From Modeling to Deployment of Component-Based Vehicular Distributed Real-Time Systems

Alessio Bucaioni*, Saad Mubeen†*, John Lundbäck†, Kurt-Lennart Lundbäck†, Jukka Mäki-Turja†* and Mikael Sjödin*
* *Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Västerås, Sweden*
† *Arcticus Systems AB, Järfälla, Sweden*
*{*alessio.bucaioni, saad.mubeen, jukka.maki-turja, mikael.sjodin*}*@mdh.se*
†{*saad.mubeen, john.lundback, kurt.lundback*}*@arcticus-systems.com*

***Abstract*—We present complete model- and component-based approach for the development of vehicular distributed real-time systems. Within this context, we model and timing analyze these systems using one of the state-of-the-practice modeling and timing analysis techniques that is implemented in the existing industrial model the Rubus Component Model and accompanying tool suite. As a proof of concept, we conduct a case study by developing an intelligent parking assist system which is a distributed real-time application from the vehicular domain. The case study shows various stages during the development such as modeling of software architecture, performing timing analysis, simulation, testing, automatic synthesis of code from the software architecture, deployment, and execution.**

I. Introduction

Development strategies for vehicular real-time embedded systems are to an increasing extent based on a model- and component-based development approach. Such an approach uses models to describe functions, structures and other design artifacts. It supports the development of large software systems by integration of software components. It raises the level of abstraction for software development and aims to reuse software components and their architectures.

A. Paper Contribution

In this paper[1] we provide a complete model- and component-based development approach for vehicular distributed real-time systems. Using this approach, we model and timing analyze these systems using an existing modeling and timing analysis technique [2], [3] that is implemented in the industrial tool suite Rubus Integrated Component development Environment (Rubus-ICE) [4]. As a proof of concept, we demonstrate various steps from modeling to deployment during the development of the Intelligent Parking Assist (IPA) system. These steps include the following.

1) *Modeling:* Developing component-based software architecture of distributed real-time application with the Rubus modeling language.
2) *Analysis:* Analyzing the modeled application with Rubus-ICE using different types of analysis including the end-to-end response-time and delay analysis.
3) *Synthesis:* Automatically generating the code for the run-time infrastructure, i.e., the execution framework.
4) *Simulation and testing:* Executing the modeled application in a simulated environment from Simulink and testing at various hierarchical levels.
5) *Deployment:* Downloading the synthesized software of the application on a hardware platform.
6) *Execution:* Demonstrating the functionality of the application by configuring its inputs and outputs.

[1]The work in this paper is the complete version of the abstract [1].

In our previous works [2], [3], [5], we developed modeling and timing analysis techniques for distributed real-time systems. However, the validators for these techniques addressed only subsets of the above steps. This paper provides the first published validation for the modeling and analysis techniques in [2], [3], [5] by addressing all of the above development steps. As a case study we use an industrially inspired application for an intelligent parking assist function. While the function is somewhat simplified, we have structured it to mimic the design-patterns and requirements used by our industrial partners

B. Paper Organization

The rest of the paper is organized as follows. Sections II and III present the background and related works. Section IV describes the development approach. Section V presents the case study. Section VI concludes the paper.

II. Background – The Rubus Concept

Rubus [4] is a collection of methods, theories and tools for model- and component-based development of resource-constrained embedded real-time systems. Rubus is developed by Arcticus Systems in collaboration with Mälardalen University. It is today mainly used for development of control functionality in vehicles by several international companies. The Rubus concept is based around the Rubus Component Model (RCM) [6] and its development environment Rubus-ICE (Integrated Component development Environment) [7] which includes modeling tools, code generators, analysis tools and run-time infrastructure.

A. The Rubus Component Model

RCM expresses the infrastructure for software functions, i.e., the interaction between the software functions in terms of data and control flow separately. The control flow is defined by triggering objects, e.g., clocks and events. In RCM, a Software Circuit (SWC) is the lowest-level hierarchical element, i.e., the basic component, which encapsulates basic functions. SWCs have the run-to-completion semantics. They communicate each other via data ports. They may be encapsulated into software assembly (ASM) for constructing the system at different hierarchical levels. RCM facilitates analysis and reuse of components in different contexts by separating functional code from the infrastructure that implements the execution environment.

B. The Rubus Analysis Framework

The Rubus model allows expressing real-time requirements and properties on the architectural level. To this end, the designer has to express real-time properties of SWCs,

such as worst-case execution times and stack usage; this can be done with external static-analysis tools, or Rubus-ICE provides tools to measure these properties from executing code. An off-line scheduler is used for time-triggered tasks, which considers these real-time constraints when constructing a schedule. For event-triggered tasks, response-time analysis is performed and the calculated response times are compared with the specified deadlines. The supported analysis includes distributed end-to-end response-time and delay analyses [3] and shared stack analysis [8].

C. The Rubus Code Generator and Run-Time System

Based on the resulting component architecture, functions are mapped to tasks which are the run-time entities. Each external event trigger defines a task; accordingly, each triggered SWC, within a trigger chain, is allocated to the corresponding task. Within trigger-chains, inter-SWC communication is aggressively optimized to use the most efficient means of communication possible for each communication link. All clock triggered chains are allocated to an automatically generated static schedule, which fulfills the precedence order and temporal requirements. There are several criteria for the allocation of SWCs to tasks and construction of schedule, e.g., response times or memory usage. The run-time system executes all tasks on a shared stack, avoiding the need for static allocation of stack memory to each individual task.

D. The Rubus Simulation Model

The Rubus SIMulation Model (RSIM) and accompanying tools enable simulation and testing of applications modeled with RCM at various hierarchical levels, e.g., SWCs, ASMs, Electronic Control Units (ECUs), and complete distributed system. Within RSIM, the testing is carried out in an automatic generated framework based on the Rubus Operating System (OS) simulator. The input data is read from external tools or files, e.g., Matlab, and fed to the simulation and test processes, which in turn stimulates input ports and state variables using probes. The output data can be fed back to the external tools or files. The simulated environment is build on top of the application to be simulated. Hence, the application can be controlled from high-level tools such as LabView or Matlab/Simulink. These tools control the application by means of commands to run and stop the target clock for a specified number of ticks. This allows the execution flow to be visualized in each time increment.

E. The Rubus Execution Platform

While the RCM and Rubus-ICE are operating-system (OS) independent, code generators are OS-dependent. In this paper we assume the usage of the Rubus OS designed for predictable and dependable execution of Rubus tasks. It supports both time- and event-triggered execution of threads. It optimizes the run-time architecture by using the hybrid scheduling [9] which combines the static cyclic scheduling with the fixed priority preemptive scheduling. It can run on various hardware platforms. In order to support the simulation in other OSs, a version of Rubus OS is adapted to host environment that runs under Windows or Linux.

III. Related Works

We focus on the component technologies that are targeted towards the development of resource-constrained embedded real-time systems in the vehicular domain.

AUTOSAR [10] is an industrial initiative to provide standardized software architecture for the development of embedded software. When it was being developed, there was no focus placed on its ability to specify and handle timing-related information. Whereas this capability was taken into account right from the beginning during the development of the Rubus concept. AUTOSAR describes the software development at a higher level of abstraction compared to RCM. Unlike RCM, it does not separate control and data flows among components within a node. It does not differentiate between the modeling of intra- and inter-node communication which is opposite to the modeling of communication in RCM. 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 in RCM. In short, AUTOSAR is more focussed on the functional and structural abstractions, hiding the implementation details about execution and communication. Whereas, RCM supports the modeling, analysis and synthesis of the execution environment of software functions. AUTOSAR hides the details that RCM highlights.

TIMMO [11], a large EU research project, is an initiative to provide AUTOSAR with a timing model [12]. It describes a methodology and a language TADL [13] to express timing requirements and constraints. TADL is inspired by MARTE [14] which is the UML profile for model-driven development of real-time and embedded systems. TIMMO methodology uses EAST-ADL language [15] for structural modeling and AUTOSAR for the implementation. TIMMO-2-USE [16], a followup on the TIMMO project, includes a major redefinition of TADL and supports the AUTOSAR extensions regarding timing model. Arcticus Systems has been involved in TIMMO-2-USE project as one of the industrial partners. These projects 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. 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 type of timing analysis, e.g., end-to-end response-time analysis.

ProCom [17] is a two-layered component model for the development of distributed embedded systems. It is inspired by RCM and there are a number of similarities between the two, e.g., both have passive components, both separate control flow from the data flow, and both use the 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. It 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 the timing analysis at the level where it is done in RCM.

IV. Development Approach

In this section, we discuss our approach for the development of component-based vehicular distributed real-

time systems. The proposed approach consists of five major phases that range from modeling to deployment. These phases along with the tools[2] used at each phase are depicted in Figure 1. The development at each phase, except for the deployment, is carried out within the Rubus-ICE tool suite. By supporting most of the development process with a single tool suite, we avoid explicit interoperability management and reduce time and cost overheads.

In the modeling phase, the Rubus Designer is used to model software architecture in terms of components (describing software functions), their interactions, structures and other design artifacts as shown in Figures 7 and 8.

In the analysis phase, the architecture is annotated with real-time requirements, properties and constraints. The Rubus Analysis Framework supports various types of analysis by means of plug-ins for the Rubus-ICE, e.g., Holistic Response Time Analysis (HRTA) and End-to-End Delay Analysis (E2EDA) plug-ins. These plug-ins calculate (1) response-times of individual tasks and messages, (2) holistic response times of trigger chains, data chains, and mixed chains[3], and (3) end-to-end delays of data and mixed chains. The analysis results can be used for design space exploration, i.e., the application architecture can be further refined to meet the real-time requirements and constraints. In the

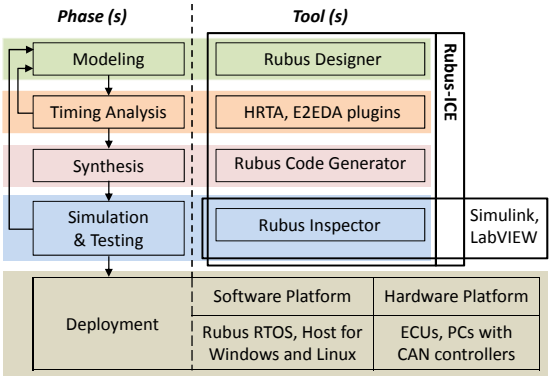

Figure 1. Development phases and corresponding tools used

synthesis phase, the Rubus code generator automatically generates the synthesizable code for the run-time framework, allowing the designer to work on a platform independent model and raising the productivity. A fragment of the generated code is depicted in Figure 2 which shows the data structures and functions for one of the SWCs (the PathCalculator SWC in the IPAssistant_ECU) in the case study that will be discussed in Section V.

In the simulation and testing phase, the Rubus Inspector (built on the RSIM) handles the component testing as well as the application simulation. The model-based testing [18] can be done at various hierarchical levels, e.g., the test object in Figure 3 can be an SWCs, an ASMs, an ECU, or a complete distributed system. Additionally, external high-level tools such as LabVIEW or Simulink can be used for feeding the simulation process with specific inputs.

The deployment phase involves both software and hardware platforms. The Rubus RTOS provides the software platform. It should be noted that the software architecture and corresponding synthesized code can be easily adapted and deployed to any RTOS. Similarly, any ECU or processor that runs the RTOS, e.g., IBM's PowerPC can serve as the hardware platform for deployment. Alternatively, a version of Rubus OS that is adapted to host Windows or Linux can be used as the software platform. In this case, a set of PCs (each equipped with a CAN controller) can serve as the hardware platform for the distributed real-time application (see SectionV-E for details).

```
/*============================================================
** Interface: PathCalculator_Interface
============================================================*/
typedef struct {
   int16_t const *rte_PathCalculatorInput;
   int16_t const *rte_AccelleratorPosition;
   int16_t const *rte_SteerTorque;
   int16_t const *rte_SteerAngle;
 } rte_IP_PathCalculator_InterfaceArgs_t;

typedef struct {
   int16_t    rte_Path;
   int16_t    rte_IPA_accellerator;
   int16_t    rte_IPA_brake;
   int16_t    rte_IPA_steer;
 } rte_OP_PathCalculator_InterfaceArgs_t;

typedef struct {
   int16_t    rte_PathCalculatorInput;
   int16_t    rte_AccelleratorPosition;
   int16_t    rte_SteerTorque;
   int16_t    rte_SteerAngle;
 } rte_IP_PathCalculator_Interface_Local_Args_t;

typedef struct {
   rteSwcInstanceAttr_t const                    *attr;
   rte_IP_PathCalculator_InterfaceArgs_t          IP;
   rte_OP_PathCalculator_InterfaceArgs_t         *OP;
   void                                          *ST;
   rte_IP_PathCalculator_Interface_Local_Args_t  *localIP;
 } rte_PathCalculator_InterfaceArgs_t;

#define RTE_PathCalculator_InterfaceArgs_t 1
/*============================================================
** Interface: PathCalculator_Interface behaviour(s)
============================================================*/
#define RTE_PathCalculator_Behaviour 1
extern void PathCalculator_Behaviour (rte_PathCalculator_InterfaceArgs_t const *args);
```

Figure 2. Fragment of the automatically generated code

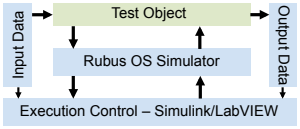

Figure 3. Testing at various hierarchical levels using the Rubus Inspector

V. Case Study

In order to provide a proof of concept for our development approach, we conduct a distributed real-time application case study from the vehicular domain. We develop the IPA system using our approach at various phases using the corresponding models and tools as shown in Figure 1.

A. Intelligent Parking Assist (IPA) System

The IPA system, also known as Advanced Parking Guidance System (APGS), is an automotive feature, which assists drivers in parking their vehicles. It uses a set of ultrasonic warning systems and a backup camera for detecting obstacles and calculating the optimum maneuvers during the parking operations. It has been divided into two subsystems namely IPAssistant and Actuator. The subsystems communicate with each other via CAN messages. Figure 4 depicts the block diagram of the IPA system.

1) IPAssistant subsystem: It reads inputs from the ultrasonic sensors, camera and manual button states. Further, it receives vehicle status information via CAN messages from the Actuator subsystem. If IPA is activated, it calculates the optimal maneuvers based on the sensed location information.

[2]In this paper, we do not look into the usage of the stack analyzer

[3]First task in a trigger chain is triggered independently, while the rest of the tasks are triggered by their predecessors. Each task in a data chain is triggered independently. A mixed chain is a combination of these chains.

Accordingly, it sends control information, as CAN messages, to the Actuator subsystem for adjusting speed, steer and brake of the vehicle during parking maneuvers. It also displays status messages on the user interface.

Figure 4. Block diagram of the Intelligent Parking Assist system

2) Actuator subsystem: It reads vehicle control information provided by the brake pedal and wheel sensors. It receives maneuvers control information via CAN messages from the IPAssistant subsystem. After processing the control information, it produces actuation signals for the brake, wheel, steer, gear and engine throttle controllers. It also sends the fresh control information via CAN to the IPAssistant subsystem for updating the maneuvers calculations.

B. Modeling of IPA System with RCM in Rubus-ICE

The component architecture of IPA System modeled with RCM is shown in Figure 5. Each subsystem is modeled as a separate ECU. The ECUs are connected to a single CAN network. We select the standard frame format for CAN messages. This means, each CAN frame uses 11-bit identifier. The CAN speed is 500 kbps.

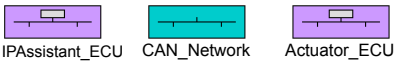

Figure 5. RCM model of the IPA system: system-level view

Figure 6 shows the eight CAN messages that are exchanged among the ECUs. A message is denoted by m. Each message is associated with one or more signals. The signal-to-message mapping information is provided in the signal database denoted by $SignalDB$ in Figure 6. Table I lists the extracted attributes of all messages. These attributes include data size (s_m), priority (P_m), transmission type (ξ_m), i.e., whether the message is periodic (P) or sporadic (S), period or minimum inter-arrival time (T_m), and transmission time (C_m) which is automatically calculated by the HRTA plug-in based on the value of (s_m).

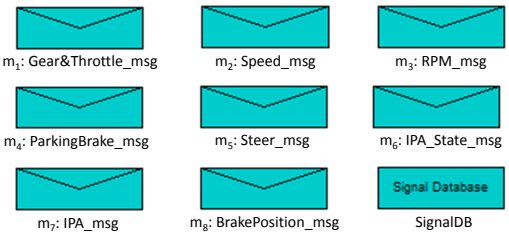

Figure 6. CAN messages and signal database modeled with RCM

1) Internal model of IPAssistant ECU: The internal architecture of the IPAssistant ECU modeled with RCM is shown in Figure 7. It consists of nine SWC, six ISWCs and two OSWCs. The SWCs denoted by UltrasonicInput1-4, Camera, and IPA_button read the inputs from the ultrasonic sensors, the camera and the IPA_button respectively. The InputProcessingIPA SWC processes these inputs together with the CAN messages m_1, m_2, m_3, m_4, m_5, and m_8 received from the corresponding ISWCs. Based on the processed inputs, the PathCalculator SWC calculates the optimal maneuvers, whereas the DiplayController SWC displays the vehicle status information on the display screen.

Table I
MESSAGE ATTRIBUTES EXTRACTED FROM THE MODEL

Msg	s_m	P_m	ξ_m	T_m (μs)	C_m (μs)
m_1	8	4	P	10000	270
m_2	8	5	P	10000	270
m_3	8	7	P	10000	270
m_4	1	8	S	10000	130
m_5	8	6	P	10000	270
m_6	1	1	S	5000	130
m_7	8	2	S	5000	270
m_8	1	3	S	10000	130

The IPA_State and IPA_message OSWCs send messages m_6 and m_7 over the CAN network respectively.

2) Internal model of Actuator ECU: Figure 8 shows the internal architecture of the Actuator ECU modeled with RCM. It is composed of seventeen SWCs, two ISWCs and six OSWCs. The InputProcessingActuator SWC collects and processes the inputs sensed from twelve SWCs along with the CAN messages m_6 and m_7 received from the IPA_State and IPA_message ISWCs. Accordingly, it calculates the actuation signals for the BrakeController, WheelController, SteerController and Gear&ThrottleController SWCs to perform the desired maneuvers. The fresh vehicle control information is sent over the network in six messages m_1, m_2, m_3, m_4, m_5, and m_8 by the respective OSWCs.

C. End-to-End Timing Requirements and Results

The IPA system is modeled with several trigger, data, and mixed chains. For convenience, we specify the end-to-end timing requirements on only five Data Chains (DCs) that are distributed over the whole system. These chains are identified as DC_1, DC_2, DC_3, DC_4 and DC_5. We specify 10 msec as the end-to-end deadline requirement on each of the first four DCs. The initiator of DC_1 is the task corresponding to the *UltrasonicInput1* SWC located in the *IPAssistant_ECU*. It is responsible for acquiring the ultrasonic sensor input and sending it to the *InputProcessingIPA* SWC. The last task of DC_1 corresponds to the *BrakeController* SWC which is located in the *Actuator_ECU* as shown in Figure 8. It is responsible for producing the actuation signal for the brake controller. All components in the data path of DC_1 (from initiator to terminator) are shown below.

- DT_1: $UltrasonicInput1 \rightarrow InputProcessingIPA \rightarrow IPA\_messageOSWC \rightarrow m_7 \rightarrow IPA\_messageISWC \rightarrow InputProcessingActuators \rightarrow BrakeController$

All components except from the initiator and terminator along the data path of DC_1 are the same in the data chains DC_2, DC_3 and DC_4. However, these chains are initiated by the *UltrasonicInput2*, *UltrasonicInput3* and *UltrasonicInput4* SWCs respectively and terminated by the *WheelController*, *Gear&ThrottleController* and *SteerController* SWCs respectively. We specify 20 msec and 30 msec as the data age and reaction constraints on DC_5 respectively. In RCM, these constraints are specified with start and end objects. The start objects are shown in Figure 7, while the end objects are visible in Figure 8. The initiator task of DC_5 corresponds to the *PathCalculator* SWC which is responsible for calculating the parking control information. Whereas, the terminator task of DC_5 corresponds to the *InputProcessingActuators* SWC in the *Actuator* ECU, as shown in Figure 8. It processes the

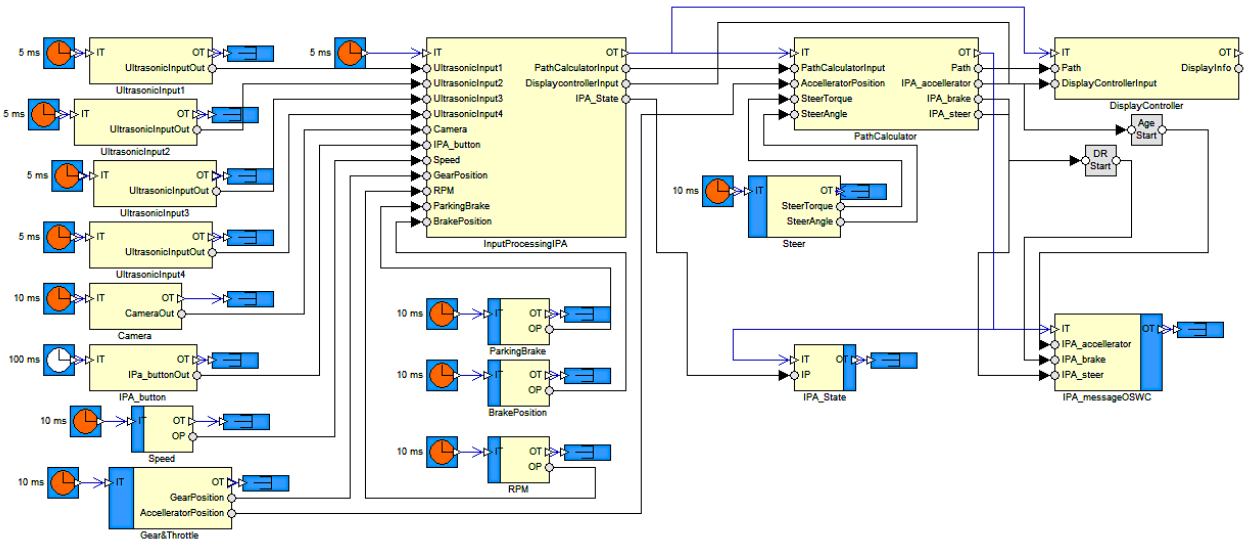

Figure 7. Internal component architecture of the IPAssistant ECU in RCM

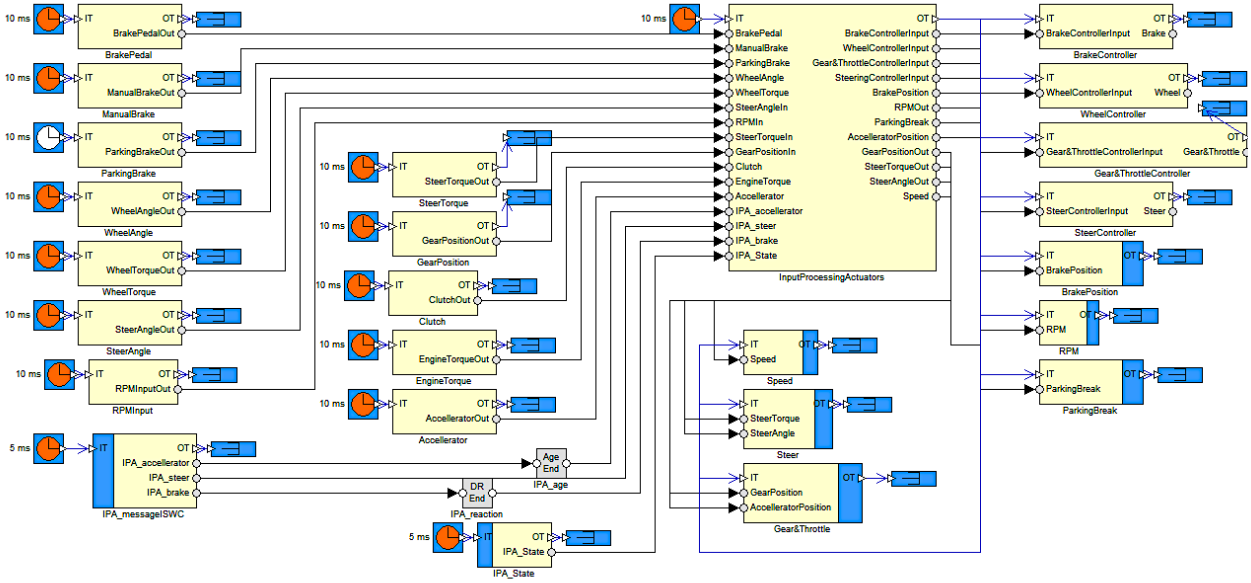

Figure 8. Internal component architecture of the Actuator ECU in RCM

vehicle control information. All the components in the data path of DC_5 (from initiator to terminator) are shown below.

- DT_5:$PathCalculator \rightarrow IPA\_messageOSWC \rightarrow m_7 \rightarrow IPA\_messageISWC \rightarrow InputProcessingActuators$

The end-to-end response times are calculated using the HRTA plug-in, whereas the end-to-end delays are calculated using the E2EDA plug-in [3]. The analysis results are shown in Table II. By comparing the end-to-end deadline requirements with the corresponding calculated end-to-end response times and delays, it is evident that all the DCs meet their deadlines and are deemed schedulable.

Table II
ANALYSIS RESULTS BY THE HRTA AND E2EDA PLUG-INS

DC	Requir. Type	Requir. Value (μs)	End-to-end Response Time(μs)	End-to-end Delay (μs)
DT_1	deadline	10000	1020	–
DT_2	deadline	10000	770	–
DT_3	deadline	10000	1250	–
DT_4	deadline	10000	1170	–
DT_5	Age	20000	–	15700
DT_5	Reaction	30000	–	25700

D. Simulation and Testing

We performed testing and simulation of the IPA system at various hierarchical levels using the Rubus Inspector. Figure 9 shows unit testing and simulation for the PathCalculator SWC in the IPAssistant_ECU. The input trigger is provided by the source clock with a period of 5 ms. The data is provided to the three data input ports by means of ramp, sine, and triangle wave generators. Whereas, the fourth data input port reads input from a text file. The outputs of this SWC are observed by display objects (showing the numerical outputs)

and plots. The real-time simulation of the plot is also shown in Figure 9.

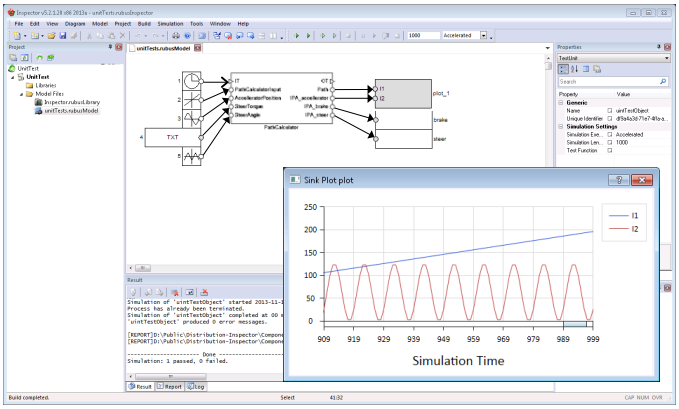

Figure 9. Unit testing and simulation using the Rubus Inspector

E. Deployment and Demonstration

As discussed in Section IV, there are two options for the deployment. In this work, we consider the second option. Hence, we deploy the synthesized code on the version of Rubus OS that is adapted to host Windows OS. For the hardware platform, we use two PCs to act as the two ECUs. Each PC uses the Kvaser USBcan II HS/HS CAN controller[4], which supports two channel CAN interfaces with a standard USB1.1 interface. We connect the CAN bus to the first channel of each controller. Whereas, the second channel is used to inspect and monitor the network communication and load, with the help of the CanKing tool by Kvaser that runs on the PCs. The deployed system is graphically illustrated in Figure 10.

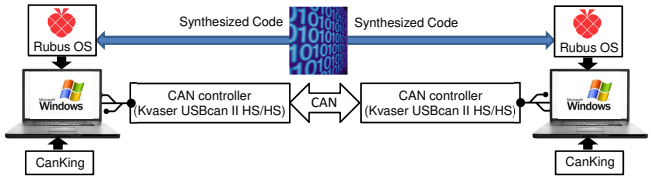

Figure 10. Deployment of the IPA system on software and hardware platforms

If the IPA system is deployed on the ECUs using the first option, the inputs are fed from the buttons, ultrasonic sensors, camera, etc., present in the experimental vehicle. In response to these inputs, the IPA system maneuvers the vehicle accordingly. Moreover, the corresponding messages are displayed on the user interface in the vehicle. Since, we deploy the IPA system using the second option, i.e., on the PCs connected to a CAN network, we show the execution by developing a simple stand-alone application that provides inputs and gets outputs to and from the IPA system. Alternatively, the application also supports input, for the IPA system, from a text file. The file may contain a trace recorded from the IPA system of an experimental vehicle.

VI. Conclusion

We presented step-by-step demonstration of a complete model- and component-based approach that is used for the development of distributed real-time systems in the vehicular-applications domain. We discussed various steps that are used in this approach including the modeling of software architecture, performing end-to-end response-time and delay analyses, automatically synthesizing the code from the software architecture, performing simulation and testing, deployment of the synthesized code, and finally demonstrating the functionality by executing the application. In order to provide the proof of concept, we demonstrated this approach by developing a distributed real-time application namely Intelligent Parking Assist (IPA) system with the existing industrial component model and accompanying tool suite. An interesting future work is to support interoperability of RCM and Rubus-ICE with other related tools used in the vehicular domain.

[4]http://www.kvaser.com/.

References

[1] A. Bucaioni, S. Mubeen, J. Lundbäck, K.-L. Lundbäck, J. Mäki-Turja, and M. Sjödin, "Demonstrator for modeling and development of component-based distributed real-time systems with rubus-ice," in *Open Demo Session of Real-Time Systems located at Real Time Systems Symposium (RTSS)*, Dec. 2013.

[2] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Communications-Oriented Development of Component- Based Vehicular Distributed Real-Time Embedded Systems," *Journal of Systems Architecture, http://dx.doi.org/10.1016/j.sysarc.2013.10.008*, Oct. 2013.

[3] S. Mubeen, J. Mäki-Turja, and 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*.

[4] "Rubus models, methods and tools," http://www.arcticus-systems.com.

[5] S. Mubeen, J. Mäki-Turja, M. Sjödin, and J. Carlson, "Analyzable modeling of legacy communication in component-based distributed embedded systems," in *37th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2011, pp. 229–238.

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

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

[8] M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Sjödin, "Bounding shared-stack usage in systems with offsets and precedences," in *20th Euromicro Conference on Real-Time Systems*, 2008.

[9] J. Mäki-Turja, K. Hänninen, and M. Nolin, "Efficient Development of Real-Time Systems Using Hybrid Scheduling," in *International Conference on Embedded Systems and Applications*, June 2005.

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

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

[12] "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," 2012.

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

[14] "The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems, 2010." OMG Group, January 2010. [Online]. Available: http://www.omgmarte.org/

[15] "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.

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

[17] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic, "A Component Model for Control-Intensive Distributed Embedded Systems," in *11th International Symposium on Component Based Software Engineering (CBSE), 2008*.

[18] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, 2007.