Simulation

*Simulation: Transactions of the Society for Modeling and Simulation International*
1–25
© The Author(s) 2017
DOI: 10.1177/0037549716684552
journals.sagepub.com/home/sim
SAGE SCS

# DEVS for AUTOSAR-based system deployment modeling and simulation

**Joachim Denil[1,2,3], Paul De Meulenaere[1,3], Serge Demeyer[1,3] and Hans Vangheluwe[1,2,3]**

**Abstract**
AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, developed by automobile manufacturers, suppliers, and tool developers. Its design is a direct consequence of the increasingly important role played by software in vehicles. As design choices during the software deployment phase have a large impact on the behavior of the system, designers need to explore various trade-offs. Examples of such design choices are the mapping of software components to processors, the priorities of tasks and messages, and buffer allocation. In this paper, we evaluate the appropriateness of DEVS, the Discrete-Event System specification, for modeling and subsequent performance evaluation of AUTOSAR-based systems. Moreover, a DEVS simulation model is constructed for AUTOSAR-based electronic control units connected by a communication bus. To aid developers in evaluating a deployment solution, the simulation model is extended with co-simulation with a plant and environment model, evaluation at different levels of detail, and fault injection. Finally, we examine how the simulation model supports the relationship between the supplier and the original equipment manufacturer in the automotive industry. We demonstrate and validate our work by means of a power window case study.

**Keywords**
Simulation-based design, software-intensive systems, multiparadigm modeling, DEVS, AUTOSAR, deployment

## 1 Introduction

Software has become a key component of a rapidly growing range of applications, products, and services from all sectors of economic activity. This can be observed in large-scale heterogeneous systems, embedded systems for automotive applications, telecommunications, wireless ad hoc systems, business applications with an emphasis on web services, etc. For example, an automobile in the 1970s was an almost completely mechanical device, in which only the radio had some electronic components. By contrast, today's vehicles contain up to 70 electronic control units (ECUs) for controlling a range of features, from safety functions, such as anti-lock braking systems and electronic stability programs, to comfort functions for air-conditioning.[1]

Systems that feature a tight interaction and coordination between physical components and cyber-components are commonly referred to as cyber-physical systems. [2] We look at a specific subset: software-intensive systems. In such systems, software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole. One of the key enablers for the design of these complex systems is the use of system simulation tools. [3] Industrial leaders identified the increased prediction of system behavior (prior to testing) as a top strategy for system design. However, some observations are needed to create a usable system simulation model for software-intensive systems and, more specifically, AUTOSAR-based automotive systems.

For performance, cost, and practical reasons, the increasingly complex systems are often implemented in a distributed fashion. This means that the computational hardware components are scattered throughout the system and need to interact using a communication medium (most commonly, a bus). System architectures, like AUTOSAR (AUTomotive Open System ARchitecture), are commonly

[1]University of Antwerp, Belgium
[2]McGill University, Canada
[3]Flanders Make, Belgium

**Corresponding author:**
Joachim Denil, University of Antwerp, Middelheimlaan 1, 2020 Antwerpen, Belgium.
Email: Joachim.Denil@uantwerpen.be

used to keep the complexity of the system design under control by providing a method to develop the software as well as a standardized middleware for deploying the software. During the process of deployment onto hardware, a myriad of design choices must be made in the middleware and on the system. These choices range from the mapping of software components to ECUs, mapping 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 the functional and extrafunctional behavior of the system, a method is needed to evaluate candidate deployment solutions. However, the effects of deployment, combined with the interaction with the physical part of the system, make it difficult to analyze the overall system behavior. Simulation provides techniques for the concurrent simulation of the control model, the physical part of the system, commonly referred to as a plant model, and the environment models.

The problem of deployment is usually tackled at different levels of detail. [4] In the literature, performance analysis models have been proposed at these different levels of detail. One such technique is schedulability analysis. Schedulability analysis uses the worst-case timing behavior of the different components and checks whether an application meets the proposed deadlines. In a periodic activation model, every time a message is transmitted or received, a task (message) may need to wait for up to an entire period of sampling delay to read (forward) the latest data stored in the communication buffers. Analysis models add worst-case delays at each of these steps to obtain the worst-case latencies of paths. The probabilities of these worst-case delays is very small. [5] The analysis models are used for the purpose of safety analysis and not to evaluate the overall behavior of the system. Moreover, many applications are not time-critical. Performance and user comfort of the system largely depends on the average response time, which also needs to be analyzed and optimized. This is also true for many time-critical functions, where user comfort must be analyzed alongside safety. The effects of the design choices during deployment must also be verified with respect to the overall system behavior. More so, creating an analysis model taking all these design choices into account is technically hard. Simulation models are more appropriate in this case.

Automotive systems are usually also highly critical systems. A failure in these critical systems may lead to severe consequences, ranging from physical damage to the loss of human lives. Critical software-intensive systems must, therefore, be dependable. A number of techniques are created to make a system more dependable, for example *triple modular* redundancy. [6] To respond to this trend, the automotive industry proposed new safety and criticality standards. The ISO-26262 functional safety standard defines the safety aspects of the development of electric and electronic automotive systems. [7] Standard ISO-26262 provides a life-cycle model, processes, risk classes, and requirements for the validation measures to ensure that a sufficient level of dependability is achieved. Simulation is an important experimental method to obtain an early measure of the performance and dependability. [8] Faults can be injected into the simulation model in a controlled way.

Finally, automotive systems are developed in a supplier–OEM setting. Original equipment manufacturers (OEMs) often outsource the development of many components to supplier companies. Suppliers must verify the correctness with respect to the requirements of the component. This means that the component should be verified in a realistic setting. Simulation techniques, like rest-bus simulation,[9] can be used to check the behavior of the created components under realistic conditions by generating the normal operating conditions in which the subsystem has to function.

The development of a system-level simulation model for the deployment of AUTOSAR-based systems must take these observations into account. We therefore define the following requirements for such a system-level simulation framework.

- *Different levels of detail*. The framework should support multiple levels of detail where design choices can be verified.
- *System behavior*. The interactions of the physical part (plant) and the control part (hardware + software) of the system must be simulated together to verify the behavior of the software with respect to the system.
- *Supplier–OEM*. The framework should be usable both within the integration company as well as in supplier companies to verify the behavior of a subsystem.
- *Verification of the system under faulty conditions*. Fault injections are considered a standard technique to verify the behavior of the system when errors are present. At the system level, fault injection in simulation models can be used, prior to testing, to verify the correct behavior of the system in the presence of faults.

Note that this is a non-exhaustive list of requirements for a modeling and simulation framework for AUTOSAR-based systems, based on previously described observations. Other generic and company-specific requirements, for example, consistency management techniques between design and simulation models in distributed development teams, are considered beyond the scope of this paper. The contribution of this paper is to demonstrate the appropriateness of the Discrete-Event System (DEVS) specification formalism to model and simulate the effects of deployment. To show that DEVS is indeed an appropriate formalism, we compare the characteristics of DEVS with the requirements we observed previously. To further show this appropriateness, we create a generic simulation model for the automotive domain, including a model of the AUTOSAR platform, as well as a single communication medium. The model is implemented using PythonDEVS. [10] The proposed simulation model is based on our previous work. [11] We extend this work in different directions by adding different levels of detail, co-simulation, and fault-injection techniques.

The paper is structured as follows. Section 2 presents some essential background concepts relating to this work. Section 3 compares the characteristics of DEVS with the formulated requirements and elaborates on the use of the DEVS formalism. Section 4 introduces the model of the AUTOSAR platform and a communication bus. In Section 5, the AUTOSAR platform model is augmented with co-simulation, different levels of detail, and fault injection. In Section 6, we discuss the use of the proposed simulation model within an automotive design process. In Section 7, the created model is used to deploy and verify the behavior of an automotive power window. Section 8 reviews relevant contributions that deal with performance evaluation of software-intensive systems and automotive systems. Finally, in Section 9 some conclusions are made.

## 2 Background

In this section, we look at the background related to the proposed work. Firstly, the DEVS formalism is introduced. Secondly, we look at the AUTOSAR software architecture. Finally, we discuss the causal-block diagram formalism.

### 2.1 DEVS

The DEVS formalism was conceived by Zeigler. [12] It provides a basis for the compositional modeling and simulation of discrete-event systems where the time base is continuous. During a certain time, only a finite number of events occur that can change the state of the system. In between these events, the state of the system remains the same. We describe here the DEVS-with-ports formalism. We select the DEVS-with-ports formalism because the formalism is considered the assembly language of simulation. [13]

A system is modeled in DEVS using a composition of *atomic* and *coupled* DEVS components. An atomic model describes the behavior of a discrete-event system as a sequence of transitions between states. It also describes how the system reacts to external input events and how it generates output events. The atomic DEVS model is specified as $M = \, < X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta >$, where:

$X = \{(p, v) \mid p \in \text{iPorts}, v \in X_p\}$. This is the set of input events, where iPorts is the set of input ports and $X_p$ is the set of values for the input ports.
$Y = \{(p, v) \mid p \in \text{oPorts}, v \in Y_p\}$. This is the set of output events, where oPorts is the set of output ports and $Y_p$ is the set of values for the output ports.
$S$. The state set $S$ is the set of sequential states.
$\delta_{\text{ext}} : Q \times X \to S$. This is the external state transition function, with $Q = \{(s, e) \mid s \in S, e \in [0, ta(s)]\}$, where $e$ is the elapsed time since the last state transition.
$\delta_{\text{int}} : S \to S$. This is the internal state transition function.
$\lambda : S \to Y$. This is the output function.
$ta : S \to R_0^+ \cup \infty$. This is the time-advance function.

The time base of a DEVS model is not explicitly mentioned but is continuous. Informally, the operational semantics of an atomic model are as follows. At any given moment, a DEVS model is in a state $s \in S$. When no external event is given, the model remains in that state for the duration defined by the time-advance function $(ta(s))$. On expiration of this time, the model outputs the value $\lambda(s)$ through a port $y \in Y$. After the output of the event, the model changes its state to a new state given by the $\delta_{\text{int}}$ function.This is called an internal transition.When an external event is received, an external transition occurs.The new state is determined by the function $\delta_{\text{ext}}(s, e, x)$,where $s$ is the current state of the model, $e$ is the elapsed time since the last transition, and $x \in X$ is the external event that was received. The definition of the time-advance function states that this can take a real value between zero and infinity. When $.ta(s) = 0$, the model will trigger an instantaneous transition and the state is called a transient state. When $ta(s) = \infty$, the model will remain in this state until an external event occurs. This is called a passive state.

A coupled DEVS describes a system as coupled components of atomic DEVS models or coupled DEVS. The connections between the components denote how they influence each other: output events of one component can become, via a network connection, input events of another component. The coupled model is formally defined as $CM = \, < X, Y, D, M_d \mid d \in D, \text{EIC}, \text{EOC}, \text{IC}, \text{select} >$, where:

$X = \{(p, v) \mid p \in \text{iPorts}, v \in X_p\}$. This is the set of input events, where iPorts represents the set of output ports and $X_p$ represents the set of values for the output ports.
$Y = \{(p, v) \mid p \in \text{oPorts}, v \in Y_p\}$. This is the set of output events, where oPorts represents the set of output ports and $Y_p$ represents the set of values for the output ports.
$D$. This is the set of component names, for each $d \in D$.
$M_d$. This is a DEVS basic (i.e., atomic or coupled) model.
EIC. This is the set of external input couplings: $\text{EIC} \subseteq \{((\text{Self}, \text{in}_{\text{Self}}), (j, \text{in}_j)) \mid \text{in}_{\text{Self}} \in \text{iPorts}, j \in D, \text{in}_j \in \text{iPorts}_j\}$.

EOC. This is the set of external output couplings, $\mathrm{EOC} \subseteq \{((\mathrm{Self}, \mathrm{out}_{\mathrm{Self}}), (i, \mathrm{out}_i)) \mid \mathrm{out}_{\mathrm{Self}} \in \mathrm{oPorts}, i \in D, \mathrm{in}_i \in \mathrm{oPorts}_i\}$.
IC. This is the set of internal couplings, $\mathrm{IC} \subseteq \{((i, \mathrm{out}_i), (j, \mathrm{in}_j)) \mid i,j \in D, \mathrm{out}_i \in \mathrm{oPorts}_i, \mathrm{in}_j \in \mathrm{iPorts}_j\}$.
select. This is the tiebreaker function, where $\mathrm{select} \subseteq D \rightarrow D$, such that, for any nonempty subset $E$, $\mathrm{select}(E) \in E$.

The semantics for a coupled model is, informally, the parallel composition of all the submodels. Each submodel in a coupled model is assumed to be a process, concurrent to the rest. There is no explicit method of synchronization between processes. However, there is a serialization of events whenever there are two or more submodels that have a transition scheduled to be performed at the same instant. Logically, the transitions are made at that instant, but the implementation of these transitions on a sequential computer is serialized. The coupled model uses the tie-breaking function to define which model will transit first.

## 2.2 AUTOSAR

To keep complexity under control and to create a competitive market for automotive software components, some leading automotive companies created the AUTOSAR consortium. [14] This consortium contains OEMs and supplier companies. 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 of creating automotive embedded systems. Using AUTOSAR, software can be developed mostly independently from the platform it will be deployed on.

AUTOSAR describes a metamodel for the deployment of automotive software components to a set of ECUs. The metamodel of AUTOSAR spans three different areas: (a) software architecture, (b) the system, and (c) the ECU. A small introduction to the AUTOSAR concepts is given next. More information about AUTOSAR can be found on the AUTOSAR website.

*2.2.1 Software architecture model.* AUTOSAR describes software using a software-component oriented approach. For reasons of scalability and transferability of these components the model is centered around standardized interfaces. The functional model of AUTOSAR consists of a set of *atomic software components*. Compositions of different components can be used to structure the models hierarchically. Components can interact with each other using *ports*. The service or data provided or required by a port is 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 two 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.

Each software component defines its behavior by a set of *runnable entities*. A runnable entity 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 entity can also wait for the arrival of certain events. This can be used when it needs another data element to continue execution. These are called *waitpoints*. Finally, the runnable entity may need to update state variables, with exclusive read–write access. This is achieved using *exclusive areas*. Each software component defines the interfaces using a *software component template*. This contains all the information regarding the interfaces and the behavior of the software component.

To achieve the goal of transferability, software components are defined independently of the hardware. The *virtual functional bus* enables a virtual functional integration of the software defined at the software architecture level. To verify the functional behavior of the created software architecture, the virtual functional bus provides an abstract platform to simulate the component diagram.

*2.2.2 System configuration model.* The system model defines the available hardware that can be used in the system. This includes the number and types of the ECUs in the system. It also describes the communication hardware involved. Finally, a topology represents how the ECUs interact with the different communication buses. The system configuration model defines how the software is deployed on the hardware. This includes the mapping of atomic software components to the hardware units. Signals that are communicated between software components on different ECUs have to be transmitted on a communication bus.

*2.2.3 ECU model.* To make software components independent of the hardware, the interface to this hardware must be standardized. This is done using the AUTOSAR basic software, shown in Figure 1. The basic software consists of the following components.

- *Operating system*. The middleware consists of a real-time operating system based on the OSEK/VDX standard. [15] The operating system schedules tasks with a fixed priority. 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

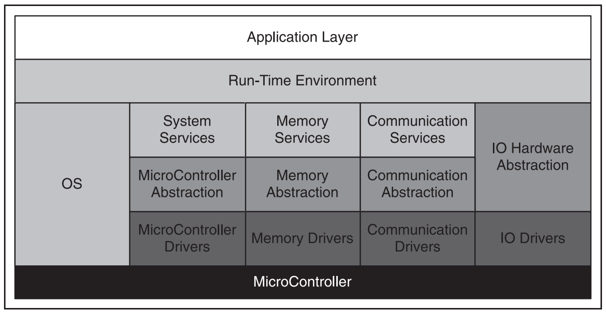

**Figure 1.** The AUTOSAR basic software.

processors and then the runnable entities must be mapped onto tasks. The mapping to tasks is not necessarily one-to-one. The rules for mapping runnables to tasks are defined in the run-time environment specification. [14] All tasks have to be assigned a priority to be scheduled by the operating system.

- *Communication services*. The middleware also contains services for sending and receiving 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, which have an impact on the timing behavior of the application. Table 1 shows the behavior in transmitting a message based on the signal transfer property and message transmission mode when a signal is written to the COM module. For a cyclic transmission, a period is required. Other design choices are parameters used to transmit the message for a set number of times or to prohibit a transmission for a certain amount of time after a previous transmission.
- *Communication abstraction and drivers*. On the communication abstraction and driver layer, the most common automotive buses, for example the CAN bus[16] and FlexRay-bus, [17] are currently supported by the AUTOSAR communication stack. These also have many configurable choices, such as the priority of the frames containing the message, which impact the real-time behavior of the full system.
- *IO abstraction*. Other services exist for reading and writing values from the hardware periphery units: analog-to-digital converter, pulse-width modulator, etc.
- *Run-time environment*. This 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 runnable entities when an event occurs.

Other services are available for diagnostics, memory management, error management, etc.

*2.2.4 Code generation.* Using the configuration templates, code can be generated on a per-ECU basis. For each ECU, an optimized middleware, containing only the required features for that specific ECU, can be generated. From the system template, the run-time environment code can be generated.

## 2.3 Causal-block diagrams

A common formalism to model the plant and environment of a software-intensive system involves causal-block diagrams. These are a general-purpose formalism used to model causal, continuous-time, and discrete-time systems. Causal-block diagrams are commonly used in tools such as Simulink®. They use two basic entities: blocks and links. Blocks represent (signal) transfer functions, such as arithmetic operators, integrators, and relational operators. Links are used to represent the time-varying signals shared between connected blocks.

Causal-block diagrams can be mapped to ordinary differential equations when continuous-time blocks are used. If discrete-time blocks are used, the causal-block diagram formalism can be mapped to difference equations. Another approach is to use a numerical method to solve the network, using a discrete time-step. Many numerical techniques are proposed in the literature to solve difference and differential equations.[18]

**Table 1.** Communication properties of the AUTOSAR COM module.

| Message mode | Direct | Cyclic | Mixed |
|---|---|---|---|
| signal property | | | |
| Triggered | Immediate transmission | Cyclic transmission | Cyclic and immediate transmission |
| Pending | No transmission | Cyclic transmission | Cyclic transmission |

The simulation of causal-block diagrams involves two steps:

- Establishing an evaluation order (including the detection of loops);
- Solving the network.

To establish the evaluation order, a directed graph is constructed. The topological sort algorithm, first presented by Kahn, is used to sort the blocks from source to sink. [19] The blocks are evaluated in order from source to sink. A problem arises when a loop is present in the directed graph, since the topological sort algorithm can only work on a directed acyclic graph. Because of this, the strongly connected components in the graph are first detected using Tarjan's algorithm.[19] These strongly connected components are solved as a single block. A multitude of solvers can be used to solve these components, ranging from Gauss–Jordan elimination for linear algebraic loops to iterative solvers for other types of loop.

## 3 Appropriateness of DEVS for deployment modeling and simulation

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

### *3.1 Concurrency*

Multiple processors and communication buses are concurrent in a software-intensive system. Even within a single ECU, there are components, like the communication buffers, that work concurrent from the processing unit. The semantics of DEVS coupled models supports concurrency by appropriate interleaving of the discrete-event behavior of individual submodels.

### *3.2 Time*

Real-time performance is a crucial property of real-time software-intensive systems. 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. This latency can be the computational time needed to execute a function or the time needed to transmit a message on the communication bus.

### *3.3 Events*

Event-triggered and time-triggered architectures use triggers in the form of either external events or timing events to execute certain pieces of functionality. DEVS implements a reaction to events using external transition functions. Time-triggering can also be handled by external events when the source of the trigger is a separate component to generate the timing event.

### *3.4 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 tiebreaking function to select an event from the set of simultaneous events.

### *3.5 Modularity*

Atomic DEVS modules are modular. They can be replaced by more abstract components. This allows us to create a simulation model that can be used at different levels of detail. It also allows us to replace an ideal component with a component that behaves as if faults are present, allowing us to inject faults into the simulation model. In the case of a fault injection, this type of component is known as a mutant. [8]

### *3.6 Compositionality*

The compositionality feature of DEVS helps us to address two different requirements for the modeling and simulation of software-intensive systems. Firstly, suppliers build part of the system concurrently with other parts of the system. The compositionality of DEVS allows the addition of extra abstract components and generators to create more realistic scenarios for simulation at the supplier. It allows OEMs to integrate the different components into a single simulation. Secondly, the compositionality feature can also be used to inject faults in the simulation. A saboteur [8] can be placed in between the output and input ports of two connected components. The component receives the events and can alter the timing behavior as well as any values piggybacked within the event.

### *3.7 DEVS as a common denominator*

DEVS is a very general formalism and is able to simulate different additional formalisms. [13] This generality stems from the infinite possible states that DEVS allows us to model and the (continuous) time elapse between the different state transitions. The hierarchical coupling techniques are used to integrate the different formalisms using DEVS as a common denominator.

## 4 The AUTOSAR simulation model

In this section, we construct a model for the timing simulation of the deployed AUTOSAR components onto a set of networked ECUs. This simulation model simulates the timing behavior of the deployed system when all deployment choices have been made, namely the operating system and communication stack are completely configured. However, the model does not evaluate the behavior of the full system because there is no connection with the environment and plant models. In the next sections, this base model will be extended with different additional features to support the modeling and simulation of software-intensive systems.

## *4.1 Coupled model*

The atomic models in the coupled model represent concurrent modules in a software-intensive system architecture. The most obvious atomic model is the *ECU*. It is the atomic DEVS model that will execute the application software. Because most software-intensive systems are distributed over several ECUs, the ECU models need to communicate. This is done using a model of the communication bus. The buffers that receive and transmit messages on the bus also work concurrently with each other. They are represented by a separate atomic DEVS model. Finally, a schedule table model is used to start time-triggered tasks on an ECU. Figure 2 shows a partial class diagram of the simulation model.

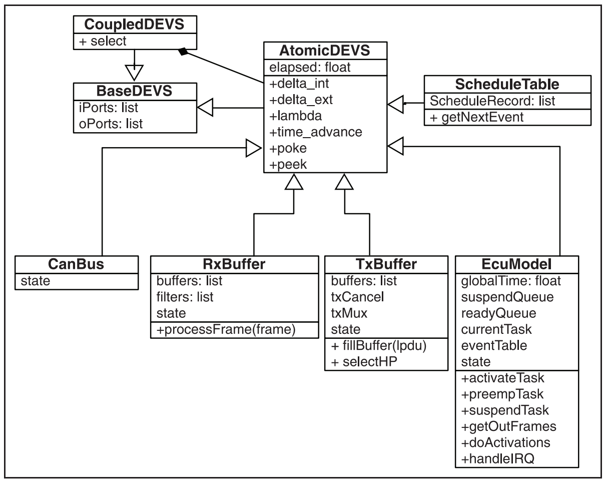

**Figure 2.** Class diagram of AUTOSAR deployment simulator.

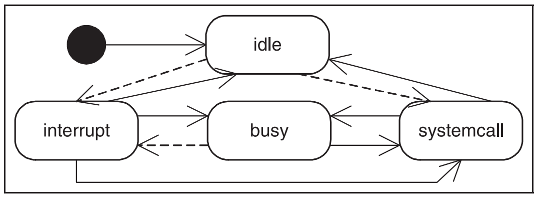

**Figure 3.** State diagram of the operating system. Full lines represent internal transitions; dashed lines, external transitions.

## *4.2 ECU*

The ECU atomic DEVS model represents the actual processor with operating system to which tasks are mapped. However, it does not contain any peripherals, since these usually work concurrently with the processing unit itself.

Figure 3 shows a state diagram for the operating system that controls the behavior of the ECU model. The ECU model starts in a *idle* state. This state defines that no tasks are executed at the processor. The ECU model will remain in this state until an external event is received. This external event is either a timing event received by the schedule table or a communication event. On reception of a timing event, the ECU model goes to a *systemcall* state. The event activates the necessary task(s) at the ECU model. The time advance for the systemcall state is defined by a parameter in the model. After completion, the ECU model starts running the task (*busy* state). On reception of a communication event from the receive or transmit buffer, the state of the ECU model is changed to the *interrupt* state. The reception of a message or a transmission confirmation can result in the activation of a task (or a set of tasks), resulting in a transition to the *systemcall* state. When there are no tasks to run after the time advance of the *interrupt* or *systemcall* state, the ECU model returns to the *idle* state.

Because the operating system keeps track of all the tasks running in the ECU model, it is similar in design as a normal operating system. It keeps track of all tasks using two different lists. The *suspended* list contains all tasks that are, at that point in time, suspended or waiting for an event. The systemcall state activates the tasks by taking them out of the suspended state and placing them in the *ready* queue. This is a priority queue, ordered by the priority of the task. Since only a single task can be executed by a single ECU, the task with the highest priority is always chosen as the running task.

When the task is activated from the suspended state, the task starts by executing the first runnable entity mapped to this task. A runnable entity, like other executable modules, has an execution time parameter that must be configured. After execution of this runnable entity, the task can go through a number of states, depending on the configuration, as shown in Figure 4. The internal transition mechanism is used to change between the different states. We distinguish the following states.

- *Run-time environment*. In the run-time environment state, the task keeps track of the data buffers of the interfaces. When an intra-ECU signal is written, the run-time environment state places this value in the corresponding receive buffer. To signal to the underlying operating system model that the work is done, the task state is changed to a ‘‘DummyState’’

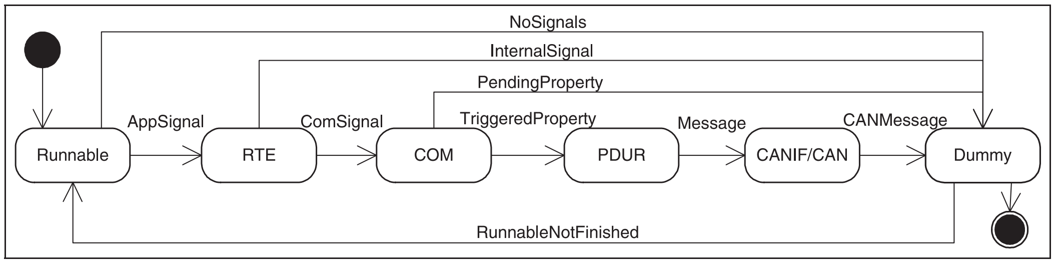

**Figure 4.** State diagram of a task.

by the internal transition function. In some cases, the run-time environment state produces activation events that are used by the operating system model to activate certain runnables or tasks. For external communication, the run-time environment state changes the application signals to communication signals and transitions to the COM state.

- *COM*. The COM state mimics the behavior of the COM module in the AUTOSAR basic software. When the COM is activated, it places the messages received from the run-time environment state in the configured message buffers. The COM state checks the signal properties and message modes. Based on the defined properties, it decides whether to make the message available to the PDU-router state or to transit to the DummyState.
- PDU-R. The corresponding AUTOSAR module is used to route the messages to the correct interface. This means that different CANIF/CAN instances can be used within a single ECU. The PDU-R contains a translation table to decide the routing to the correct interface or driver instance.
- CANIF/CAN. This state represents the AUTOSAR interface and the driver of the CAN bus. The CAN and CANIF modules are used to place the messages into the CAN transmit buffers. The module adds the message priority and length to the message. Occasionally, it buffers certain messages when the hardware buffers are full. These modules must be executed in an atomic way, so the task cannot be preempted during the execution of these modules. The CANIF/CAN state has a similar behavior; it adds message priority, length, and the number of the buffer before making the CAN message available to the operating system model.
- DummyState. The DummyState is introduced to notify the operating system that there could be CAN messages pending for transmission to the buffers or that there is an activation event pending. It does not take any time to execute. However, it can happen that the runnable is not fully finished after the DummyState. In this case, the task reactivates the runnable. Another situation can occur where multiple runnables are mapped to a single task. In this case, the operating system checks whether another runnable entity can be executed. If no other runnables are available, the task is suspended by the operating system.

Because of the priority mechanisms in the operating system, a running task can be preempted by a higher priority task that has become available due to the interrupt and systemcall mechanisms described. It is the responsibility of the task to keep track of which state it has to resume and how much time has already passed before the preemption occurred.

Besides the normal task behavior, a second type of task can be created: the schedule manager task (SchM_Task). The SchM_Tasks are special tasks that execute the main functions of the COM stack. They are used for the cyclic transmission of messages, for the reception of messages using the polling mode, and for other special behavior in the COM module. The SchM_Task thus contains. as runnable entities, the main functions of the different modules of the COM stack. The DummyState is also used to signal to the operating system that the task is completed.

The described behavior of the ECU atomic DEVS model is captured in the class diagram shown in Figure 5. For communication with the other atomic models in the coupled DEVS model, the EcuModel has two input ports and two output ports. Depending on the configuration not all of them need to be connected to another model.

- *FromRxBuffer*. This port receives an event, with a piggybacked CAN frame, from the connected RxBuffer. Depending on the configuration of the AUTOSAR ECU model, the external transition functions will unpack a frame in either an interrupt-based way or a polling-based way:
- *Polling*. Store the frame in the CAN/CANIF module. This will set the state of the ECU to an *interrupt* state. The internal transition time is a configurable parameter of the model.
- *Interrupt*. Go through the several layers of the communication stack and unpack the different signals to application signals in the run-time environment.

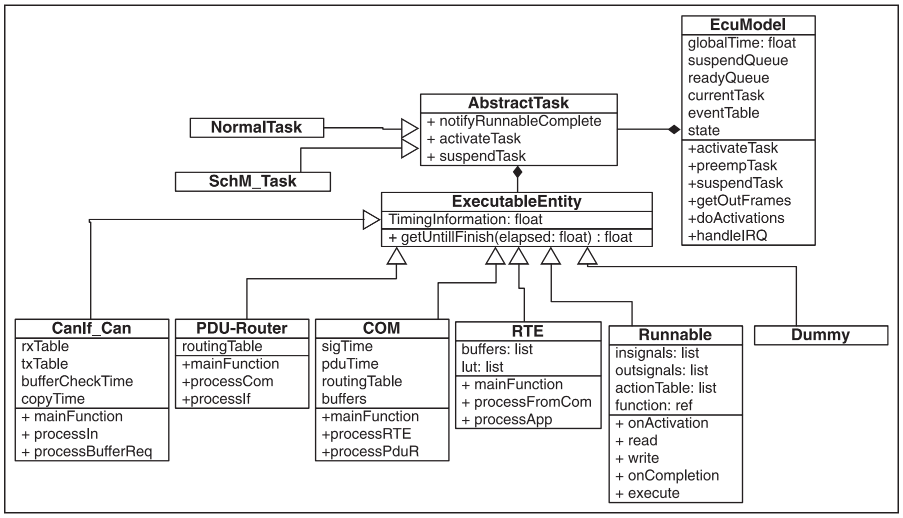

**Figure 5.** Class diagram of the ECU atomic DEVS model.

The state of the ECU is also changed to *interrupt* state, but the time advance is a sum of the different configurable timing parameters of the communication stack modules. Note that several task activation events can be generated from the unpacking of the frame into signals. These are stored and evaluated after the *interrupt* time-delay is finished. This results in the transition of the *interrupt* state to the *systemcall* state.

- *FromTxBuffer*. The FromTxBuffer port receives an event to notify the processor that a frame has been transmitted on the CAN bus. This is to allow notification of transmission events to the application layer. These events are, like the frames, processed within an interrupt or by using a polling mechanism. The behavior is as described before. The FromTxPort is also used for returning frames out of the TxBuffer when the configuration has the *cancellation* option defined or when software buffering is enabled and the transmit buffer rejects the frame. This will store the frame in the CAN/CANIF software buffer for resending when the TxBuffer is available.
- *ToTxBuffer*. This is used for sending an event, with a piggybacked CAN frame, to the TxBuffer.
- *FromScheduleTable*. This port receives timing events for the activation of time-triggered tasks. A look-up table is used to activate tasks this way. The EcuModel will activate a set of tasks by changing to the *systemcall* state.

## 4.3 Buffers

The TxBuffer and RxBuffer work independently of the processor. In most common processor systems, these buffers are within the communication controller of the ECU. The CAN controller, implementing the CAN bus protocol defines how a message is placed on the bus. Both the TxBuffer and RxBuffer have a specifiable number of internal buffers. Each ECU model that needs to transmit or receive a message on the CAN bus needs its own set of buffer models. The TxBuffer has two input and two output ports:

- *FromECuU*. The TxBuffer receives a piggybacked frame from the ECU model via this port. It also contains the buffer number where the frame should be placed.
- *ToCAN*. Depending on the configuration of the CAN module, either the first or the highest-priority message is put on the port to the CAN bus. The event contains a complete frame. Because several buffers in the whole system can transmit a message at the same time, the tiebreaking function is used to select the highest-priority message among the competing messages.
- *FromCAN*. The CAN bus transmits a notification message to the buffer when the transmit of the frame is complete. All transmit buffers receive this event, though only the source buffer of the transmitted message forwards this to the ECU.
- *ToECU*. This port is used to notify the processor that the frame has been transmitted on the bus. It is also possible to change the content of a buffer when *cancellation* is supported or when the buffer is full. The TxBuffer returns either the content of the buffer to the ECU model for storage in the software buffers or returns the incoming frame from the ECU model.

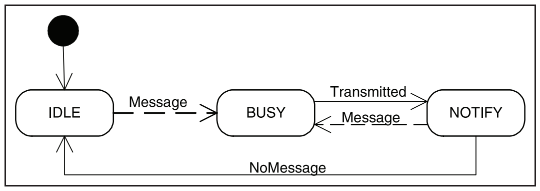

**Figure 6.** State diagram of CAN bus simulation model. Full lines represent internal transitions; dashed lines, external transitions.

### 4.4 Buses

The AUTOSAR standard supports different communication mediums. In this paper, we only model the behavior of the CAN bus. 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 tiebreaking function checks the priority of the message and selects the one with the highest priority.

The model then changes to the busy state. This state reflects the physical processes of transmitting the frame to the communication medium. It stays in this state based on the length of the message and the configured bandwidth of the bus: $t = (1/\text{speed}) * \text{size}$.

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. If there are pending messages, it returns to the busy state. Otherwise, the bus returns to the idle state.

The CAN atomic DEVS model has three ports to communicate with the buffer models.

- *InFrame*. The InFrame port accepts incoming frames. It connects to all the TxBuffers. Frames are only accepted when the bus is in an idle and notify state.
- *Notify*. The Notify port lets the TxBuffer models know that the bus is no longer in a busy state (transmitting a frame on the bus).
- *OutFrame*. A frame is put on the OutFrame port when the transmission delay is completed. It connects to all the RxBuffer models.

### 4.5 Schedule table

The schedule table model implements the schedule table mechanism of the AUTOSAR operating system. It generates timing events to activate periodic tasks. The only output port transmits this event to the attached ECU model. The ECU model activates the necessary tasks based on this timing event.

## 5 Augmenting the base model

In this section, the base model is augmented with (a) co-simulation, (b) simulation at different levels of detail, and (c) fault injection.

### 5.1 Co-simulation

In the previous section, all aspects of the computational part of a software-intensive system have been modeled. With this model, it is already possible to evaluate the timing behavior of applications, because the DEVS formalism interleaves the executions of the different runnables, basic software modules, and buses with the incurred delays. However, we cannot check the behavior of the overall system with this model. A co-simulation of the DEVS formalism with the plant and environment model is needed to evaluate the full behavior of the system, together with the execution of the software functionality within the defined runnables.

For this, three behaviors must be added to the previously described model.

- *Execution of the software functions*. In the previous model, no application code is executed during execution. To observe the system behavior, the application code must be executed during the simulation at the correct time. The input values are accessed by the runnable entity in the run-time environment. The output values of the runnable entity are written in the run-time environment.
- *Passing data in the events*. The functions need the input data of other runnables. As in the implemented AUTOSAR system, the values are passed through the communication stack, while building up a frame that is transmitted on the bus. The communicated data are piggybacked in the event that is exchanged between the different coupled DEVS models.
- *Integration of the plant and environment models in the simulation model*. The plant and environment models must be co-simulated with DEVS. Two extra ports are added to the ECU model to receive and transmit values to the plant and environment models.

The signals from the environment and plant model are stored in the run-time environment of the ECU model. This is a simplification that we made during the design. The different runnable entities sample (polling-based) the values, as in the AUTOSAR implementation. In this case, the time needed to sample this value is reflected in the time advance of the runnable entity. When an interrupt-based approach is needed, a dedicated atomic DEVS model is used to detect the interrupt and store the value in the run-time environment. The interrupt mechanism of the ECU model is extended with the time needed to store the value in the ECU. The run-time environment starts different tasks or sets a number of events in the real-time operating system, reflecting the actual behavior.

As shown previously,[13] DEVS is an appropriate formalism to combine different formalisms. We will give a single example of integrating another formalism in DEVS. To co-simulate causal-block diagrams with our simulation model, we embed the full causal-block diagram model and solver in an atomic DEVS block. The time-step is generated by the time-advance function of the DEVS atomic model. We use the same time-step for all the blocks in the model; it is, therefore, necessary to change the select function of the coupled model so that it reflects the same source to sink execution as in the causal-block diagram simulation model. The topological sort algorithm is used for this purpose.

With this extension to our base model, it is possible to verify the behavior of the whole system when all deployment choices have been made. Using a set of scenarios, usually already defined as use-cases in previous development steps, the choices made during deployment can be evaluated.

## 5.2 Levels of detail

Depending on the abstraction level, the AUTOSAR design and deployment model does not yet contain enough information to create the full simulation model described in Section 4. We identified three levels of detail that are useful to check the behavior of the system when not all the information is present in the design model.

*5.2.1 Virtual functional bus.* The first level of detail, depicted in Figure 7(a), uses a completely abstract platform where no inter-ECU communication is used. This is known as the virtual functional bus. The model is used to verify the behavior of the AUTOSAR software component diagram. The software component diagram contains such information as software components, runnable entities, and triggering events.

The virtual functional bus model uses a single ECU atomic DEVS model to simulate the behavior of the software component diagram. Each runnable entity defined in the AUTOSAR software component diagram is mapped to its own task for simulation. Because every runnable entity is mapped to the same ECU, there is no need for the AUTOSAR communication stack. Thus, the communication stack is not included in the simulation model and does not need a configuration. The run-time environment can handle all the communication between the different

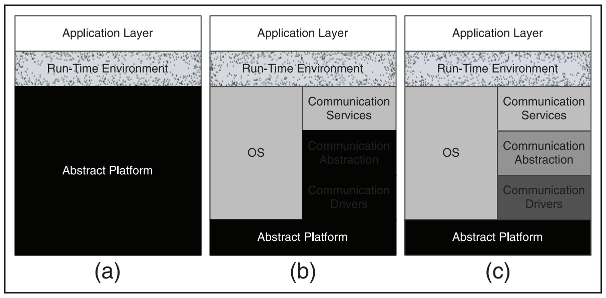

**Figure 7.** Levels of detail defined with the simulation model: (a) virtual functional bus, (b) task and message scheduling, (c) full deployment.

software components. The triggering events to execute the runnable entities on the ECU model are defined in the design model. The coupled model contains a ScheduleTable model when timing events are used to trigger the execution of one or more runnable entities. The time advances for executing a single runnable entity or a run-time environment call are set to zero. The model is untimed because there is no available information on the concurrency of runnable entities in the component diagram. However, to create a correct simulation model, the execution chain of runnable entities must be correct. The priority of the tasks in the simulation model are, therefore, based on the topological ordering of the flattened (i.e., only runnable entities that communicate without the hierarchy of the software components) AUTOSAR software component model. Again the topological sort algorithm is used for this.

*5.2.2 Task and message scheduling.* At the second level of detail, all information about the distribution of software components to ECUs is known. Also, the underlying network is specified, the runnable entities are mapped to tasks, and the communication signals are mapped to messages on the bus. This means that all the information is available to create the correct amount of ECU models, the configuration of the operating system, the configuration of the COM module, a partial configuration of the interface and driver module (the frame ID of the message), and the bus that connects the different ECU models.

Figure 7(b) shows the configuration of the ECU model. The low-level details on how the messages are handled by the buffers are not yet known at this stage. This means that the interface and driver modules of the ECU model cannot be configured. The module is configured with a standard configuration that delivers the frame to the transmit buffer without any configuration. However, the interface or buffer modules can already have a time advance. This time advance reflects the time normally needed to store the message in the buffer, but could also contain a probabilistic component to take jitters from software buffering into account.

A new transmit buffer atomic DEVS model is created that supports an infinite buffer capacity. The new transmit buffer model has the exact same ports and functionality of the one presented in Section 4 but does not include the design choices for the number of buffers and other related choices. The infinite buffer function to select the next message to be transmitted on the bus has two options: (a) FIFO, select based on arrival time; (b) Priority, always select the highest-priority message.

*5.2.3 Full deployment.* Finally, in the last level of detail, all design decisions with respect to the AUTOSAR basic software are made. Communication buffers are a constrained resource in many ECUs. The limited amount of buffers need to be configured as either receiving or transmitting buffers. However, buffers can be shared between different messages, a transmit buffer can transmit different frame IDs, while a receive buffer can be used to receive multiple messages. The model presented in Section 4 has all the capabilities needed to simulate the full deployment of a software-intensive system.

## 5.3 Fault injection

Fault injection is a general technique used in different simulation tools.[8,20,21] While it is beyond the scope of this paper to provide the reader with an extensive survey of fault-injection techniques, we briefly look at two techniques for adding faulty behavior in a simulation.

- *Simulator commands*. The technique is based on using the simulator or simulation framework to modify the values of the variables in the model. The main advantage of this technique is that it does not require any changes to the model. The application, however, depends on the underlying functionalities of the simulator. This approach has been implemented in the DEVSimpy framework[22,23] but it is not available in our PyDEVS simulator. Similarly, other variants of the DEVS formalism have been created to add faulty behaviors to DEVS models, for example the BFS-DEVS formalism,[24] which allows concurrent fault simulation with the DEVS formalism.
- *Model modification*. The model can be modified to include injections directly. Mutants and saboteurs can be inserted into the model to change the behavior. Mutants with the DEVS formalism have been previously used by Zia et al.[25] to model the behavior of a pump control system.

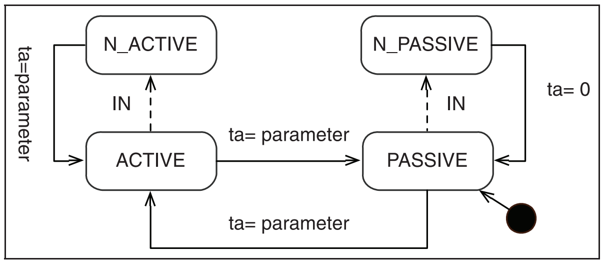

**Figure 8.** State diagram of serial saboteur model.

In the AUTOSAR model, the behavior of the different runnable entities can be augmented with errors. The mutants of the runnable entities allow us to look at the behavior of our system under the influence of software bugs. However, the effects on the behavior of the system can already be verified at a higher abstraction level when designing and testing the software. Because we are interested in the system behavior, there are other, more interesting, fault injections to add. Sensors and actuators are prone to different errors, e.g., short circuits, damage, or calibration errors. The values of the sensors are provided by our environment model. A saboteur can be inserted in between the system under study and the environment and plant models. Figure 8 shows a model of a simple saboteur. The atomic DEVS model has a single input port and a single output port. The saboteur can be either *active* or *passive*. In the latter case, when an event arrives at the input port, the saboteur immediately sends the event out on its output port (using the *N_PASSIVE* state). When the saboteur is active, the piggybacked data in the event can be changed and the event can be delayed using a parameterized time advance from the *N_ACTIVE* to the *ACTIVE* state. Changing from passive to active and vice versa is also parameterizable. The saboteur allows us to mimic the effect of broken and faulty components and intermittent errors. It has the advantage that no atomic models need to be changed; only the couplings between atomic DEVS models must be changed. It also requires no change to the standard DEVS formalism or the simulator to be used for introducing faulty behavior in the model.

Introducing errors in the communication system is also a common technique. The CAN bus standard already has a mechanism to deal with faulty conditions. When something happens during the transmission of a message, the bus goes into an error state and the nodes start to transmit an error frame. This error state lasts for a maximum of 23 bits, with the inclusion of the inter-frame spacing. The aborted message remains in the transmit buffer and can again compete for the bus. In Section 4, we did not model this error state because our modeling goal did not require this level of detail. The model of the CAN bus (and also the depending buffer) must be updated to include this behavior. This is done by adding the *ERROR* state with a time advance of 14 to 23 bit-times. The notification event for the buffers is augmented with a field to indicate whether the message has been transmitted without any errors. An extra input port is added that allows the external transition from the busy state to the error state. A model to create errors on the bus must be added to the coupled model and can be arbitrarily simple or very complex, depending on the required testing scenarios. Similarly, the buffers are adapted to comply with this new event type.

## 6 Simulation model within an automotive design process

In this section, we discuss how the presented model and its features can be used in an automotive design process. The automotive industry is characterized by an integrator (OEM) and supplier relation. The OEM, as a system integrator, integrates the different components produced by the supplier companies. For more details about this relation, we refer the reader to Volpato. [26]

Automotive systems, and more general software-intensive systems, are commonly engineered using the V life-cycle model. [27] The V-model is a life-cycle process guideline for planning development projects. [28] The development process of the system is represented as a V. The time and maturity of the system follow the V-shape from left to right. The left leg of the V-model starts from system requirements over system development, higher-level subsystem development (architecture) and finally low-level subsystem development. The right leg of the V-model represents the different realization (and integration) phases of the system.

AUTOSAR allows for a top-down or bottom-up creation of the software architecture and subsequent deployment. [29] Simplified, in the top-down approach, a system architect starts by defining the software architecture with its components and interfaces. These components are further refined and afterward outsourced to supplier companies. The supplier company receives the specifications of the component (including a worst-case execution time constraint). In the bottom-up approach, a large collection of legacy components is used to create an architectural model.

Components in the automotive process are tested at different levels of abstraction and at different levels of integration. [27] Different integration platforms are used during the verification process.

- *Model-in-the-loop*. The system is fully realized in models. The model of the software is simulated. The simulation is either (a) open-loop, where the control software inputs are provided by the developer to test the model, or (b) closed-loop, where models of the plant and environment are simulated together with the model of the control.
- *Control-in-the-loop*. This is also known as rapid control prototyping. The model is run on a (high-end) computer that is embedded in the real car. The real car acts as the environment and plant model for the control model.
- *Software-in-the-loop*. The fixed point source code of the software is run in open-loop, or closed-loop together with the plant and environment.
- *Processor-in-the-loop*. The software runs on the target processor or target processor simulator. The plant and environment are simulated.
- *Hardware-in-the-loop*. The software runs on the target hardware; the plant and environment is simulated. Debugging of the control model is much harder with this test execution platform.
- *The car*. All components are fully realized and integrated in the vehicle.

This research focuses on the use of a virtual platform on which to deploy the software and thus complements these X-in-the-loop approaches. It specifically targets the integration of the different control algorithms on networked control units. The simulation model can thus be seen as a model-in-the-loop for integration. The model can be used for rapid control prototyping if it is executed in real time together with the behavior components. The car replaces the co-simulation with the plant and environment models. However, this is beyond the scope of this paper and is considered a topic for future work. To be usable as model-in-the-loop for integration, the OEM and supplier must have the capability to create this virtual ECU model with ease.

### *6.1 Simulation model creation and traceability*

The presented DEVS model in Python is not the most appropriate level of abstraction to design AUTOSAR-based systems. More appropriately are domain-specific languages and tools that allow the creation of such an AUTOSAR model, and the behavior of each of the components. These tools are commonly referred to as AUTOSAR authoring tools. However, these tools have little or no simulation capabilities. It is, therefore, necessary to translate the model created in the authoring tool to the simulation model. Model transformation technology allows for a translation between these different types of model. The simulation model has to be initialized with all the design choices, such as the number of ECUs, the tasks and runnable entities per ECU, and all of its properties, e.g., the task priorities. Template-based model-to-text transformations help in creating a simulation model from the model defined in the authoring tool. For example, the MOF model-to-text transformation language is such a template-based transformation language. [30] The transformation starts by generating all the components of the ECU model and configuring the ECU model based on the design choices made by the designer in the authoring tool. This code is generated for each of the ECUs present in the AUTOSAR model. Finally, a coupled model is constructed using these components.

Each element created in an AUTOSAR authoring tool is, in compliance with the AUTOSAR standard, identified with a global unique identifier or GUID. This GUID has the advantage that we can trace back different errors and configurations within the simulation model, back to the components in the authoring tool. This traceability is necessary to allow for the integration of a DEVS-based simulation environment within the automotive development process.

## 6.2 Simulation calibration

Before the simulation model can be used in practice, it must be calibrated (i.e., parameter values must be estimated). Since we focus on real-time behavior, time delays for all actions in the simulation model must be measured on the used hardware platforms. Here are some of the measurements or estimations that must be completed:

- execution time of all the runnable entities or states in the runnables, without the calls to the run-time environment;
- execution time of activating or suspending tasks, as well as the context switching times;
- execution time of the transmission and reception of messages in every part of the communication stack, including the run-time environment;
- amount of time needed to handle each type of interrupt.

The timing analysis of the basic software components must be done once for each hardware processor used in the design. For the runnables, we can differentiate between the top-down approach, which results in timing constraints, and the realized components, which need timing analysis to verify if they comply with the constraints.

In the top-down approach, timing requirements are decomposed until the component level is reached. These timings are the constraints that are put on the design of these components. AUTOSAR supports the modeling of these timing constraints and decompositions using the AUTOSAR Timing Extension (TIMEX). [31] Because AUTOSAR is only part of the whole design process, other tools are also used in the process for this decomposition, for example, the MARTE profile for UML and SysML. [32]

In the bottom-up approach and with the realized components in the top-down approach, the timing of the different runnables can be measured or analyzed. This gives the modelers either a worst-case execution time or a timing profile of the runnable on the chosen hardware. The embedded systems community has spent significant efforts in the measurement and analysis of timing behavior of source code on a processor. A complete overview can be found in Wilhelm et al.[33]

The execution times in our simulation model can be sampled from a distribution of execution times based on a scenario or the worst-case execution times. This is dependent on the interest of the developers.

## 6.3 Original equipment manufacturer

The OEM can use the presented model in both the top-down and the bottom-up approach. The simulation model complements the use of analysis techniques at different phases of the V-model. At the architectural design level, it allows for design choices to be made and trade-off analysis. Because the behavior of the components is not yet designed, the simulation model can be used to look at the behavior of the embedded system in open-loop. Co-simulation with the plant, fault injection, etc., is not used at this stage. Engineers can use what-if analysis and design-space exploration to create a good software and hardware architecture.

What-if analysis is used to experiment with parameters of the model. The modeler experiments with some parameters of the model to try and evaluate the outcome of such a change, hence the name what-if. A simulation model is excellent for this purpose because it returns quantifiable results that the engineer can use to make decisions. Similarly, design-space exploration tries to look at different design alternatives to help designers select an optimized design with respect to a goal function. Automatic design-space exploration techniques create a set of solutions and evaluate these with respect to the goal function. The simulation model can be used for this evaluation, as shown by Denil et al. [34]

Once the supplier has provided the behavior models or (software and hardware) components, the OEM can use the simulation model in the different integration phases of the V-model. The model is simulated in closed-loop to check and optimize the behavior of the whole system. Finally, the simulation model provides OEMs with the capability to explore the impact of new components and upgrades to current vehicles.

## 6.4 Supplier

The supplier can use the model at the different levels of abstraction during the design, validation, and verification phase. The design of a single subsystem also follows a system life-cycle process, for example, the V-model. The functional verification of the subsystem architecture and models can be made using the simulation model at the virtual functional bus level. Other, more integrated, simulations are run when the components are developed.

Because the supplier needs to test the components under real operating conditions, techniques like rest-bus simulation need to be supported. Rest-bus simulation simulates the rest of the network to create the normal operating conditions for the subsystem. It is also used, when developing a subsystem, to generate the inputs of the model that originate in another subsystem, the environment of the model. The information for system-wide communication is often described using standard interchange formats; examples are DBC-files or Fibex [35] for automotive networks. All information about messages, signals, data-types, etc., is described in the interchange file. The file is often called a communication matrix and is (partially) shared with subcontractors.

An atomic DEVS generator model is used to generate the frames at the correct instances. The transmit buffer, with infinite capacity, is used to store the frames in the buffer and transmit them on the bus. A model transformation from the communication matrix in the DBC-format or Fibex format can be used to generate the atomic DEVS model automatically.

## 7 Case study

In this section, we use the constructed simulation model to deploy a power window. Power windows are automobile windows that can be raised and lowered by pressing a button or switch, as opposed to using a hand-turned crank handle. Such devices exist in the majority of automobiles produced today. The basic controls of a power window include raising and lowering the window. An increasing set of functionalities is being added to increase the comfort, safety, and security of vehicle passengers. To manage this complexity while reducing costs, automotive manufacturers use software to handle the operation and overall control of such devices. However, as a power window is a physical device that may come into direct contact with human beings, it becomes imperative that sound construction and verification methods are used to build such software.

Safety requirements of the power window system are detailed by government bodies, such as the Road Safety and Motor Vehicle Regulation Directorate of Transport Canada. [36] They address safety issues of the power window, for example, the maximum force that may be exerted on an object by a window going up. Other requirements of the system are not safety requirements and are thus not addressed by the governing bodies. These are requirements originating in the features that a company wants to present to its customers. Prahbu and Mosterman define some textual requirements of the power window system.[37] We adapted these requirements to take safety requirements[36] into account.

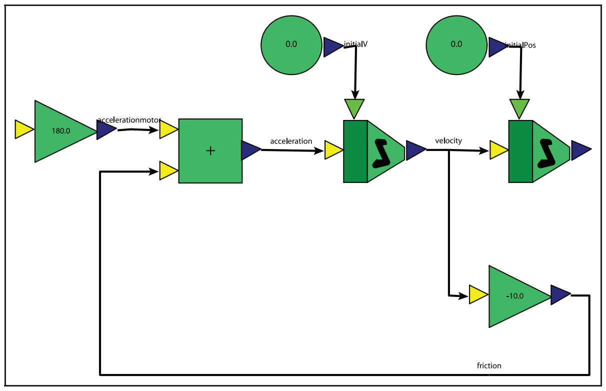

**Figure 9.** Power window plant model using causal-block diagrams.

1. The window must start moving within 200 ms after a move command is issued.
2. The window must be fully opened or closed within 4 s.
3. The force to detect when an object is present should be less than 100 N.
4. When an object is present, the window should be lowered by approximately 12.5 cm.
5. The window can only be operated when in the ‘‘start,’, ‘‘on,’’ or ‘‘accessory’’ position.
6. Driver commands have precedence over passenger commands.

The power window system has all the essential complexity typical of a software-intensive system. A physical window has to be moved within certain real-time bounds, while information on the detection of an object is fed back to the control component for the safety requirement. It also has a distributed nature, since the controller sensing the interactions of the driver is physically on the other side of the car. Hence, information must be transmitted using a communication medium that connects the different control units.

The supplier receives the specification of the power window together with the messages and priorities of messages on the CAN bus. He also receives access to a communication matrix to simulate the power window in real operating conditions. During the design of this software-intensive system, different models are created, starting with the requirements. From these requirements, design models are created, not only to control the power window, but also for the physical window and motor (the plant model). A block diagram model of the power window plant is shown in Figure 9. The model receives a motor command from the control model. The motor uses this command to raise or lower the window. Two integrator blocks are used to determine the position of the window based on the acceleration given by the motor. The window also has a friction component based on the velocity of the

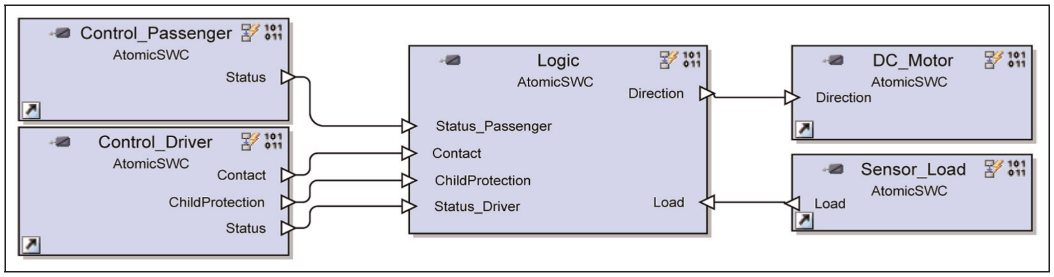

**Figure 10.** AUTOSAR software component diagram of power window case study.

window. Similarly, block diagrams are also used to describe the scenarios to test the models. As an example we use a single test-scenario that will raise the window. In this scenario, the driver issues a command to raise the window. After 3 s, an object is present between the window and the window frame. This results in an excessive force. The driver releases after 3.5 s. No commands are issued for another 3.5 s. Finally, the driver issues a command to lower the window. A control model using Statecharts is used to model the logic of the window controller. These models can be simulated together to verify the behavior of the window. However, no delays are taken into account, since transitions in Statecharts are instantaneous. We will go through the different simulation models created for the power window during the deployment process. More information about the design process of the power window can be found in Mustafiz et al.[38]

## 7.1 Virtual functional bus

During the deployment of the power window controller, an AUTOSAR software architecture must be modeled. Figure 10 shows the component diagram of our power window controller.

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 to the power window and checking whether the ignition of the car is on. The *Load_Sensor* component reads out the resistive force placed on the window. When the execution of the runnables inside these components is finished, they make the sensor values available to the *logic* component that decides how to control the window using these sensor values. A code generation step of the model describing the logic is needed to provide the behavior of the *logic* component. Finally, the *DC_Motor* component uses the output to control the window physically.

We configured the AUTOSAR model to use a timing event of 100 ms to trigger the execution of the *Control_Driver* component. The *Control_Passenger*, *Load_Sensor* and *logic* components are triggered by a data-receive event of the state signal, transmitted by the driver. The *DC_Motor* also uses a data-receive event to trigger the execution, but uses the direction signal for this purpose.

This information is used to create the virtual functional bus model, described to check whether the behavior of this model complies with the functional requirements. Other extrafunctional properties cannot yet be checked because the virtual functional bus model does not use timing information to include delays of computations and delays of communication. Figure 11 shows the results of the simulation. As expected, the windows starts to move up until the object is present, shown by the position signal, because of the driver signal. When the object is detected by the sensor, the window control switches to the emergency state and lowers the window. Finally, when the down button is pressed, the window starts moving down. The PlantIn signal shows the driving signal to the electromotor.

## 7.2 Task and message scheduling

Extra information is added to the AUTOSAR model for task and message scheduling. A major design decision is the mapping of the components to the different ECUs. On the driver side, a single task executes the *DriverControl* runnable entity. This task transmits three signals on the bus, mapped to two different messages. The arrival of two signals in the communication stack causes 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: a high-priority task, executing all the runnable entities, and a lower-priority task, executing the *DCMotor* runnable. The priorities of the tasks and messages are chosen based on experience in building a power window system, though in a normal design process schedulability analysis and design-space exploration techniques are used to create a valid configuration.

Once this information is available, the simulation model for the second abstraction level is constructed. Timing information is added to include the delays caused by computation of the runnable entities and part of the communication stack, as well as the delay of the messages on the

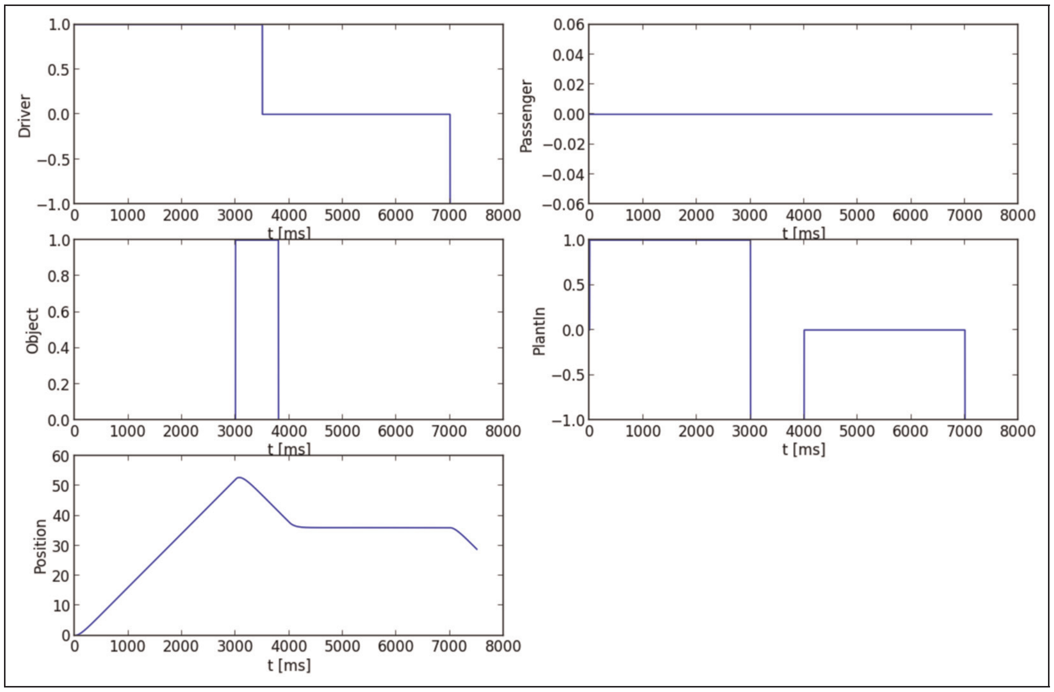

**Figure 11.** Virtual functional bus simulation results.

**Table 2.** Communication matrix for the other messages on the bus.

| Frame ID | Period, ms | Offset, ms | Size, byte |
|---|---|---|---|
| 2 | 10 | 1 | 8 |
| 3 | 10 | 1 | 7 |
| 4 | 10 | 1.5 | 8 |
| 5 | 15 | 1.15 | 8 |
| 8 | 50 | 2.15 | 4 |

bus. The interface model and driver model are not yet configured, meaning that no messages can be lost as a result of locked buffers or errors in the configuration. Rest-bus simulation is already included to include the delays of the other communication messages on the bus. Table 2 shows a partial communication matrix of the shared communication bus. This configuration is used to create the rest-bus component in our simulation.

The results of the behavior are similar to Figure 11. To inspect the timing behavior, it also possible to look at the timing traces of the simulation. Figure 12 shows a visualization of this trace. In this figure, the timing behavior is depicted as a Gantt diagram. All delays due to computations of the runnable entities, the partially configured communication stack, and the bus are included. Because of the priority mechanism of the CAN bus, the message to transmit the status of the driver button (frame ID 10, underlined) has to wait its turn. However, this does not influence the overall behavior of our system in a significant way. The prescribed end-to-end latency of the power window is met. The configuration is thus valid.

## 7.3 Full deployment

The final step in the deployment of our power window system is making the last design choices related to the communication stack. The number of available hardware buffers must be divided between transmit and receiver buffers.

In a first attempt, we assigned a single transmit buffer to send both messages from the driver ECU to the passenger ECU. Figure 13 shows the behavior of the system using this configuration. The behavior is not what we expected. The trace of our simulation model indicates that the transmit message buffer is still locked when the other message arrives at the transmit buffer. We did not enable any software buffering in our driver; this results in a lost message.

Figure 14 shows a valid configuration. By utilizing a second buffer to transmit the frame, the message is correctly transmitted to the passenger ECU. A viable alternative is to enable software buffering in the CAN interface configuration. This will store the second message in a software buffer until the first frame has been transmitted. A notification from the buffer to the ECU model results in an interrupt to the processor. During this interrupt, the frame is moved from the software buffer to the hardware buffer.

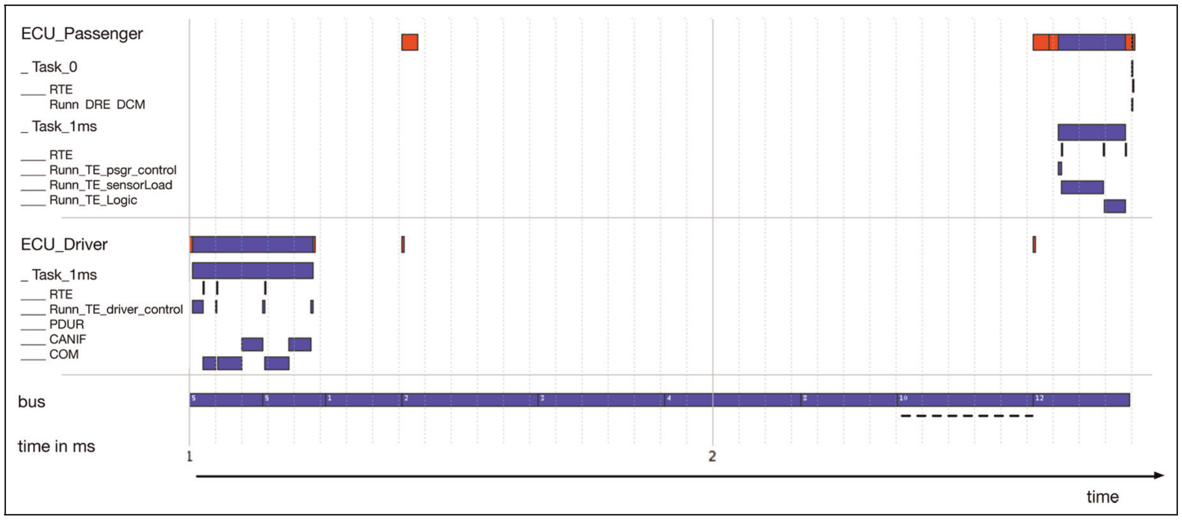

**Figure 12.** Visualization of timing behavior trace.

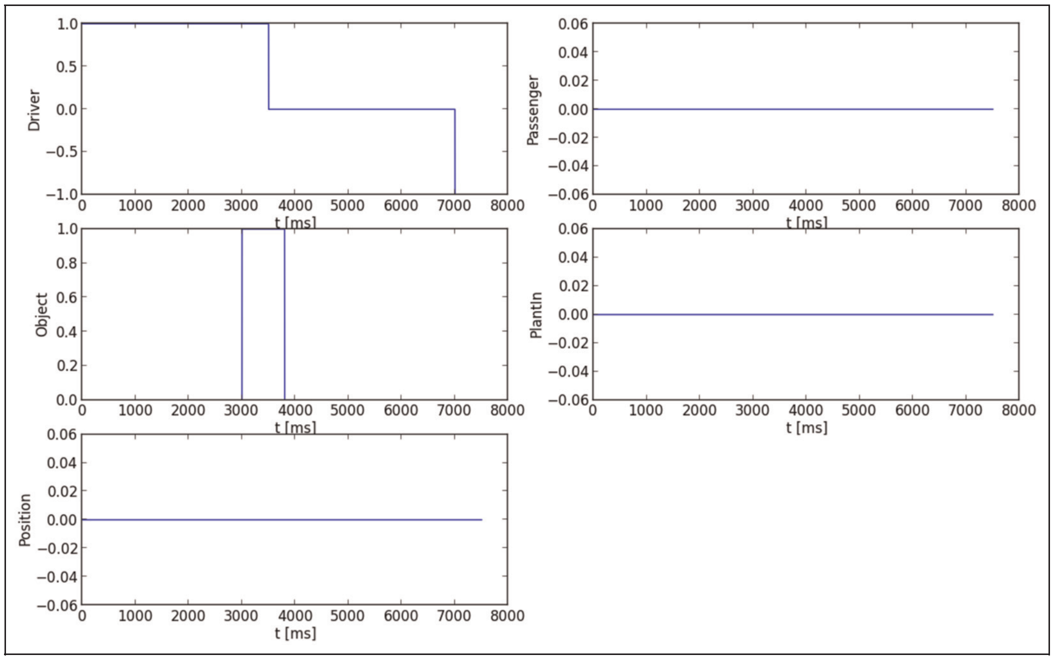

**Figure 13.** Resulting behavior using a wrong interface–driver configuration.

The behavior of our power window system is again similar to the graph shown in Figure 11.

## 7.4 Fault injection

To test our created system under faulty conditions, two fault injections are added to our model. The first one introduces errors during the transmission of the second CAN frame. Figure 15 shows the resulting behavior on the bus. The introduction of the error during the transmission results in an error state on the bus. The second message, however, has to compete for the bus when the error state is finished. However, a higher priority message is already available. This creates an added delay to the response time of the window but is negligible as a whole. The behavior of our window is still within the bounds of the requirements.

Our second injection introduces a saboteur component between the sensor to measure the resistive force on the bus and the passenger ECU. The saboteur simulates a

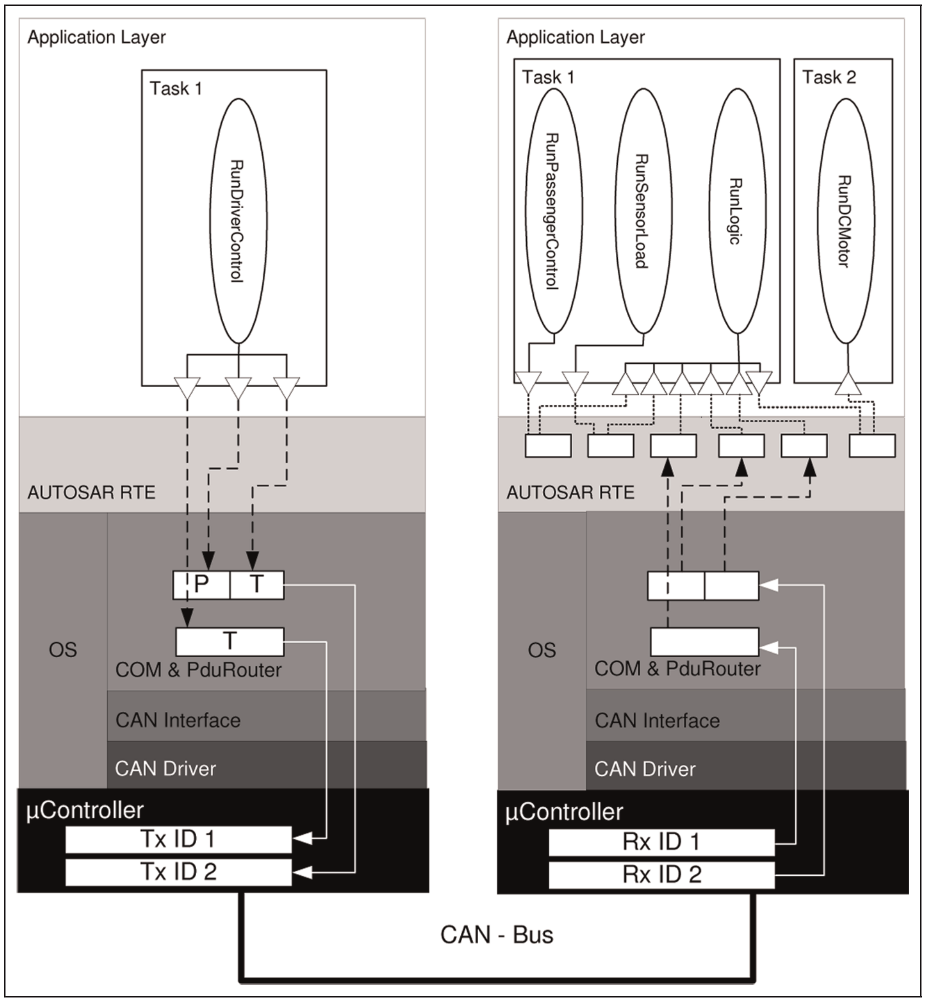

**Figure 14.** Power window application deployed on the hardware. P(ending) is a signal that does not cause the message to be transmitted, in contrast with the T(riggered) signals. Signal and message names are removed for of clarity.

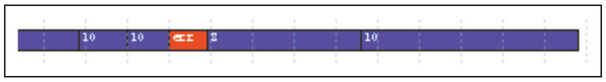

**Figure 15.** Behavior of CAN bus with errors.

Byzantine failure [39] of the sensor, i.e., the sensor produces erroneous results when the saboteur is in the active state. Figure 16 shows one of the experiments with the introduced saboteur. After ≈1 s, the window behavior changes because of the faulty sensor. The window controller detects a high resistive force on the window while no resistance is present. The window starts moving down. For the customer, this would result in an uncomfortable situation but not an unsafe behavior. However, at the third second, there is a resistive force on the window but the sensor fails to detect this. The window continues to go up. This results in an unsafe state. Other mechanisms must be introduced to ensure safe operation of the window under faulty conditions.

## 7.5 Calibration of the model

We used the technique presented by Denil et al. [40] to calibrate the execution times of our runnable entities. The technique uses the plant, environment, and control models to generate a calibration infrastructure. The source code of the control models is instrumented so that time measurements can be made during the execution of the model. The same scenario is used during the calibration phase. The other parameters are estimated based on hardware measurements.

## 7.6 What-if analysis

Now that the full component is created at the supplier side, the OEM can integrate the power window. The system integrator deploys messages of lights and blinkers together with the messages of the power window on the same CAN bus. The behavior of the deployed window in this context was already partially evaluated through rest-bus simulation.

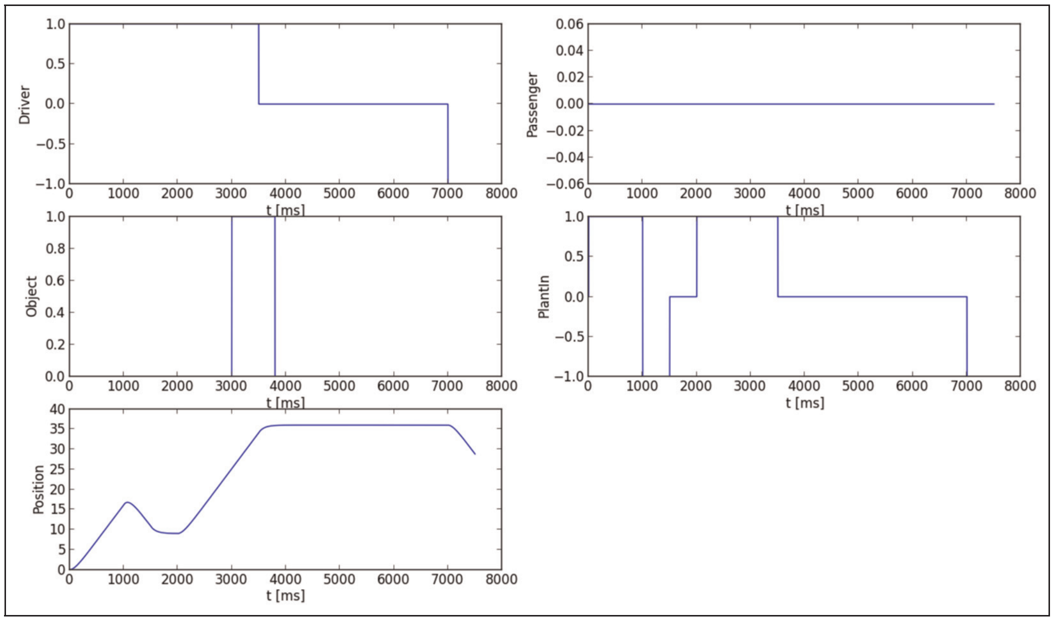

**Figure 16.** Result of a fault injection.

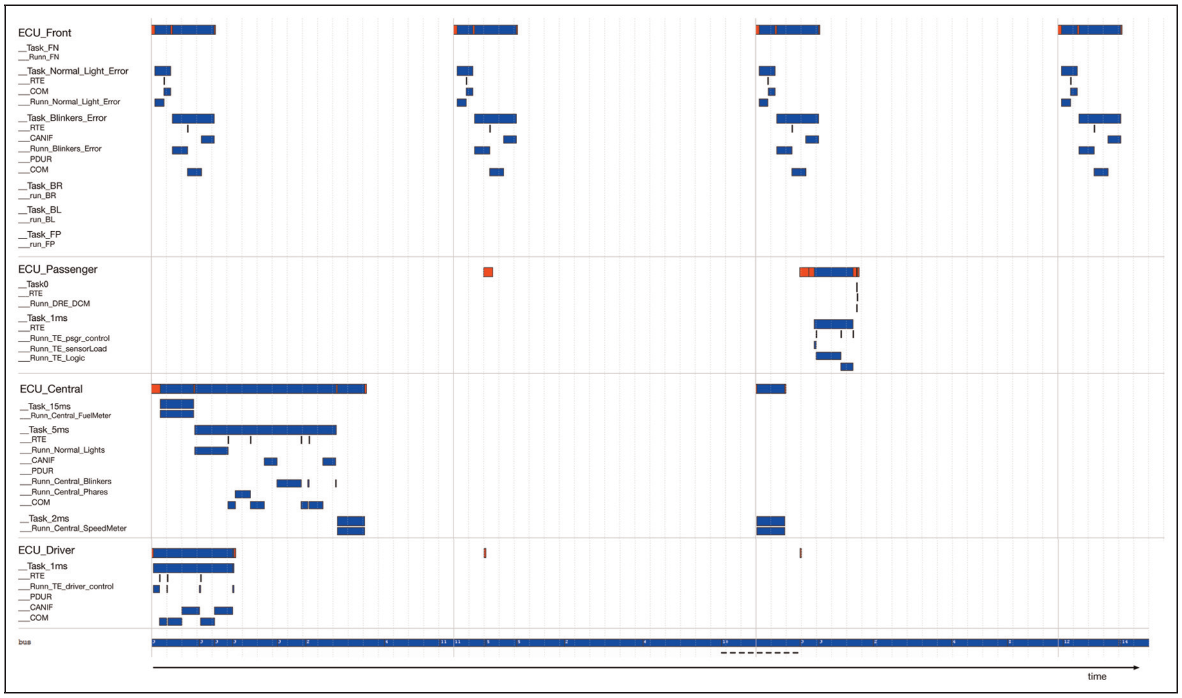

**Figure 17.** Original equimpment manufacturer integrating different ECUs and software components.

The controller of the turning lights receives a message from the dashboard controller to enable or disable the blinking lights. The turning lights controller periodically checks the physical condition of the lamp by means of a current measurement. Only when the lamp is broken, is a diagnostic message returned to the dashboard. This behavior was not specified by the communication matrix that was given by the OEM to the supplier. The message is transmitted periodically with a very high priority. The simulation trace of such a scenario is depicted in Figure 17.

As can be observed, CAN message 10 (underlined) is delayed for more than 1 ms, compared with the previous trace in Figure 12. It does not affect the prescribed deadline of the power window so there is no problem there. However, by analyzing the trace of the simulation further, the engineer observes that the message for the blinkers never arrives at its destination and that the blinkers are no longer enabled or disabled on user request. The engineer notices that this is because of the behavior of this high-priority message. Another hint of this problem is that the message buffers of the different components are showing errors because of overwritten messages in the output buffers. The engineer can use what-if analysis to, for example, change the priority of the message, the frequency of the polling, etc., to correct this wrong behavior.

## 8 Related work

To evaluate AUTOSAR-based systems, both analysis techniques and simulation models are available at different abstraction levels.

On the analysis side, techniques are available to predict the timing behavior after deployment. A well-known technique is schedulability analysis, which uses the worst-case timing behavior of the different components to check whether an application meets its deadlines. A multitude of schedulability analysis techniques for different application domains, schedulers, and buses is described in the literature. [41–46] These mathematical models focus on the time and order of the scheduling of tasks and messages. Schedulability analysis provides a method to verify whether the end-to-end deadlines of the application are met. Schedulability analysis does not focus on the behavior of the application. The simulation models we propose do not replace schedulability analysis but can be used as an augmentation to verify the behavior of the fully integrated system under different timing constraints.

Different commercial AUTOSAR tool vendors support the simulation of software components at the virtual functional bus level. For example, the DSpace SystemDesk and Vectors DaVinci tools are able to simulate the designed software components on the virtual functional bus. For this, all software components are mapped to a single virtual ECU. The formalisms used to model this virtual ECU are unknown. Using techniques like the functional mock-up interface,[47] the simulation of the virtual platform can be connected to another tool to simulate the environment and plant models. Our approach is able to co-simulate different formalisms. We do not need the functional mock-up interface standard for this because we use the DEVS formalism as a common denominator for the simulation of heterogeneous models.

Krause et al. [48] evaluated SystemC as a language for modeling and performance evaluation of AUTOSAR-based software. Krause et al. [49] also introduced different levels of detail, where SystemC can be used to evaluate the timing behavior of the application. The levels of detail proposed are similar to our own. An extra level with cycle-accurate simulation is also introduced where the real middleware and application code is executed on a cycle-accurate simulation model of the processing unit. The cycle-accurate approach is computationally expensive and not recommended by the authors. The integration of plant and environment models is not discussed by the authors. Similarly, we evaluate DEVS as an appropriate formalism for the virtual prototyping of AUTOSAR deployment.

Another approach involves incorporating the effects of scheduling in Simulink® models. [50] In TrueTime, all application components are simulated in combination with operating system and communication bus blocks, mimicking the delays due to the communication hardware and the operating system scheduler. Similarly, Vanherpen et al. [51] use the technique of schedulability analysis to lift extrafunctional properties to the behavioral model in Simulink. By round-tripping this design information, information about delays is woven into the Simulink application model. The domain expert can use this information to predict and adapt the model. 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 in combination with the plant and environmental models in causal-block diagrams.

More generally, different modeling, analysis, and simulation frameworks for software-intensive systems have been proposed in the literature. Metropolis [52] is an interdisciplinary research project that develops a design methodology, supported by a comprehensive design environment and tool set, for embedded systems. Metropolis is able to devise a simulation model from the defined model in the Metropolis language.[53] This simulation model is written in Java or C++. The underlying code is specific to the Metropolis approach. We propose to use a well-known general-purpose simulation formalism, DEVS, for the simulation of deployed software-intensive systems. A more complete overview of tools that support the deployment of applications on platforms is given by Törngren.[54]

The DEVS formalism has also been used to develop embedded real-time applications. Wainer et al.[55,56] introduce a model-driven method to develop these real-time embedded applications. The authors show that the use of DEVS improves reliability, promotes model reuse, and permits the reduction of development and testing times for the overall process. Finally, DEVS-like operating systems have been proposed to close the gap between modeling and deployment of software-intensive systems.[57–59]

## 9 Conclusions

In this article, we compared the requirements for the modeling and simulation of the deployment of automotive

AUTOSAR-based systems with the characteristics of the DEVS formalism. It is shown that DEVS is an appropriate formalism to model the behavior of AUTOSAR-based automotive systems. The DEVS formalism provides an intuitive manner to model the timing behavior of the application deployed on the system architectures used in automotive systems. Because DEVS is a common denominator for simulating different formalisms, the model can be co-simulated with the plant and environment models so a full assessment of the behavior of the system is possible. Finally, such techniques as fault injection and rest-bus simulation can easily be added to the model because of the modularity and compositionality of the DEVS formalism.

To support this reasoning, we constructed a generic simulation model for the automotive domain. The simulation model is able to simulate the deployment of automotive applications on the AUTOSAR basic software and CAN bus. By using model transformation, the simulation model is automatically constructed based on the modeling artifacts of the deployment process. The model is usable at different stages of the design process because different abstraction levels are possible. This gives early feedback to the designer when making design choices related to the deployment. The results obtained by using this simulation model will help the AUTOSAR developer to analyze the impact of different choices on the behavior of the system. It can be used to explore various trade-offs while deploying automotive applications to AUTOSAR-based ECUs.

Finally, we used the simulation model to evaluate the virtual functional bus and deployed behavior of the software of a power window controller on the AUTOSAR platform and CAN bus. The plant and environment are expressed using the causal-block diagram formalism, which is co-simulated with our deployed models. At different approximation levels, design choices are evaluated with respect to the global behavior of the system under realistic conditions. This is achieved by using a rest-bus simulation of a CAN communication matrix. Faults are injected in the CAN bus and at the sensor outputs to investigate the fault tolerance of our constructed model. Furthermore, we looked at the behavior of the window when integrated with other components.

Because of the properties of the DEVS formalism, the simulation model can be further extended with other communication buses and components without any problems. In future developments, multicore processors will become the dominant execution platforms for automotive systems. In a first stage, the multicore processors will feature their own operating systems and act as redundant components. The current simulation model can be used to model these situations. When the multicore processors are used by a single operating system to schedule different tasks, a new atomic model of this operating system should be constructed. This is a topic for future work.

## Acknowledgments

The authors would like to acknowledge the three anonymous reviewers whose comments and suggestions helped improve and clarify this manuscript.

## Funding

This research was partially supported by Flanders Make vzw. This work is also supported in part by the Natural Sciences and Engineering Research Council of Canada as part of the NECSIS Automotive Partnership with General Motors, IBM Canada, and Malina Software Corp.

## References

1. Broy M. Challenges in automotive software engineering. In: *Proceeding of the 28th international conference on Software engineering—ICSE '06*, Shanghai, China, 20–28 May 2006, p.3. New York: ACM.
2. Lee E. Cyber physical systems: Design challenges. In: *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*, Orlando, FL, 5–7 May 2008, pp.363–369. Los Alamitos, CA: IEEE.
3. Boucher M and Kelly-Rand C. System design: Get it right the first time. Technical Report, Aberdeen Group, August 2011.
4. Sangiovanni-Vincentelli A and Di Natale M. Embedded system design for automotive applications. *Computer* 2007; 40: 42–51.
5. Di Natale M, Giusto P and Sangiovanni-Vincentelli A. Stochastic analysis of CAN-based real-time automotive systems. *IEEE Trans Ind Inf* 2009; 5: 388–401.
6. Lyons RE and Vanderkulk W. The use of triple-modular redundancy to improve computer reliability. *IBM J Res Dev* 1962; 6: 200–209.
7. ISO 26262:2011 Road vehicles—Functional safety— Part 10: Guidelines on ISO 26262.
8. Ziade H, Ayoubi R and Velazco R. A survey on fault injection techniques. *Int Arab J Inf Technol* 2004; 1: 171–186.
9. Köhl S and Jegminat D. How to do hardware-in-the-loop simulation right. SAE paper 2005-01-1657(724), 2005.
10. Van Tendeloo Y and Vangheluwe H. The modular architecture of the Python(P)DEVS simulation kernel: Work in progress paper. In: *Proceedings of the symposium on theory of modeling & simulation—DEVS integrative*, Tampa, FL, 13–16 April 2014, pp.14:1–14:6. San Diego, CA: Society for Computer Simulation International.
11. Denil J, Vangheluwe H, Ramaekers P, et al. DEVS for AUTOSAR platform modelling. In: *Proceedings of the 2011 symposium on theory of modeling & simulation: DEVS integrative M& S symposium*, Boston, MA, 3–7 April 2011, pp.67–74. San Diego, CA: Society for Computer Simulation International.
12. Zeigler BP. *Multifaceted modelling and discrete event simulation*. Cambridge, MA: Academic Press, 1984.
13. Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In: *Proceedings of the IEEE international symposium on computer-aided control*

*system design (CASC)*, Anchorage, AK, 25–27 September 2000, pp.129–134. Piscataway, NJ: IEEE.

14. AUTOSAR. www.autosar.org (2012, accessed: 15 June 2012).
15. OSEK. OSEK operating system v.2.2.3. http://web.archive.org/web/20120204070317/http://www.osek-vdx.org/ (2005, accessed: 1 November 2013).
16. Farsi M, Ratcliff K and Barbosa M. An overview of controller area network. *Comput Control Eng J* 1999; 10: 113–120.
17. Makowitz R and Temple C. FlexRay—a communication network for automotive control systems. In: *2006 IEEE international workshop on factory communication systems*, Torino, Italy, 28–30 June 2006, pp.207–212. Piscataway, NJ: IEEE.
18. Press W, Teukolsky S, Vetterling W, et al. *Numerical recipes in C: the art of scientific computing*. Cambridge, UK: Cambridge University Press, 1992.
19. Cormen T, Leiserson C, Rivest R, et al. *Introduction to algorithms*. Cambridge, MA: MIT press, 2001.
20. Gil D, Baraza JC, Gracia J, et al. VHDL simulation-based fault injection techniques. In: Benso A and Prinetto P (eds) *Fault injection techniques and tools for embedded systems reliability evaluation*. Dordrecht: Kluwer, 2004, pp.159–176.
21. Lu W and Radetzki M. Concurrent and comparative fault simulation in SystemC and its application in robustness evaluation. *Microprocess Microsyst* 2013; 37: 115–128.
22. Santucci JF and Capocchi L. A proposed evolution of DEVSimPy environment towards activity tracking. In: *ACTIMS workshop*, May 28–1 June 2012, pp.1–10. Cargese, Corsica: HAL-CCSD.
23. Capocchi L, Santucci J, Poggi B, et al. DEVSimPy: a collaborative Python software for modeling and simulation of DEVS systems. In: *2011 IEEE 20th international workshops on enabling technologies: infrastructure for collaborative enterprises*, Paris, France, 27–29 June 2011, pp.170–175. Piscataway, NJ: IEEE.
24. Capocchi L, Bernardi F, Federici D, et al. BFS-DEVS: A general DEVS-based formalism for behavioral fault simulation. *Simul Modell Pract Theory* 2006; 14: 945–970.
25. Zia M, Mustafiz S, Vangheluwe H, et al. A modelling and simulation based process for dependable systems design. *Software Syst Model* 2007; 6: 437–451.
26. Volpato G. The OEM-FTS relationship in automotive industry. *Int J Automot Technol Manage* 2004; 4: 166–197.
27. Zander J. *Model-based testing of embedded systems in the automotive domain*. Ph.D. Thesis, Technical University Berlin, Germany, 2009.
28. Blanchard BS. *System engineering management*. Hoboken, NJ: John Wiley & Sons, 2004.
29. Sandmann G and Thompson R. Development of AUTOSAR software components within model-based design. SAE paper 2008-01-0383, 2008.
30. OMG. MOF model to text language (MTL). Technical report, OMG, http://www.autosar.org/fileadmin/files/standards/classic/4-2/methodology-and-templates/methodology/auxiliary/AUTOSAR_TR_TimingAnalysis.pdf (2008, accessed: 15 October 2012).
31. AUTOSAR. Timing analysis v.4.2.2, http://www.autosar.org/fileadmin/files/standards/classic/4-2/methodology-and-templates/methodology/auxiliary/AUTOSAR_TR_Timing Analysis.pdf (2015, accessed: 10 July 2016).
32. OMG. UML profile for MARTE: modeling and analysis of real-time embedded Systems v.1.1, http://www.omg.org/spec/MARTE/ (2011, accessed: 1 November 2013).
33. Wilhelm R, Engblom J, Ermedahl A, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans Embedded Comput Syst* 2008; 7(3): 36.
34. Denil J, Cicchetti A, Biehl M, et al. Automatic deployment space exploration using refinement transformations. Electron Comm EASST, http://journal.ub.tu-berlin.de/eceasst/article/view/711/718 2012; 50: 1–13.
35. ASAM. ASAM MCD-2 NET, v4.1.1. www.asam.net (2014, accessed 10 July 2016).
36. Canada Transport. Power-operated window, partition, and roof panel systems. Technical report, Standards Research and Development Branch—Road Safety and Motor Vehicle Regulation Directorate, 2009.
37. Prabhu S and Mosterman P. Model-based design of a power window system: modeling, simulation and validation. In: *Proceedings of IMAC-XXII: A conference on structural dynamics*, Dearborn, MI, 26–29 January 2004. Bethel, CT: Society for Experimental Mechanics, Inc.
38. Mustafiz S, Denil J, Lúcio L, et al. The FTG + PM framework for multi-paradigm modelling: An automotive case study. In: *Proceedings of the 6th international workshop on multi-paradigm modeling*, Innsbruck, Austria, 1 October 2012, pp.13–18. New York: ACM.
39. Lamport L, Shostak R and Pease M. The Byzantine generals problem. *ACM Trans Program Lang Syst* 1982; 4: 382–401.
40. Denil J, Vangheluwe H, De Meulenaere P, et al. Calibration of deployment simulation models: A multi-paradigm modelling approach. In: *Proceedings of the 2012 symposium on theory of modeling and simulation—DEVS integrative M& S symposium*, Orlando, FL, 26–30 March 2012, paper no. 13, pp.13:1–13:8, San Diego, CA: Society for Computer Simulation International.
41. Hamann A, Jersak M, Richter K, et al. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Syst* 2006; 33: 101–137.
42. Pop T, Eles P and Peng Z. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In: *Proceedings of the tenth international symposium on hardware/software codesign*, Estes Park, CO, 6–8 May 2002, pp.187–192. New York: ACM.
43. Lakshmanan K, Bhatia G and Rajkumar R. Integrated end-to-end timing analysis of networked autosar-compliant systems. In: *Proceedings of the conference on design, automation and test in Europe*, Dresden, Germany, 8–12 March 2010, pp.331–334. Leuven: European Design and Automation Association.
44. Pop T. *Analysis and optimisation of distributed embedded systems with heterogeneous scheduling policies*. Ph.D. Thesis, Linköping University, 2007.
45. Palencia J and Gonzalez Harbour M. Schedulability analysis for tasks with static dynamic offsets. *Proceedings 19th IEEE real-time systems symposium*, Madrid, Spain, 2–4 December 1998, paper no. 98CB36279, pp.26–37. Piscataway, NJ: IEEE.

46. Tindell K and Clark J. Holistic schedulability analysis for distributed hard real-time systems. *Microproc Microprog* 1994; 40: 117–134.
47. ITEA2. Functional mock-up interface for co-simulation. Technical report 07006, Modelisar, 2010.
48. Krause M, Bringmann O, Hergenhan A, et al. Timing simulation of interconnected AUTOSAR software-components. In: *DATE*, Nice, France, pp.474–479. Piscataway, NJ: IEEE.
49. Krause M, Bringmann O and Rosenstiel W. Verification of AUTOSAR software by SystemC-based virtual prototyping. In: Ecker W, Müller W and Dömer R (eds) *Hardware-dependent software*. New York: Springer, 2009, pp.261–293.
50. Henriksson D, Cervin A and Årzén K. TrueTime: Simulation of control loops under shared computer resources. *IFAC Proc Vols* 2002; 35: 417–422.
51. Vanherpen K, Denil J, Vangheluwe H, et al. Model transformations for round-trip engineering in control deployment co-design. In: *Proceedings of the symposium on theory of modeling & simulation: DEVS integrative M& S symposium*, Alexandria, VA, 12–15 April 2015, pp.55–62. San Diego, CA: Society for Computer Simulation International.
52. Balarin F, Watanabe Y, Hsieh H, et al. Metropolis: an integrated electronic system design environment. *Computer* 2003; 36: 45–52.
53. Balarin F, Lavagno L, Passerone C, et al. Concurrent execution semantics and sequential simulation algorithms for the Metropolis meta-model. In: *Proceedings of the tenth international symposium on hardware/software codesign CODES '02*, Estes Park, CO, 6–8 May 2002, pp.13–18. New York, NY: ACM.
54. Törngren M, Henriksson D, Redell O, et al. Co-design of control systems and their real-time implementation—a tool survey. Technical report. Report no. TRITA-MMK 2006:11, 2006. Stockholm: Department of Machine Design, Royal Institute of Technology.
55. Wainer G, Glinsky E and MacSween P. A model-driven technique for development of embedded systems based on the DEVS formalism. In: Beydeda S, Book M and Gruhn V (eds) *Model-driven software development*. Berlin: Springer-Verlag, 2005, pp.363–383.
56. Wainer G. DEVS modelling and simulation for development of embedded systems. In: *Proceedings of the 2015 winter simulation conference*, Huntington Beach, CA, 6–9 December, pp.73–87. Piscataway, NJ: IEEE.
57. Yu YH and Wainer G. eCD++: An engine for executing DEVS models in embedded platforms. In: *Proceedings of the 2007 summer computer simulation conference, SCSC '07*, San Diego, CA, 16–19 July 2007, pp.323–330. San Diego, CA: Society for Computer Simulation International.
58. Furfaro A and Nigro L. A development methodology for embedded systems based on RT-DEVS. *Innov Syst Softw Eng* 2009; 5: 117–127.
59. Niyonkuru D and Wainer G. Towards a DEVS-based operating system. In: *Proceedings of the 3rd ACM conference on SIGSIM—principles of advanced discrete simulation*, London, UK, 10–12 June 2015, pp.101–112. New York: ACM.

## Author Biographies

**Joachim Denil** is currently a post-doctoral researcher at the University of Antwerp. He received his Ph.D. in computer science and his B.Sc. and M.Sc. in electronics from the University of Antwerp. He received his B.Sc. in computer science from the Free University of Brussels. Joachim also pursued post-doctoral research at McGill University on the Canada-wide NECSIS project. His main research interest is the design of software-intensive and cyber-physical systems, in particular multiparadigm modeling, embedded system design, and simulation-based design.

**Paul De Meulenaere** is Professor of Automotive Engineering at the faculty of Applied Engineering of the University of Antwerp. His research is mainly oriented to software deployment onto embedded microcontroller platforms. In this area, software architectures such as AUTOSAR and OSEK are widely applied. He runs various research projects, often in collaboration with R& D divisions of mechatronic or automotive companies. He is also a member of Flanders Make, the Flemish research center for the mechatronics industry. Paul is also spokesperson for the CoSys-Lab research group, which focuses on the design of embedded technology for cyber-physical systems.

**Serge Demeyer** is a professor at the University of Antwerp and the spokesperson for the Antwerp System Modelling research group. He directs a research lab investigating the theme of software reengineering' (LORE—Lab On REengineering). Serge Demeyer is a spokesperson for the NEXOR interdisciplinary research consortium and an affiliated member of the Flanders Make Research Centre. In 2007, he received a ''best teachers award'' from the Faculty of Sciences at the University of Antwerp and as a consequence remains very active in all matters related to teaching quality. His main research interest concerns software evolution, more specifically how to strike the right balance between reliability (striving for perfection) and agility (optimizing for improvements). He is an active member of the corresponding international research communities, serving in various conference organization and program committees. He has co-authored a book*, Object-Oriented Reengineering Patterns*, and co-edited a book, *Software Evolution*. He also authored numerous peer-reviewed articles, many of them in top conferences and journals.

**Hans Vangheluwe** is a professor in the Department of Mathematics and Computer Science at the University of Antwerp in Belgium, an Adjunct Professor in the School of Computer Science at McGill University, Montréal, Canada, and an Adjunct Professor at the National University of Defense Technology in Changsha, China. He holds a D.Sc. degree and M.Sc. degrees in computer science and in theoretical physics, as well as a B.Sc. in education, all from Ghent University in Belgium. He has been a research fellow at the Centre de Recherche Informatique de Montréal, Canada, the Concurrent Engineering Research Center, WVU, Morgantown, WV, USA, at the Delft University of Technology, in the Netherlands, and at the Supercomputing and Education Research Center of the Indian Institute of Science, in Bangalore, India. He teaches courses on software design, computer architecture, modeling and simulation, and model-based systems design. At McGill University, he helped establish a new software engineering program. He heads the Modelling, Simulation and Design (MSDL) Research Lab, geographically distributed over McGill and Antwerp. He has been the principal investigator of a number of research projects focused on the development of a multiformalism theory and enabling technology for modeling and simulation. Some of this work has led to the WEST++ tool, which was commercialized for use in the design and optimization of bio-activated sludge wastewater treatment plants. He was the co-founder and coordinator of the European Unions ESPRIT Basic Research Working Group 8467 Simulation in Europe, a founding member of the Modelica Design Team (www.modelica.org), and an adviser to national and international grant agencies in Europe and North America. In a variety of projects, often with industrial partners, he applies the model-based theory and techniques of computer automated multiparadigm modeling in a variety of application domains. The adapID project, for example (funded by the Flemish government), investigated how the Belgian electronic ID card can be made more secure and privacy-preserving. In the NECSIS project (funded by the Automotive Partnership Canada), he works on making model transformation industrially usable and analyzable. He is a frequent keynote speaker at software engineering, as well as simulation, conferences. He has published over 150 peer-reviewed papers. He is Associate Editor of several Springer's journals: *Software and Systems Modeling*, *International Journal of Critical Computer-Based Systems*, *Simulation: Transactions of the Society for Computer Simulation*, and *International Journal of Adaptive, Resilient and Autonomic Systems*. His current interests are in domain-specific modeling and simulation, including the development of graphical user interfaces for multiple platforms. The MSDLs tool AToM3, developed in collaboration with Professor Juan de Lara uses metamodeling and graph transformation to specify and generate domain-specific environments. A web-based successor called AToMPM is currently under development. Recently, he has become active in multiabstraction modeling and simulation of buildings and cities.