] NIKHEF, Amsterdam. CANopen high-level protocol for CAN-bus, Version 3.0, 2000, http://www.nikhef.nl/pub/departments/ct/po/doc /CANopen.pdf.

[30] Hägglunds Controller Area Network (HCAN), Network Implementation Specification, BAE Systems Hägglunds, Sweden (internal document), 2009.

[31] MilCAN (CAN for Military Land Systems domain), http://www.milcan.org/.

[32] S. Mubeen, J. Mäki-Turja and M. Sjödin, Response-Time Analysis of Mixed Messages in Controller Area Network with Priority- and FIFO-Queued Nodes, in: 9th IEEE International Workshop on Factory Communication Systems (WFCS), 2012.

[33] S. Mubeen, J. Mäki-Turja, M. Sjödin, Worst-case response-time analysis for mixed messages with offsets in controller area network, in: 17th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), 2012.

[34] K. Tindell, Adding Time-Offsets to Schedulability Analysis, Technical Report, Department of Computer Science, University of York, England, 1994.

[35] J.C. Palencia and M. Gonzalez Harbour, Schedulability Analysis for Tasks with Static and Dynamic Offsets, Real-Time Systems Symposium, IEEE International (1998) 26.

[36] J. Mäki-Turja and M. Nolin, Efficient implementation of tight response-times for tasks with offsets, Real-Time Syst. 40 (2008) 77–116.

[37] T. P. Baker, Stack-based scheduling for realtime processes, Real-Time Systems 3 (1991) 67–99.

[38] L. Sha, R. Rajkumar, J. Lehoczky, Priority inheritance protocols: An approach to real-time synchronization, IEEE Transactions on Computers 39 (1990) 1175–1185.

[39] FlexRay Consortium, FlexRay Communications System - Protocol Specication Version 2.1 Revision A, December 2005, http://www.flexray.com/.

[40] Adaptive Cruise Control System Overview, in: 5th Meeting of the U.S. Software System Safety Working Group, 2005. Available: http://sunnyday.mit.edu/safety-club/workshop5/Adaptive_Cruise_Control_Sys_Overview.pdf.

[41] S. Mubeen, J. Mäki-Turja, M. Sjödin, Support for Holistic Response-time Analysis in an Industrial Tool Suite: Implementation Issues, Experiences and a Case Study, in: 19th IEEE Conference on Engineering of Computer Based Systems, 2012, pp. 210 –221.