DEVS for AUTOSAR Platform Modelling

Joachim Denil[1,2] , Hans Vangheluwe[2,3], Pieter Ramaekers[1], Paul De Meulenaere[1], Serge Demeyer[2]

[1]TERA-labs
Karel de Grote University College
2660 Antwerp, Belgium

[2]AnSyMo
University of Antwerp
2020 Antwerp, Belgium

[3]School of Computer Science
McGill University
H3A 2A7 Montréal, Canada

{Joachim.Denil@, Paul.Demeulenaere, Pieter.Ramaekers}@kdg.be, {Hans.Vangheluwe, Serge.Demeyer}@ua.ac.be

Keywords: Discrete-event simulation, AUTOSAR, embedded software, performance analysis

Abstract

AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. Its design is a direct consequence of the increasingly important role software plays in vehicles. As design choices during the software deployment phase may have a large impact on the real-time properties of the system, designers need a method to explore various trade-offs. In this paper we evaluate the appropriateness of DEVS, the Discrete-Event system Specification, for modelling and subsequent performance evaluation of AUTOSAR-based systems. We demonstrate and validate our work by means of a power window and ABS case study.

1. INTRODUCTION

Software plays an increasingly important role in cars. About 30% of all innovations in current vehicles is related to software [1]. To keep complexity under control and to create a competitive market for automotive software components, some leading automotive companies created the AUTOSAR consortium [2]. The AUTOSAR technical goals include modularity, scalability, transferability and reusability of functional components. To achieve these goals, the AUTOSAR initiative has a dual focus. On the one hand it defines an open platform (middleware) for automotive embedded software through standardized interfaces. On the other hand it provides a method to create automotive embedded systems. Using AUTOSAR, software can be developed mostly independently from the platform it will be deployed on.

During the process of deployment onto hardware a plethora of configuration choices have to be made in the middleware. These choices range from the mapping of software functions onto tasks and assigning these tasks a priority, to parameters that affect the sending and receiving of messages on the bus. Because of the impact these choices have on real-time system performance, a method is needed to evaluate candidate deployment solutions.

A complicating factor is the reusability of functional components. There is no guarantee that a component will behave as intended in a new hardware/middleware configuration with respect to performance requirements such as timing. This non-compositionality means that during integration, these behavioural properties must be evaluated.

In this paper we evaluate DEVS, the Discrete-Event system Specification [3], as a suitable formalism for modelling and subsequent performance evaluation of AUTOSAR-based systems. We base our evaluation on simulation of the behaviour of a power window and an ABS application, two representative cases for applications to be deployed on an AUTOSAR platform.

The rest of the paper is organized as follows. Section 2. gives a brief survey of related work. Section 3. introduces relevant aspects of the AUTOSAR standard. Section 4. elaborates on the choice of the DEVS formalism. Section 5. explains the construction of the model of the AUTOSAR middleware as well as the power window application. Section 6. describes the experimental setup of the power window application and the analysis of the results obtained using the simulation model. Section 7. analyzes the reusability of the created models and how such models may be used to predict the timing behaviour in other configurations or cases, which is shown using another case study: the deployment of an ABS application and section 8. concludes.

2. RELATED WORK

Performance analysis is crucial for the deployment of safe and cost-effective software-intensive systems. For component based software engineering, the Palladio component model [4], provides a domain-specific modelling language to evaluate the performance of software architecture models. The models can be transformed into a queuing network based simulation model.

The DEVS formalism has also been used for developing embedded real-time applications, in [5], a model-driven method to develop these real-time embedded applications is introduced. For the evaluation of AUTOSAR-based systems both analysis techniques and simulations methods are available. On the analysis side, techniques are available to pre-

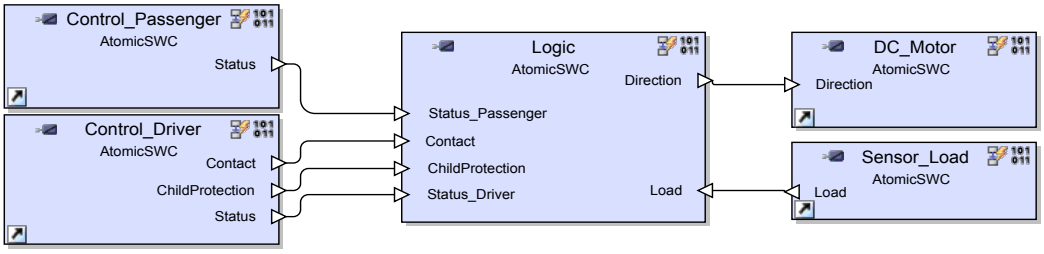

Figure 1. The software model of a power window application, showing the components and the interfaces for interaction

dict the timing behaviour after deployment. These techniques can yield worst and best case response times of the tasks and messages [6, 7]. On the simulation side, models can be built at various levels of abstraction, ranging from functional simulation with little timing information to true cycle simulations using binary code. In [8], SystemC was evaluated as a language for modelling and performance evaluation of AUTOSAR based software. Another approach is incorporating the effects of scheduling in Simulink models [9]. In both approaches, application components are simulated in combination with delays due to the communication hardware and the operating system scheduler. Our approach takes this a step further by simulating not only the application level, scheduler and communication bus level, but also the effects of the *configuration* of the full AUTOSAR platform. A more complete overview of tools that support the deployment of applications on platforms can be found in [10].

3. THE AUTOSAR FUNCTIONAL MODEL AND MIDDLEWARE

The functional model of AUTOSAR consists of a set of *atomic software components*. These components can interact with each other using *ports*. The service or data provided or required by a port are defined by its *interface*. This can be either a data-oriented communication mechanism (sender/receiver interface) or a service-oriented communication mechanism (client/server interface). The data-oriented interface can support 2 types of semantics. The first is "last-is-best", where only the last received value is stored. The other is a queued version where the data is stored in a queue until it is read.

Figure 1 shows the application model of the power window application. The application components communicate with each other using the sender/receiver interface. The unconnected ports are service-oriented client interfaces to communicate with the AUTOSAR middleware. Each software component defines its behaviour by means of a set of *runnables*. A runnable is a function that can be executed in response to *events*, for example from a timer or due to the reception or transmission of a data element. A runnable can also wait for the arrival of certain events for example when it needs another data-element to continue execution. These are called *waitpoints*. Finally, the runnable may need to update state variables, with exclusive read/write access. This is achieved using *exclusive areas*.

To make software components independent from the hardware, the interface to this hardware must be standardized. This is done using the AUTOSAR basic software, shown in Figure 2. This middleware consists of a real-time operating

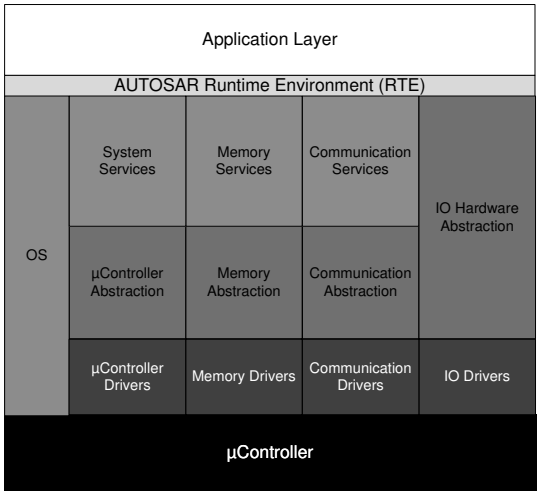

Figure 2. Structure of the AUTOSAR basic software, the run-time environment and the application layer

system based on the OSEK/VDX standard [11]. The operating system schedules tasks in a fixed priority way. Some tasks can be preemptive while others are not preemptive. Since the concept of a task is not known at the functional level, the components must first be mapped to the processors and then the runnables must be mapped onto tasks. The mapping to tasks is not necessarily 1-to-1. The rules for mapping runnables to tasks are defined in the RTE specification, available on [2]. All tasks have to be assigned a priority to be scheduled by the operating system.

The middleware also contains services for sending and re-

ceiving messages on a communication bus. These are composed of signals that originate in the application layer. Communication signals and messages have certain configurable properties, such as the signal transfer property and the message transmission mode, that have an impact on the timing behaviour of the application. More information about the configuration parameters in the communication stack can be found in the AUTOSAR communication specifications available on [2].

On the communication abstraction and driver layer, the most common automotive buses, for example the CAN-bus[12], are currently supported by the AUTOSAR communication stack. These also have many configuration parameters, such as the priority of the frames containing the message, that impact the real-time behaviour of the full system.

The run-time environment (RTE) is used as a glue between the functional components and the AUTOSAR basic software. It is responsible for storing the internal messages using buffers or forwarding the external messages to the communication stack. It also activates the runnables when an event occurs.

4. DEVS AND AUTOSAR

DEVS is a formalism for modelling discrete-event systems in a hierarchical and modular way, rooted in systems theory[3]. DEVS support two kinds of models. The first defines the behaviour of a primitive component and is called the atomic model. The other is the coupled model and is a composition of two or more atomic models that are explicitly connected. The coupled model itself can be part of an other coupled model. This allows for a hierarchical construction of DEVS models. A more thorough introduction to the formalism can be found in [3]. We use the Classic DEVS variant as it best matches the AUTOSAR semantics.

From an abstract point of view, the DEVS formalism provides excellent features for modelling AUTOSAR based systems. Here is a list of some properties of DEVS and their mapping to properties of automotive software and systems.

- Concurrency: Multiple processors and communication buses run concurrent in an automotive system. DEVS supports this by means of coupled models.
- Time: Real-time performance is a crucial property of automotive embedded software. End-to-end latencies are part of the requirements for these applications. The time advance function of an atomic DEVS model can be used to model latency.
- Events: Event-triggered and time-triggered architectures use triggers in the form of either external events or timing events to start certain pieces of functionality. DEVS implements reaction to events using the external transition functions.
- Priorities: Some automotive buses use a priority-based mechanism to arbitrate the bus (for example, the CAN-bus). DEVS supports this by means of a tie-braking function to select an event from the set of simultaneous events.

Some concrete examples of the mapping of AUTOSAR concepts to the DEVS formalism are given below.

- Runnables, the RTE and other basic software components are executable entities. They are pieces of code that change the state of the system during their time of execution. These can be mapped to the time-advance and the internal transition functions of an atomic DEVS model.
- The runnables are mapped onto tasks that run on a given processor. The tasks run on the AUTOSAR operating system. When an executable entity has finished its execution, the operating system could change state. This also maps to the internal transition function of an atomic DEVS.
- Messages and input/output hardware can interrupt the execution of the tasks. This could take time and could possibly trigger the execution of a runnable. DEVS can implement this using the external transition function.

5. THE AUTOSAR SIMULATION MODEL

In this section we construct a simulation model of an AUTOSAR-based power window. The simulation model is used to investigate the timing behaviour of the application.

The software model of this power window is based on [13]. The application controls the window on the passenger side, though both passenger and driver are allowed to open or close the window. When an object is present while closing the window, it will automatically detect this and lower the window. Figure 1 shows the AUTOSAR platform independent software architecture.

The two ‘Control’ components read out sensor signals from the buttons that control the window. The driver side component is also responsible for applying child protection on the power window and checking whether the ignition of the car is on. The ‘Load_Sensor’ component reads out the resistive force being placed on the window. When the execution of the runnables inside these components have finished, they make the sensor values available to the ‘Logic’ component. The ‘Logic’ component decides how to control the window using these sensor values and calculates the direction and speed of the window. The ‘DC_Motor’ component uses this to physically control the window. Both the ‘Control’ components, the ‘Logic’ component and the ‘Sensor_Load’ component are triggered by a periodic timing event (every 1 ms). The ‘DC_Motor’ component is triggered by the arrival of the ‘Direction’ signal from the ‘Logic’ component. In this case, all software components contain exactly one runnable.

The hardware contains two microcontrollers. One on the

driver side and the other on the passenger side. Both hardware units communicate through a CAN-bus with a bandwidth of 500 kbit/s.

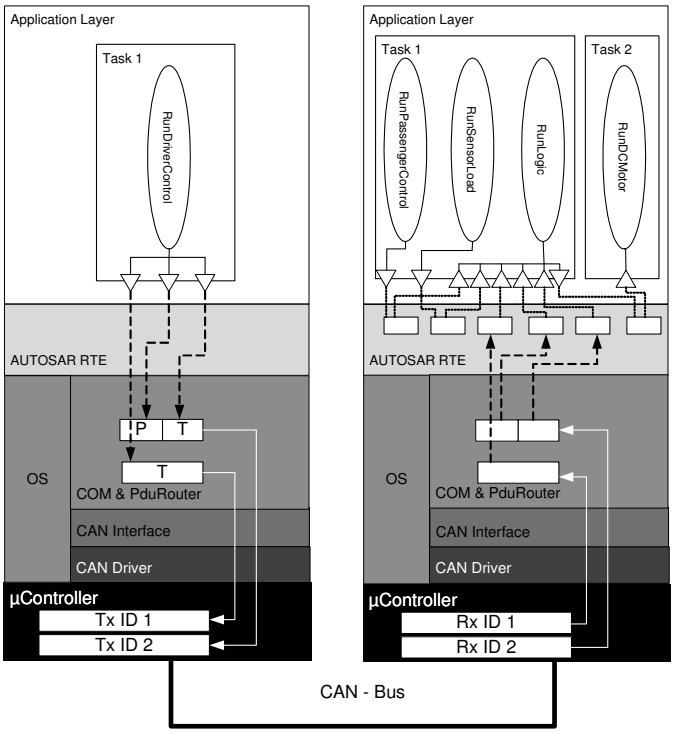

Figure 3. The power window application deployed on the hardware. P is a signal that does not cause the message to be transmitted in contrast to the T signals. Signal and message names are removed for reasons of clarity.

Figure 3 shows the full deployment of the application on the hardware. Major design decisions include the mapping of the components to the different control units. On the driver side, a single task executes the 'DriverControl' runnable. This task also transmits three signals on the bus, mapped to two different messages. The arrival of two signals in the communication stack cause the transmission of a message on the bus, while the other signal is only stored in the message without causing a transmission. On the passenger ECU two tasks are configured, one high priority task executing all the time-triggered runnables and a lower priority task executing the 'DCMotor' runnable.

5.1. The coupled DEVS model

Figure 4 shows the coupled DEVS model representing the deployment of the power window application. A short overview of the components is given below:

- DEVS Driver model: This atomic DEVS model represents the behaviour of the AUTOSAR operating system and the task running on the driver ECU. The time-triggered events from the timer module interrupt the execution of the model. During execution, the model sends two messages to the CAN transmit buffers.
- DEVS Passenger model: This atomic DEVS model represents the behaviour of the AUTOSAR operating system and the task running on the passenger ECU. The time-triggered events from the timer module and the reception of messages from the CAN receive buffer interrupt the execution of the model.
- DEVS timing event generators: These generates the timing events to activate the time triggered runnables. An event is generated every 1 ms.
- CAN transmit buffer: The output buffers of the CAN bus contain the frames that have to be send over the CAN bus. Due to priority mechanism, it is necessary to keep these in the buffer.
- CAN receive buffer: The input buffers receive messages from the CAN bus. It forwards all received messages to the passenger model.
- CAN bus: The CAN-bus is responsible for the physical transmission of frames. The delay that occurs when sending this frame depends on the size of the message and the speed of the bus. When 2 or more frames are available in the output buffers, the select function needs to select the message with the highest priority.

5.2. The model of a single control unit

The simulation model of the AUTOSAR basic software is abstracted to the component level. Execution times of these components are based on scenarios to match the real behaviour of the component.

Both driver and passenger models are similar, only the configuration of the runnables, tasks and BSW parameters differ. The state of the atomic DEVS model of a control unit (driver or passenger) can be divided into 3 major parts.

At the finest level of granularity there are the runnables. All the runnables in the power window application read the input values, do calculations on them and then transmit the result to the next process. While doing these calculations, the runnable is executed for a certain amount of time. This is reflected in the behaviour of the runnables in our simulation model. The time-advance for a single runnable is a configuration parameter of the simulation model.

Although this type of behaviour works for the 'Driver_Control' component, a finer granularity should be used to describe the way the runnable is executed. The runnable reads out three values from the hardware in a sequential order, sending out the value after every read in a different signal. This can be modelled by splitting up the execution in three parts using a state machine, where every state ends with a write operation of a signal.

The next abstraction level in the state of a control unit intro-

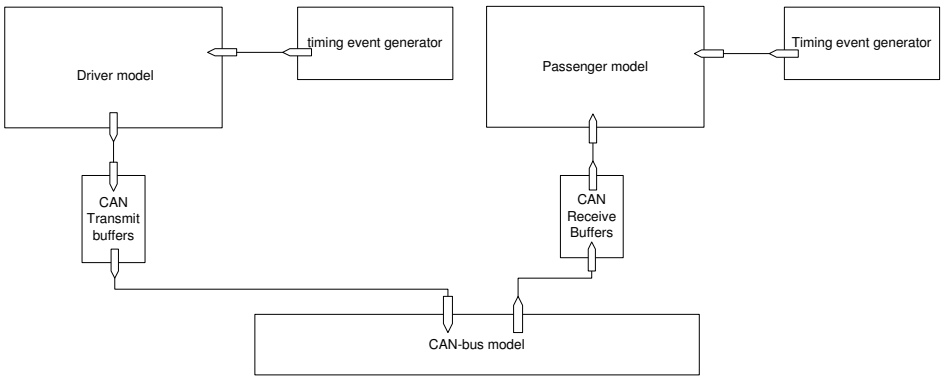

Figure 4. Overview of the coupled DEVS model, showing the connections between the different atomic models.

duces tasks. A task contains a set of runnables. When the task gets activated, it chooses the first runnable in the set where the state has been changed to activated. The task also keeps track of the time the runnable has executed, since the task could be preempted by an other task with a higher priority. Tasks are also responsible for sending the signals and messages through the communication stack. It therefore has access to the RTE and communication stack modules. These modules can be regarded, like the runnables, as entities that change the state of the model while taking a specific amount of time.

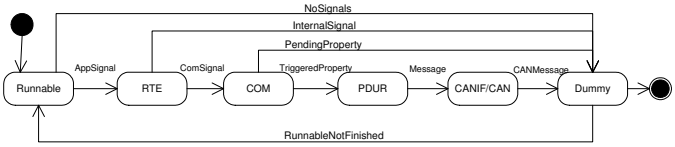

Figure 5. State of a task

Here is an overview of the responsibilities of each individual state in the task model, shown in Figure 5:

- RTE: The run-time environment serves as a glue for the communication between the different runnables on a single control unit or as glue to the basic software. Therefore it keeps track of the receive buffers of the interfaces. When an internal signal is written, the RTE places this value in the receive buffer. The RTE does not make any signals available for the task. The tasks reacts to this by changing to the 'DummyState'. A special case occurs when an activation event is coupled to the reception of a signal, as is the case for the 'DCMotor' component. In this case, the RTE generates an new activation event before going to the 'DummyState'. For external communication in the 'DriverControl' component, the RTE changes the application signals to communication signals and makes them available to the task. This way, the task may decide to go to the COM state.
- COM: The COM state mimics the behaviour of the COM module in the AUTOSAR basic software. When the COM receives a signal, it places this in the configured message buffer. The COM module checks the signal properties and message modes. Based on this, it decides whether to make the message available for the PDU-router.
- PDU-R: The PDU-router is used if more than one bus is attached to the control unit. It redirects the messages to the correct interface and driver for transmission on the bus. In our case study, there is only a single bus. The PDU-router makes the message available to the CANIF/CAN module after execution.
- CANIF/CAN: The AUTOSAR interface and driver of the CAN bus are used to place the message into the CAN transmit buffer. The module adds the message priority and length to the message. Occasionally, it buffers certain messages when the hardware buffers are full. These modules need to get executed in an atomic way, so the task cannot get preempted during the execution of these modules. The CANIF/CAN state has a similar behaviour, it adds message priority, length and the number of the buffer before making the CAN message available to the task.
- DummyState: The DummyState is introduced to notify the operating system that there could be CAN messages pending for transmission to the buffers, as is the case in the 'DriverComponent' or that there is an activation event pending when the 'runLogic' has finished execution. It does not take any time to execute. In the 'DriverComponent', it can happen that the runnable is not fully finished after the DummyState. In this case the task reactivates the runnable. Yet another situation can occur on the passenger side, where multiple runnables are mapped to a single task. Here, the task looks whether another runnable can be executed. If no other runnables are available, the task gets suspended by the operating system.

At the coarsed grained level there is the AUTOSAR operating system. The operating system keeps track of the tasks in

the model. If there are no tasks running, the operating system is in an *idle* state. The model can only get out of this state by means of an external event. External transitions can occur when the timing event is received by the control unit. The operating system checks, based on the received event, whether runnables and tasks need to be activated. Since this also takes a certain amount of time, the operating system changes to the *systemcall* state to compensate for this delay.

After this delay, the model goes to a *busy* state where the tasks are executed. The time-advance of the current state is looked up, by querying the current running runnable or executable entity in the running task. On the passenger side, another type of external transition can occur due to the reception of the CAN messages. In reality, the hardware will respond to an interrupt and take the message out of the hardware buffer and process the message using the communication stack. The simulation model reflects this behaviour by interrupting the current state for processing the message in the *interrupt* state. After the processing it returns to the previous state.

When an internal transition occurs in the *busy* state, the model notifies the running task that the current entity has finished its execution. When the task is in the 'DummyState' it reads out the activation events and the messages to be transmitted on the CAN-bus. It causes a transition of the operating system, to suspend the current task and/or activate other tasks based on the activation events, possibly even preempting or suspending the running task. This is reflected as before in the *systemcall* state.

While intuitively the executable entities, like the RTE, runnables and other communication stack modules, are responsible for keeping track of the elapsed time and their execution time, in the model they are not. Since the code for the RTE, COM, PDUR and CANIF is shared between the different tasks running on the operating system, the tasks are responsible for storing the time-advance and elapsed time of the entities executing within the task. This is to prevent a mismatch in timing behaviour when a task is preempted by another task.

5.3. The CAN-bus model

The CAN-bus model introduces the delays imposed by physically transmitting a frame on the bus. Figure 6 shows the state diagram of the simulation model. The model starts in an IDLE state with an infinite time advance. This represents the state when no messages are being transmitted on the bus. It changes state when one or more messages are put into the CAN transmit buffer. The tie-braking function checks the priority of the message and selects the one with the highest priority.

It then changes to the BUSY state. This state reflects the physical processes of transmitting the frame onto the communication medium. It stays in this state based on the length

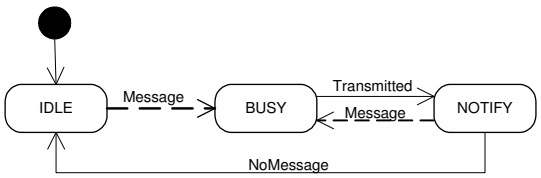

Figure 6. State diagram of the CAN-bus simulation model. Full lines represent internal transitions, dashed lines external transitions.

of the message and the configured bandwidth of the bus. On completion, the model writes the message to the CAN receive buffer on the passenger side. It also notifies the transmit buffers that the bus is ready for arbitration. In case there are pending messages, it returns to the BUSY state. Otherwise the bus returns to the IDLE state.

6. EXPERIMENTAL SETUP AND RESULTS

6.1. Calibrating the model

Before using the simulation model, it has to be calibrated (i.e, parameter values have to be estimated). Since we focus on the real-time behaviour, time delays for all actions that cause time to elapse need to be measured before the simulation model can be used. Here are the measurements that need to be completed:

- Execution time of all the runnables or states in the runnables, without the calls to the RTE;
- Execution time of activating or suspending tasks as well as the context switching times;
- Execution time of the transmission and receiving of messages in every part of the communication stack including the RTE.

We obtained these values by annotating the source code of our application. The measurements were done using an MPC560xP microcontroller with a 64 MHz clock speed. The microcontrollers are running an AUTOSAR basic software from Elektrobit GmbH. The source code was compiled using the MULTI compiler of Green Hills Software.

Table 1 shows the measured execution times of all the runnables, runnable states, communication modules and operating system primitives.

6.2. Results and analysis

The results of the simulation model execution of the power window application are shown in Figure 7. This shows how both processors execute the defined tasks in the correct order. The driver unit sends out two different messages on the CAN-bus during the execution of 'Task 1'. These signals are received by the passenger ECU after the delay introduced by the CAN-bus. In parallel, the passenger ECU executes the passenger input components and logic in its 'Task 1'. Based on the inputs, it controls the direction of the window in the

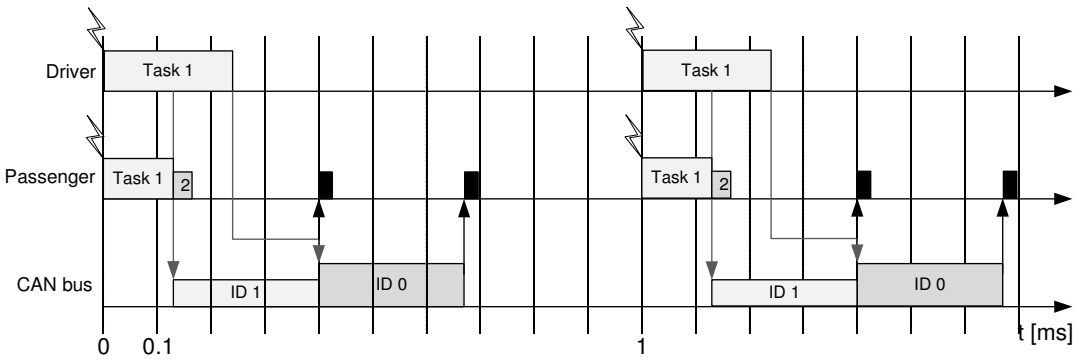

Figure 7. Graphical representation of the results of the power window simulation, the runnables are not shown for reasons of clarity

Runnables and states in runnables	
DriverReadContactState	2.44 μs
DriverReadChildProtectionState	2.36 μs
DriverReadStatusState	4.16 μs
PassengerRunStatus	5.01 μs
PassengerRunLoad	81.2 μs
PassengerRunLogic	39.7 μs
PassengerRunDCMotor	2.0 μs
Transmit communication stack	
RTEWrite (no event, internal)	0.7 μs
RTEWrite (event, internal)	1 μs
RTEWrite (no event, external)	1.6 μs
COMWrite (no trigger of message)	23.8 μs
COMWrite (Trigger of message)	46.6 μs
PDURouter	1 μs
CANIF/CANWrite	40.6 μs
Receive communication stack	
RxProcess CAN, CANIF, COM 1 message	32.2 μs
RxProcess COM, RTE 1 signal	2.8 μs
OS primitives	
ActivateTask	6 μs
SuspendTask	5 μs

Table 1. Execution times of the AUTOSAR primitives and runnables

second task. No buffers are overwritten before being read during the execution of the model.

The results obtained from the simulation model helps the AUTOSAR developer to analyse the impact on the real-time properties of his choices. It can be used to explore the various trade-offs while deploying automotive applications to AUTOSAR based ECUs.

The execution of the model matched the observed execution on our hardware platform. Since the source components are triggered by a timing event, the case is also very deterministic. This allows for an easy verification of the results obtained by executing the simulation model. It is known a priori when a runnable should start and what the order of execution will be. The results presented in this paper can be reproduced by downloading the complete simulation model from `www.iwt-kdg.be/teralabs/index.php?q=node/26`.

7. A GENERIC SIMULATION MODEL

In the previous sections we constructed the DEVS simulation model for the AUTOSAR basic software and the power window running in the application layer. The atomic components involved while building this model can be reused by changing the parameters of these components. This includes the number of tasks, runnables, signals, messages and all their AUTOSAR configuration options.

To show that the simulation models are reusable, we developed an ABS application, based on [14]. The software model of the ABS, shown in figure 8, contains 11 software components. The model is deployed on two ECUs, the UI (User Interface) component is mapped to the UI ECU, while all other components are mapped to the ABS ECU. The ABS ECU transmits two signals, mapped to two different CAN messages, to the UI ECU. The input components and the logic component are mapped to a single task that is triggered by a timing event. All other runnables are mapped to separate tasks that respond to the arrival of a data-element. The simulation model executed the ABS model as expected. The model can be downloaded from the same location as the power window application.

All the atomic models developed in this paper can be reused for the simulation of AUTOSAR based systems. The design choices made while deploying an application on an AUTOSAR based system, are input to these simulation models. Though, not all parameters available in the AUTOSAR standard are included in this version of the model, like the cyclic transmission of messages. The current model is the basis for a more generic model to simulate all the available configuration options when deploying AUTOSAR based sys-

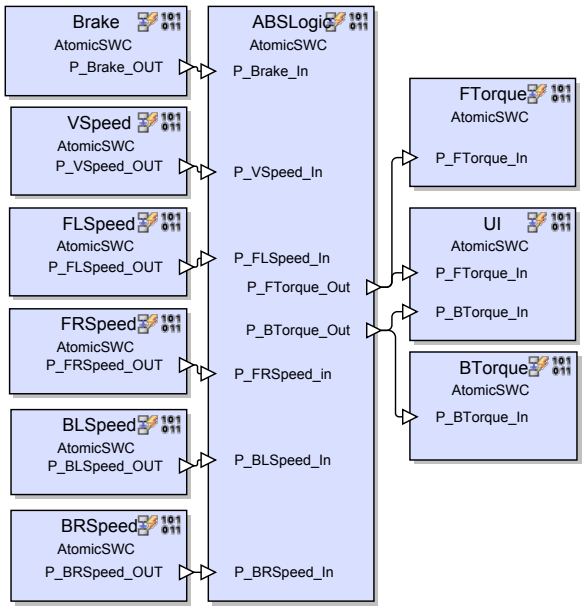

Figure 8. Software architecture of the ABS case study.

tems.

8. CONCLUSIONS

In this paper we have compared the properties of the DEVS formalism to the required properties for modelling AUTOSAR based systems. It is shown that DEVS is an excellent formalism to model the performance behaviour of AUTOSAR based systems. To support this reasoning, we constructed the simulation model of the AUTOSAR basic software and used it to simulate the real-time characteristics of a power window system. The execution of the model matched the observed execution on our hardware platform. As shown by the ABS model, the constructed AUTOSAR basic software model and CAN-bus model are a good starting point for building a generic model for the performance evaluation of AUTOSAR based systems, since the models can be reused and reconfigured for other case studies.

The results obtained by using this simulation model will help the AUTOSAR developer to analyse the impact of his choices on the real-time behaviour of the system. It can be used to explore the various trade-offs while deploying automotive applications to AUTOSAR based ECUs.

REFERENCES

[1] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings Of the IEEE*, 95, February 2007.

[2] AUTOSAR. Official webpage. `http://www.autosar.org`, 2010.

[3] B.P. Zeigler, H. Praehofer, and T.G. Kim. *Theory of modeling and simulation*, volume 276. Academic press New York, 2000.

[4] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[5] G. Wainer, E. Glinsky, and P. MacSween. A Model-Driven Technique for Development of Embedded Systems Based on the DEVS formalism. *Model-Driven Software Development*, pages 363–383, 2005.

[6] A. Hamann, M. Jersak, K. Richter, and R. Ernst. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Systems*, 33(1):101–137, 2006.

[7] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 187–192. ACM, 2002.

[8] M. Krause, O. Bringmann, A. Hergenhan, G. Tabanoglu, and W. Rosentiel. Timing simulation of interconnected AUTOSAR software-components. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.

[9] D. Henriksson, A. Cervin, and K.E. Årzén. TrueTime: Real-time control system simulation with MATLAB/Simulink. In *Proceedings of the Nordic MATLAB Conference, Copenhagen, Denmark*, 2003.

[10] Martin Torngren, Dan Henriksson, Karl-Erik Arzen, Anton Cervin, and Zdenek Hanzalek. Tool supporting the co-design of control systems and their real-time implementation: Current status and future directions. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 1173 –1180, 2006.

[11] OSEK consortium. OSEK operating system, v2.2.3. `http://www.osek-vdx.org`, 2005.

[12] CAN Bosch. Specification Version 2.0. *Robert Bosch GmbH*, 1991.

[13] S.M. Prabhu and P.J. Mosterman. Model-Based Design of a Power Window System: Modeling, Simulation and Validation. In *Proceedings of IMAC-XXII: A Conference on Structural Dynamics, Society for Experimental Mechanics, Inc., Dearborn, MI*. Citeseer, 2004.

[14] H. Ebrahimirad, M.J. Yazdanpanah, and R. Kazemi. Sliding mode four wheel slip-ratio control of anti-lock braking systems. In *Industrial Technology, 2004. IEEE ICIT '04. 2004 IEEE International Conference on*, volume 3, pages 1602 – 1606 Vol. 3, 2004.