VIRTUAL PROTOTYPING WITH FERAL – ADAPTATION AND APPLICATION OF A SIMULATOR FRAMEWORK

Tobias Braun, Dennis Christmann, Reinhard Gotzhein, Anuschka Igel
Networked Systems Group, University of Kaiserslautern, Germany
{tbraun, christma, gotzhein, igel}@cs.uni-kl.de

Thomas Forster, Thomas Kuhn
Fraunhofer IESE, Kaiserslautern, Germany
{Thomas.Forster, Thomas.Kuhn}@iese.fraunhofer.de

ABSTRACT
Nowadays, complexity and extent of embedded functionality grows rapidly, leading to distributed solutions. Distributed embedded systems can, for instance, be found in time- and safety-critical domains, such as avionics and automotive. Suitable approaches to develop these systems are virtual prototyping and model-based development. Virtual prototyping enables early testing and evaluation of systems in realistic simulated environments; model-based development is applied to design, implement, and deploy the embedded functionality itself.

In this paper, we briefly survey our modular simulator framework Feral, which provides a generic solution to virtual prototyping by rapid coupling of diverse simulators, including simulators for model-based specifications. We demonstrate the adaptation of Feral by incorporating several simulators, in particular, existing simulators of Simulink and SDL models, ns-3, and our newly developed simulators for CAN and FlexRay, which are current communication technologies in the automotive domain. We then apply Feral to an adaptive cruise control system, evaluating different design alternatives in a real world scenario. In particular, we evaluate early design decisions by substantiating them with facts, e.g., regarding the performance of a particular system architecture or communication technology.

KEY WORDS
Virtual prototyping, network simulation, simulation tools and languages.

1 Introduction
Nowadays, embedded systems control numerous high-impact functions, e.g., in automotive, avionic, and medical devices. In the past, these embedded systems and their software functions were developed almost independently of each other. Today, there is an evident trend towards high-integration, which is driven by the need to reduce space, energy consumption, and cost of modern systems. Examples for high-integration activities are the integration of functions with mixed criticality in a segregated, i.e., isolated manner on common hardware for automotive domains, or modern avionic execution environments like the IMA platform [1].

High-integration must be included in architecture decisions, and the resulting system performance must be evaluated early to assess the validity of deployment concepts. Testing of these systems also becomes more challenging. This requires development and validation approaches that assess system level artifacts on different abstraction levels in a holistic manner, which includes not only functional behavior, but also communication networks and the system environment. While specialized simulators for individual aspects already exist, holistic evaluations of highly integrated systems require an integrated simulation environment that enables testing in a realistic context. Therefore, there is a need for frameworks that enable efficient and rapid coupling of specialized simulators. When coupling simulators, it is necessary to harmonize their Models Of Computation and Communication (MOCCs), and to enable coupling of different abstraction levels to support larger scenarios with manageable complexity and simulation times. The resulting simulation is a virtual prototype that enables early and realistic evaluations of system properties and performance.

In this paper, we present the extensible Framework for the Efficient simulator coupling on Requirements and Architecture Level (Feral) for rapid coupling of simulators, and its application to virtual prototyping. We introduce simulators that are integrated into Feral and model different aspects of distributed embedded systems, namely behavioral aspects (Simulink, SDL) and the characteristics of the communication medium (CAN, FlexRay, ns-3).

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 outlines the main features of our simulator framework Feral. Section 4 describes the adaptation of Feral by incorporating simulators for modeling functionality and communication behavior. Section 5 focuses on the exchangeability of simulation components. Section 6 shows the application of Feral in a realistic scenario from the automotive domain. Section 7 summarizes results and discusses lessons learned. Finally, Sect. 8 draws conclusions and lays out future work.
2 Related Work

There exist several approaches for coupling individual simulators or to extend existing simulators by new features. One such approach which couples ITU-T's Specification and Description Language (SDL) [2] to the network simulator ns-3 [3] is described in [4]. There, the authors integrate C-code generated from SDL specifications as modules into ns-3. This approach is extended in [5] with the focus on the modeling of configurations. A drawback of this approach is that it currently only couples SDL with ns-3. The authors assume that their tool can be adapted to support other simulation libraries, but without providing an implementation.

The work presented in [6] illustrates the application of simulator coupling to the networking domain by coupling simulators for link layer protocols and network layers. A drawback of this solution is that it only provides a single vertical coupling between two simulators in the networking domain.

In [7], the authors apply the simulator coupling approach to the automotive domain for simulating the impacts of Car-to-X systems. This approach requires coupling of a network simulator, a road traffic simulator and a radio channel simulator, but is also limited to a special application domain.

In [8], a model-based approach for the development of X-by-Wire applications is suggested, using Matlab Simulink [9]. The first step is the development of a functional model of the application ignoring communication aspects such as delays. The signal paths are replaced in the next step by specific blocks simulating the behavior of the selected communication system such as CAN or FlexRay. Two products in this segment for Simulink are xCom by DECOMSYS and the Vehicle Network Toolbox by MathWorks. In [10], Simulink is used to develop a model of the CAN bus allowing performance simulations regarding the communication. A major drawback of both approaches is the restriction to one modeling technique, which makes them only applicable if all functional and environmental models are developed in Simulink. Also, simulation of functional components with different abstraction levels (some realized as Simulink models, others already implemented) in a single scenario is not covered by these approaches. Since the usage of communication-specific blocks enforces the modification of the models, the evaluation of different communication technologies is very time-consuming and error-prone.

3 Outline of Feral

Feral (Framework for the Efficient simulator coupling on Requirements and Architecture Level) is a Java-based simulator framework with the objective to evaluate functional and non-functional requirements of networked systems. Due to its modular structure and the loose coupling of simulation components, Feral supports the rapid development of holistic simulation environments, which is, for instance, needed by virtual prototyping to evaluate design alternatives (e.g., different communication technologies) in early development phases. Because of the extensibility with specialized, domain-specific, and tailored simulators like Matlab Simulink, simulation systems built with Feral can consist of components with different abstraction levels. Thereby, Feral enables continuous testing and integration during the whole development process and thus reduces development costs and risks.

Each simulation component of a simulation system built with Feral is realized by a specialized simulator executing a behavior model. Simulation components have unique names and interact via interfaces consisting of named ports, which are connected by unidirectional links. To integrate a simulator into Feral, the simulator must implement the SimulationControl interface, allowing Feral to control its execution. Therefore, the control interface offers methods to initialize and execute the corresponding simulator respectively the concrete simulation component by the methods init, prefire, fire and postfire. The major challenge of combining different simulators is the harmonization of their MOCCs (Models Of Computation and Communication), controlling their execution and defining their communication patterns. To solve this problem, we have adopted the Ptolemy [11] concept by incorporating hierarchical directors. A director controls the execution of simulation components and the forwarding of messages on links with respect to their MOCCs, thereby supporting time-triggered as well as event-triggered semantics.

The meta model of Feral's simulation systems is shown in Figure 1. Each simulation consists of one root director, controlling the execution of at least one simulation component. Simulation components are either controlling components (directors), behavioral components (Communication-based Simulation Components and Functional Simulation Components; CSCs and FSCs), or converting simulation components (bridges and gateways). By designing directors as simulation components, arbitrary nesting of directors is supported. Behavioral components define the characteristics of communication technologies (CSCs) or application functionalities (FSCs). We elaborate on them in Sect. 4. Converting simulation components state the glue for the coupling of behavioral components and take the key role for their exchangeability. They interconnect either FSCs and CSCs (bridges), thus enabling a
4 Adaptation of Feral

To obtain a simulation system, the framework Feral is adapted in three steps. First, existing simulators (e.g., Simulink, ns-3) to be used in the simulation system are coupled to Feral. This is achieved by implementing the Feral control interface `SimulationControl`, which needs to be done only once per simulator. Second, for each coupled simulator and type of simulation component, the Feral component-specific interface for the exchange of events among simulation components (e.g., transmission and reception of messages, application specific events) is refined and implemented. Third, simulation components are instantiated by choosing a simulator already coupled to Feral and by specifying and inserting a behavior model (e.g., Simulink model or SDL specification) or communication technology (e.g., CAN or FlexRay bus) to be simulated.

4.1 Functional Simulation Components

In this section, we outline the integration of simulators supporting the model-based specification of single functions or nodes of a distributed system. Besides the specification of functions with Matlab Simulink or SDL for fast prototyping, the direct implementation of simulation components as well as the joint use of all techniques is also supported. Therefore, developers can choose the tools suitable for the task and abstraction level of the developed components.

4.1.1 Native Simulation Components

A fast way to get simulation components for prototypes of system functionalities is to implement them directly in Java, together with support for simulation control. The implementation of a component-specific event exchange is optional. If a component skips adding specific event exchange support, it falls back to the generic event exchange mechanism Feral offers and can use the concept of bridges (see Sect. 5.2) to interact with CSCs. Thus, we obtain native simulation components. In this particular case, the three steps of adapting the framework Feral are done together, as there is no separation between simulator and behavior model. Prototypes of system functionalities can, for instance, implement the externally visible interface behavior, but exhibit a simplified internal behavior. This approach helps to prevent interface and interaction incompatibilities during early development stages. Furthermore, by continuously refining native simulation components, testing and integration is supported during all development stages.

4.1.2 Integration of Simulink

MathWorks’ Matlab Simulink [9] is a well-known and widely used environment for model-based development and simulation of dynamic and embedded systems [8]. Simulink supports the modeling of time-discrete control algorithms and dynamic systems. Together with Stateflow, the development of state-based systems is also supported. Furthermore, Simulink enables the specification and development of embedded system functionality on different abstraction levels, ranging from early and incomplete prototypes to full algorithms. The behavior of a Simulink system is modeled by hierarchical block diagrams based on predefined block libraries.

In conjunction with Feral, engineers can evaluate and debug a single Simulink model in a realistic context. For this purpose, we have developed a director allowing the Simulink IDE to control the whole simulation run, thereby enabling the use of all features of the Simulink IDE.

We have also developed and integrated a Simulink simulator into Feral, capable of executing Simulink models as FSCs in a simulation system. Each incorporated Simulink system becomes an independent simulation component. Our integration approach is based on C-code gen-
erated by MathWorks’ Embedded Coder, which is executed under the control of our simulator. To couple Feral and the generated C-code, the Java Native Interface (JNI) has been utilized. Simulink simulation components are executed under the control of a time-triggered director, thereby directly realizing the time-triggered execution and communication semantics of Simulink models. During their execution, values of the input and output ports of the Simulink model are mapped to suitable data types of Feral and the corresponding ports of the simulation component, and vice versa.

4.1.3 Integration of SDL

ITU-T’s Specification and Description Language (SDL) [2] is a formal description and modeling technique for reactive and distributed real-time systems. The behavior of a system is modeled by asynchronously communicating extended finite state machines. Systems are usually specified with a graphical representation of SDL called SDL-GR that is also supported by several tools such as IBM’s Rational SDL Suite.

For the integration of SDL into Feral, we use the Rational SDL Suite to specify the FSC’s behavior, which is afterwards converted into the textual representation SDL-PR and, finally, into C++-code using the code transpiler ConTraST [12]. By compiling and linking the system with the SDL Runtime Environment (SdlRE) and the SDL Environment Framework (SEnF) [13], an executable system is generated. SdlRE implements the SDL virtual machine controlling the execution of the SDL system according to the SDL semantics. Triggering the execution of an SDL-based FSC causes SdlRE to fire executable SDL transitions. For SDL, we support both time- and event-triggered MOCCs. SEnF, on the other hand, provides platform-specific hardware drivers to access the peripherals of the corresponding platform. To interface SDL specification with Feral, special virtual drivers for CAN, FlexRay, and several ns-3 communication technologies have been integrated into SEnF. On implementation level, the interface between the Java implementation of Feral and the C++ implementation of the SDL system is realized by JNI.

4.2 Communication-based Simulation Components

To evaluate design alternatives, to predict the runtime behavior, and to perform integration tests of distributed systems, realistic simulators for current communication technologies are necessary. In this section, we present the integration of communication technology simulators with Feral, and selected implementation details.

4.2.1 Integration of CAN and FlexRay Simulators

We have developed realistic simulators for CAN and FlexRay, two widely used fieldbus technologies in the automotive and automation domain. In this section, we describe the integration of these tailored and domain-specific simulators, allowing architects to predict non-functional requirements, such as communication delays, depending on the used communication technology.

The CAN bus is an event-triggered communication bus developed by Robert Bosch GmbH since 1983 with data rates of up to 1 MBit/s. Especially in the automotive domain, CAN is widely used for various critical and comfort-related functions [14]. CAN is based on the CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) mechanism. Each frame transmission starts with the transmission of a CAN message identifier, representing the priority of the frame. If several nodes start a transmission at the same time, the node transmitting the frame with the highest priority wins the arbitration and can transmit its frame. The transmissions of other nodes are delayed until the start of the next arbitration.

FlexRay is a deterministic, time-triggered fieldbus with two independent physical channels with up to 10 MBit/s, especially devised for the needs of the automotive industry [15, 16]. FlexRay uses TDMA (Time Division Multiple Access) in the static segment and FTDMA (Flexible TDMA) in the optional dynamic segment as medium access control mechanism. The static segment is divided into static slots of equal length and enables deterministic guarantees concerning frame latency and jitter. The FTDMA mechanism of the dynamic segment allows priority-based transmissions with dynamic bandwidth acquisition. Our simulator supports versions 2.1a and 3.0.1 of the FlexRay protocol specification.

Both simulators are realized in Java. Figure 3 shows an excerpt of the logical connection structure of a simulation system with a CAN bus as CSC and two FSCs as network nodes. For every node connected to the CAN bus, a stub is instantiated – a so-called Protocol Control Unit (PCU) –, implementing the protocol-specific behavior according to the protocol standard. CAN messages between the FSCs and its stub are exchanged via unidirectional links. The PCUs can be implemented to reflect the behavior of concrete bus connection circuits used to connect an embedded system with the bus, based on the data sheets of the vendor (e.g., vendor specific features or processing delays). The Medium Control Unit (MCU) implements node-spanning protocol characteristics, such as the arbitration mechanism of CAN. Since the architecture of the FlexRay simulator is very similar, we omit a detailed description.

CAN and FlexRay simulators are frame-based, i.e., resulting delays of frame transmissions are calculated according to the protocol specifications with respect to the given configuration parameters of the bus. Thereby, the impacts of the used frame format and encoding are also evaluated (e.g., CAN uses bit stuffing to retain synchronicity between sender and receiver, which influences the transmission length of the message). Physical effects, such as signal propagation, are not simulated.

The simulation of transmission errors is also supported by incorporating stochastic error models. Using
To represent this medium, a corresponding simulation system with an ns-3 simulated communication medium is instantiated on Java side. An interface to execute the simulator for a specified timespan is provided, which serves as communication counterparts for the Java endpoints on C++ side. Communication between an endpoint and its virtual application is realized via message exchange.

The NS3Scenario serves as a container for storing the PCUs and representing the simulation system for ns-3. The PCUs are connected by a channel, which simulates the actual communication medium and forms the counterpart of the NS3CommMedium. Amongst others, Feral currently supports WLAN, Ethernet and a switched Ethernet variant. The type of communication used for a concrete medium is determined by its communication mode. Currently, we support the protocols TCP and UDP as communication modes and communication without using a protocol stack, by sending broadcasts directly via net device.

5 Exchangeability of CSCs and FSCs

In simulator frameworks, a crucial requirement for the early and continuous evaluation of design alternatives is the support of exchangeability of simulation components. Although Feral supports development-accompanying evaluations by integrating various simulators running models with different abstraction levels, further extensions of Feral are necessary to evaluate design alternatives, i.e., to evaluate different concrete simulation systems that are generated from the same abstract simulation system. To enable a rapid exchange of CSCs and FSCs, Feral has been extended with communication-specific configurations, bridges, and gateways. The exchangeability of CSCs is basically achieved by providing an abstract factory pattern with a common interface to create the desired CSCs. By introducing bridges and gateways encapsulating communication-specific knowledge, FSCs can be specified without knowledge of the connected communication medium, and communication technologies can be exchanged without requiring changes to the FSCs. Thereby, Feral becomes an attractive tool to assess the different communication qualities such as message loss and delays.

5.1 CSC-Specific Configurations

The configuration of communication technologies is based on a common configuration structure, providing
technology-specific configurations for all CSCs, which may be instantiated in the abstract simulation system to obtain a concrete simulation system. The concrete communication technology is selected by providing the respective factory in the simulation system description, which can be given in Java or Java Script.

All configuration files are stored in a common config folder, containing sub-folders named according to the unique name of each CSC in the abstract simulation system. This folder contains again sub-folders for all supported communication technologies, which may be instantiated for this CSC. These sub-folders contain for each FSC a node-specific XML configuration setting options of the PCU representing the node within the respective CSC (such as buffer configurations for FlexRay) and a configuration file to specify medium-related properties, e.g., the transmission rate of the CAN bus. The folder components has sub-folders for every FSC connected to the CSC, containing the DBL description for the bridges of this FSC (see Sect. 5.2). Figure 5 shows an example configuration structure for a simulation system consisting of a communication medium ControlBus with two connected FSCs Node1 and Node2, providing configurations for CAN and FlexRay.

5.2 Interfacing FSCs and CSCs

To enable the evaluation of different communication technologies, the exchange of different CSCs must be as simple and expeditious as possible. This can only be achieved, if – besides the existence of a common configuration structure for all desired communication technologies – no modifications or redefinitions of the abstract simulation system and the FSCs are necessary. Therefore, bridges and gateways are introduced to separate communication-related aspects and the models of FSCs. We are focusing on FSCs based on native implementations and Simulink models, because they are often used as prototypes, black box components, or placeholders in early development phases, when design alternatives such as different communication technologies are typically evaluated.

Bridges connect FSCs with CSCs by encapsulating communication technology-specific knowledge (e.g., CAN message priorities or FlexRay buffers) and functionality. To exchange communication technologies, behavior configurations of the bridges regarding the desired technologies have to be provided. Since this solution avoids the injection of communication-specific knowledge into the FCSs, there is no need to adjust the Simulink models or implementations of native simulation components if the communication technology is changed. Furthermore, bridges provide a common interface to the CSCs and, therefore, ensure that components specified on different abstraction levels may be linked to each other. Message transfer between bridges and CSCs is encapsulated by specific events, such as events to send and receive a message from the communication medium.

The behavior of bridges is specified in a novel domain-specific language called DBL (Domain Bridge Language). A benefit of Xtext is the automatically generated Eclipse editor supporting the developer during the specification of a DBL description.

We distinguish between two different kinds of bridges: Input2ComBridges and Com2OutputBridges. An Input2ComBridge connects the output ports of an FSC to the input ports of a CSC. The behavior is described in DBL and specifies triggers defining the conditions for creating events, e.g., periodic triggers, sporadic triggers, custom application-specific triggers, and logical combinations of these. Actions released by a trigger define which events should be created, e.g., creating events for message transmissions with a payload consisting of the concatenated values of the input ports.

| Dist = BusConfig(  |
| Generic {         |
|   ConcatPayload("RadarObstacleDistance", |
|   "RadarObstacleSpeed"); }         |
| CAN { BaseID → 20; } |
| FlexRay { MessageBuffer→0; MessageBuffer→1 } |
| NS3 { Protocol→UDP, Broadcast→true, |
|       Port→200; }   |
| CAN, NS3 {        |
| on OnChange("RadarAlarm") do SendMessage(Dist) exclusi|
| exclusive block for (0:20000000); |
| on PeriodicTrigger(interval→0:100000000) do |
| SendMessage(Dist) exclusive; |
| FlexRay: on PeriodicTrigger(interval→0:25000000) do |
| SendMessage(Dist); |

Listing 1. DBL description of a radar distance sensor

Listing 1 shows the DBL description for a radar distance sensor measuring the distance to an obstacle and its speed. Using CAN or a switched Ethernet topology simulated with ns-3, its values are transmitted periodically every 100 ms (line 10). The sensor has also an alarm port allowing to immediately issue a message if a minimal safety distance has been deceased (line 9). A periodical transmission will be skipped, if scheduled within an interval of 20 ms after an alarm has triggered a transmission. A new alarm trigger will also be ignored within this period.
of time. With CAN, identifier 20 is used for the resulting message. Using FlexRay, slots 3 and 88 are exclusively reserved for radar messages. These slots are associated with the internal message buffers 0 and 1 (see [16] for more details about the internal buffer structure) and have a maximal distance of approximately 2.5 ms in our example configuration. Therefore, we are providing new sensor values for every reserved slot, independent from the alarm trigger (line 13). For Ethernet, UDP broadcast transmissions with unique ports are used, where the radar sensor messages are associated with port 200 (line 6). The payload for each communication technology consists of the concatenated values of two output ports of the radar sensor simulation component (“RadarObstacleDistance” and “RadarObstacleSpeed”, line 3) and is created by the predefined concatenation payload generator.

A Com2OutputBridge is the logical counterpart to an Input2ComBridge. It filters and decomposes the received data stream and assigns the extracted values to the output ports of the bridge according to optional extraction rules, which are specified in the DBL description. These output ports are connected to the input ports of the associated FSC. Listing 2 shows a DBL description of a Com2OutputBridge handling incoming brake requests. In this example (lines 1-4), values from messages with identifier 10 (CAN), messages received on port 100 (ns-3), or messages received by message buffer 2 and 3 (FlexRay), are handled by the consume statement in line 8. Additionally, line 6 specifies that the data from the payload should be extracted as double value taking little endian encoding into account and forwarded to the input port of the FSC with the name InitiateBrake.

```
IncomingBrakeMessage = MessageFilter(
  FlexRay { MessageBuffer→2; MessageBuffer→3 } 
  CAN { BaseId→10, RemoteFrame→false; } 
  NS3 { Protocol→UDP, Port→100; } );
extractBrakeData = Consume(extract 0:8 
  (LittleEndian, NoAlign) as Double to "InitiateBrake")

Listing 2. DBL description for receiving a brake signal
```

Besides the predefined triggers, filters, actions, and payload generators, DBL also supports the instantiation of application-specific triggers, filters, actions, and payload generator classes, which must implement the corresponding Java interfaces. This allows the application and adaptation of the bridge approach in very specific scenarios.

The concept of bridges can also be extended to connect several CSCs. This is achieved by introducing gateways forwarding messages between communication media. By providing such interfaces between different communication technologies, CSCs can be replaced and different combinations of communication technologies can be evaluated. To realize this function, a set of rules has to be provided mapping incoming messages from one CSC to a number of outgoing messages forwarded to other connected CSCs. For example, a gateway connecting a CAN and a FlexRay bus may forward received CAN messages with certain message IDs to the FlexRay bus by sending them in the first slots of the dynamic segment. Also messages from the FlexRay bus received in a certain slot may be retransferred on the CAN bus with a specific ID. These mapping rules can be described with DBL extended by additional filters for selecting CSCs by their unique names.

### 6 Application of Feral

In this section, we apply Feral to a realistic example of an Adaptive Cruise Control system (ACC) from the automotive domain and evaluate different design alternatives regarding the used communication technology. ACC is an enhanced cruise control system focused on retaining a reference speed against disturbance variables such as the current gradient or aerodynamic resistance. In addition to a simple cruise control, a radar sensor is used to detect the distance to obstacles in front of the car. Depending on the speed of and distance to the obstacles, the reference speed is adjusted to keep a minimal safety distance and initiate an emergency braking if necessary.

We introduce the example by presenting the abstract simulation system in Sect. 6.1. Afterwards, we instantiate the abstract system with concrete simulators and models resulting in a set of concrete simulation systems (Sect. 6.2) utilizing our concepts of exchangeability for CSCs, and finally present simulation results (Sect. 6.3).

#### 6.1 Abstract Simulation System

Figure 6 shows the abstract simulation system of the ACC scenario. The algorithm controlling the speed of the car is a Proportional-Integral-Derivative (PID) controller hosted by ACController. Speed and distance of other cars or obstacles ahead are measured by RadarSensor. The physical model providing the actual speed of the car including the engine and engine controller is represented by CarModel. SpeedInput provides the target speed via the EngineCon-
Comparing the gradient input to the controlled system, the actual gradient delivered by the radar sensor acts as a disturbance variable for the controlled system. Messages sent by the RadarSensor, SpeedInput, and GradientInput can, for instance, be recorded messages to enable the replay of specific scenarios or data from field tests. Additionally, a configurable traffic generator BusTrafficGen simulates the communication characteristics of further components. To illustrate the coupling of several CSCs, we extend our system with a further medium (Car2Car), simulating communication between cars by propagating traffic warnings from car to car. These traffic warnings are forwarded by gateway GW1 to ECB.

Traffic warnings are modeled as sporadic messages, whereas actual speed, engine control values, and target speed are periodic messages with an interval of 20 ms (actual speed and engine control) and 100 ms (target speed), respectively. Distance and speed values of an obstacle are delivered by the radar sensor with an interval of 100 ms. This interval is reduced to 20 ms if an obstacle is within the minimal safety distance. The brake signal is a sporadic signal created by the ACController, if the minimal safety distance is violated.

6.2 Concrete Simulation Systems

By selecting specific models, simulators, and communication technologies, a set of concrete simulation systems is created from the abstract simulation system. The concrete simulators are summarized in Table 1. Except for the ECB, every FSC and CSC is simulated with a single simulator and behavior model. To analyze the impacts of different communication technologies regarding transmissions delays, the ECB is instantiated with three different communication technologies: FlexRay, CAN, and switched Ethernet. The CAN bus runs with 1 MBit/s, FlexRay with 10 MBit/s with exclusive static slot reservations, and switched Ethernet with 10 MBit/s, too. For Ethernet, the messages are transmitted via UDP broadcasts and are delivered to unique ports for each message type.

Additional properties of each message type are summarized in Table 2. Note that FlexRay applies a TDMA scheme, where some safety-critical messages (actual speed, engine control, radar, and brake messages) obtain two transmission slots per cycle to reduce jitter and reaction delays.

To simplify the exchange of ECB’s communication technology, most FSCs are decoupled by bridges (specified by a DBL description) from the CSC. An excerpt of DBL descriptions of this scenario is provided by Listing 1 and 2. The listings formulate the communication behavior regarding the three communication technologies for the generation of radar messages and the reception of brake messages. Similar to the description of bridges, the behavior of the gateway has been specified in a DBL description to enable a rapid exchange of the communication technology.

6.3 Simulation Results

Figure 7 shows the results of a simulation system run with CAN as ECB. The dotted line in the upper graph shows the target speed, specified by a driver, whereas the solid line shows the actual speed adjusted by the ACController. In the lower graph, an altitude profile of the simulated track is depicted as dashed line. The resulting gradient acts as disturbance variable in the controlled system. The other line shows the distance to the next car ahead.

The controlled car starts with a target speed of 50 km/h, which is increased to 140 km/h after 3 sec simula-

---

**Table 1. Simulators in the concrete simulation system**

<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>RadarSensor</td>
<td>FSC</td>
<td>Native Java</td>
</tr>
<tr>
<td>GradientInput</td>
<td>FSC</td>
<td>Native Java</td>
</tr>
<tr>
<td>SpeedInput</td>
<td>FSC</td>
<td>Native Java</td>
</tr>
<tr>
<td>BusTrafficGen</td>
<td>FSC</td>
<td>Native Java</td>
</tr>
<tr>
<td>OtherCar</td>
<td>FSC</td>
<td>Native Java</td>
</tr>
<tr>
<td>CarModel</td>
<td>FSC</td>
<td>Simulink</td>
</tr>
<tr>
<td>ACController</td>
<td>FSC</td>
<td>SDL</td>
</tr>
<tr>
<td>EngineControlBus</td>
<td>CSC</td>
<td>CAN, FlexRay, Ethernet</td>
</tr>
<tr>
<td>Car2Car</td>
<td>CSC</td>
<td>IEEE 802.11 (WLAN)</td>
</tr>
</tbody>
</table>

**Table 2. Message types in the ACC system**

<table>
<thead>
<tr>
<th>Message Type</th>
<th>CAN ID</th>
<th>FlexRay Slots</th>
<th>Ethernet Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radar Signal</td>
<td>20</td>
<td>3, 88</td>
<td>200</td>
</tr>
<tr>
<td>Actual Speed</td>
<td>70</td>
<td>1, 86</td>
<td>700</td>
</tr>
<tr>
<td>Target Speed</td>
<td>45</td>
<td>2</td>
<td>450</td>
</tr>
<tr>
<td>Engine Control</td>
<td>110</td>
<td>5, 90</td>
<td>1100</td>
</tr>
<tr>
<td>Brake Signal</td>
<td>10</td>
<td>6, 91</td>
<td>100</td>
</tr>
<tr>
<td>Traffic Warning</td>
<td>188</td>
<td>67</td>
<td>1880</td>
</tr>
</tbody>
</table>

**Figure 7. Speed adaptation**
transmission time. The car ahead drives with a constant speed of 100 km/h and starts braking after 54 sec. After 20 sec, the distance to the car ahead becomes smaller than twice the minimal safety distance, and the controlled car automatically reduces its speed, ignoring the given target speed. At the minimal allowed safety distance (approx. 50 meters for 100 km/h), the controlled car has adapted to the speed of the car ahead. When the car ahead brakes, the ACC controller automatically enforces braking until the car stops to prevent a crash.

Figure 8 shows the observed maximum, minimum, and average transmission delays for the ECB using CAN, FlexRay, and switched Ethernet for ACC relevant messages. The traffic generator adds additional transmissions increasing the total medium usage to 39.7% (CAN), 9.8% (FlexRay) and 23.9% (Ethernet). Since CAN and Ethernet allow an immediate transmission of frames, the average transmission delay is smaller than with FlexRay. However, both protocols can not guarantee a maximal transmission delay. For instance, another traffic configuration with 79% bus usage on CAN (not shown in the figure) yielded a maximum transmission delay for the brake messages of 5.5 ms. The average performance gain of Ethernet compared to CAN is not very high, although the data rate is higher by factor ten. Yet, the maximum delay is higher with Ethernet. A reason is the overhead added by the Ethernet protocol in relation to the small payload that is only 2-8 bytes.

The FlexRay schedule contains exclusive static slot reservations for every message and a 5 ms communication cycle (static and dynamic segment have approx. a length of 2.5 ms each). Important messages such as current speed, radar distance, engine control, and brake commands have two slot reservations per cycle (with a slot interval of approx. 2.5 ms). The maximal delay is given by the maximal duration between consecutive reserved slots, whereas the average case delay is 50% of the interval of the reserved slots. A major benefit of FlexRay is that the maximal delay is guaranteed upper bound and is independent of the medium load. The delay of the important brake signal is minimized by design. Since a brake signal is eventually created as reaction to an incoming radar signal, the reserved slot for a brake signal is two slots after the slot of a radar signal. This allows an immediate reaction to new radar sensor values and results in very low delays and jitter.

Transmission delays regarding traffic warnings consist of the delay for transmitting the message from car to car over IEEE 802.11, a gateway processing delay, and the final transmission over the ECB. Thus, delays of traffic warnings are in general higher.

In summary, all evaluated communication technologies fulfill the requirements of this scenario, but only FlexRay can guarantee maximal transmission delays independent of the load of the medium. In all concrete simulation systems, the control performance was adequate.

7 Discussions and Results

As our realistic simulation scenario shows, Feral provides a holistic platform for virtual prototyping. In particular, Feral supports the use of simulation components on different abstraction levels. In the ACC scenario, we have combined prototype components written in Java, design models specified in SDL, and environment components generated from a Simulink model into one simulation system. Furthermore, several simulators controlled by the same director have been used together to run the simulation system.

Flexibility of Feral is further enhanced by our concepts of bridges and gateways, which we have used in our simulation system to hide the specifics of different CSCs. With bridges and gateways, it has been straightforward to switch from an abstract simulation system to a concrete simulation system by exchanging communication technologies, and to compare their impact on the overall system behavior. We note that running simulations with CSCs requires some domain knowledge. In particular, this is needed to configure the communication systems, to derive valid schedules for messages to be transmitted, for instance, by assigning priorities or time slots, and to configure bridges and gateways using DBL (Domain Bridge Language).

A focus of the work reported in this paper has been the coupling of several simulators, e. g., Simulink, ns-3, and simulators for CAN and FlexRay, to Feral. This has been achieved by implementing, for each simulator, the Feral simulation control and component-specific interfaces. The effort to integrate simulators into Feral varies and depends on the architecture, offered interfaces, the used MOCC, and the implementation language of the respective simulator. Besides the coupling of existing simulators, we have introduced specialized simulation components called bridges, and DBL for their high-level configuration. We have applied bridges and gateways to incorporate CSCs. Since DBL is based on Xtext and the bridges and gateways are designed to provide several extension points, it is straightforward to extend DBL in order to incorporate further communication technologies and their technology-specific parameters. In addition, the usage of bridges in conjunction with new types of FSCs is possible, since the bridge con-
cept is built upon the basic input and output ports all FSCs must already provide to interact with Feral.

In summary, the design of Feral, our bridges, gateways, and configuration concept enable the fast and systematic integration of functional and communication-based simulators. Furthermore, the evaluation of scenarios using different communication technologies and components of different abstraction levels is supported. This enables the exploration of the impact of design decisions and early integration tests.

8 Conclusion

In this paper, we have presented our generic approach for simulator coupling, enabling early testing and virtual prototyping, which is especially important in the domain of embedded systems. System components may be available on different abstraction levels, and can be simulated together. We have applied our approach to couple different kinds of existing and self-developed simulators, and to instantiate and simulate an adaptive cruise control system, where we have switched between communication technologies using the concept of bridges and gateways.

In our future work, we will integrate platform simulators to create and evaluate virtual prototypes of control units, such as ECUs in the automotive domain. For that purpose, we are currently evaluating Open Virtual Platform [18]. Also, by extending the component-specific interface, we are planning to make more features of ns-3 usable in the framework, e.g., further communication media and routing protocols. Last but not least, we are working to obtain a tighter integration between real-world hardware and simulated components, using hardware-in-the-loop solutions to integrate sensor nodes into the simulator framework.

References


