Hindawi Publishing Corporation
Journal of Control Science and Engineering
Volume 2013, Article ID 678016, 15 pages
http://dx.doi.org/10.1155/2013/678016

Research Article

Model-Based Control Design and Integration of Cyberphysical Systems: An Adaptive Cruise Control Case Study

Emeka Eyisi, Zhenkai Zhang, Xenofon Koutsoukos, Joseph Porter, Gabor Karsai, and Janos Sztipanovits

Institute for Software Integrated Systems (ISIS), Vanderbilt University, Nashville, TN 37212, USA

Correspondence should be addressed to Emeka Eyisi; emeka.eyisi@vanderbilt.edu

Received 7 September 2012; Accepted 18 December 2012

Academic Editor: Sabri Cetinkunt

The systematic design of automotive control applications is a challenging problem due to lack of understanding of the complex and tight interactions that often manifest during the integration of components from the control design phase with the components from software generation and deployment on actual platform/network. In order to address this challenge, we present a systematic methodology and a toolchain using well-defined models to integrate components from various design phases with specific emphasis on restricting the complex interactions that manifest during integration such as timing, deployment, and quantization. We present an experimental platform for the evaluation and testing of the design process. The approach is applied to the development of an adaptive cruise control, and we present experimental results that demonstrate the efficacy of the approach.

1. Introduction

Cyberphysical systems (CPS) represent a class of complex systems characterized by the tight interactions between the physical dynamics, computational dynamics, and communication networks. Automotive systems are classical examples of CPS and have recently been gaining increased attention due to the emerging challenges in their design. Current automotive systems employ up to 100 electronic control units (ECUs) exchanging more than 2500 signals over up to 5 different bus systems [1, 2]. These ECUs control and monitor many subsystems of a vehicle such as chassis control, vehicle stability, and engine control. The development of control software has become one of the greatest challenges in the automotive domain due to the increasing complexity of automotive systems as well as the increasing roles of control, computing, and communication [3–6].

The increased pressure to integrate as much functionality on as few ECUs as possible, the persistent effort for low production costs, and tight time-to-market constraints further complicate the development of automotive software control systems. Due to these challenges, there is a dire need for a reliable and efficient approach for control software development and integration. A major problem in the current state of art is that most issues with deployed control applications are typically discovered in the final phases of the development cycle, and at these later phases, correcting the issues is very expensive as it involves the modification of specifications, requirements, and design. Another issue is the lack of realistic experimental platforms for integrating and testing developed control software prior to deployment.

Systematic design and analysis of automotive control software early in the development cycle is very crucial. The model-driven development approach has been found to be very beneficial in addressing these issues [7]. However, the lack of a sound approach, for the integration of components from the control design phase with the components from software generation and deployment on actual platform/network, makes the model-driven approach very challenging because the tight interactions between the design phases often manifest during integration. The current state-of-the-art often resorts to ad hoc methods with the goal of "making the system work." These ad hoc methods are becoming unpractical as the complexity of the system increases.

In this paper, we present a step towards addressing the challenges in the design and integration of components from control design with components from software generation and deployment on actual platform/network. We present a systematic methodology and a toolchain to integrate control design and scheduling in the development of automotive control applications with specific emphasis on restricting the interactions that manifest during integration such as timing, deployment, and quantization. Our design process utilizes a model-based toolchain, Embedded Systems Modeling Language (ESMoL) [8]. ESMoL, designed in the Generic Modeling Environment (GME) [9], is a single multiaspect embedded software design environment, which streamlines control design with software modeling, code generation, and deployment on platform/network, filling the very important details between the design phases promoting a high-confidence software development process.

In order to evaluate our software development process and toolchain, we employ an experimental platform based on the time-triggered paradigm that enables the deployment and testing of automotive control applications. The time-triggered paradigm is used to address the complexity and composability challenges by precisely defining the interfaces between components in order to provide predictability [10]. Also, our choice of a time-triggered paradigm falls in line with the increased ongoing efforts towards the standardization of in-car communication networks, such as FlexRay and Time-Triggered Ethernet (TTEthernet or TTE), with the overall goal of guaranteeing highly reliable, deterministic, and fault-tolerant system performance [11].

The software development process and toolchain together with the experimental platform efficiently connect all the phases of development of automotive control applications, essentially going from control design using Matlab/Simulink to deployment and hardware-in-the-loop simulations. In order to demonstrate our approach, we apply the proposed process to the development of an adaptive cruise control (ACC). The adaptive cruise control (ACC) system is an active safety and driver-assistance vehicle feature that automatically controls a vehicle's longitudinal velocity in a dynamic traffic environment. ACC enables an ACC-equipped vehicle to follow a leading forward moving vehicle while maintaining a desired distance from the leading vehicle as determined by the vehicle's velocity and a specified time gap or headway. We present experimental results from the hardware-in-the-loop simulations of the designed ACC on the experimental platform.

This paper is organized as follows. The related work is presented in Section 2. In Section 3, we describe our view of CPS design and integration, and we formulate the specific problem considered in this paper. We present the proposed systematic methodology for model-based control design and integration in Section 4. Section 5 presents the system architecture for the experimental platform. Section 6 describes the control design of the adaptive cruise control. Section 7 describes the software design process for the adaptive cruise control. Section 8 presents an experimental evaluation of our proposed approach using the adaptive cruise control case study. Finally, Section 9 provides a brief discussion and concludes the paper.

2. Related Work

Model-based software development approach as well as test-beds for testing automotive control systems architecture is a very active research area. There is an increasing amount of work in this area attempting to address various cross-layer challenges [12]. In [13], an automotive testbed for electronic controller unit testing and verification was presented. The presented platform provides many advantages for testing ECUs and is complementary to our work. The authors in [14] present a software-based implementation and verification scheme for a FlexRay-based automotive network. The main focus of the paper is on verifying timing in control signals and the network and providing a basis for detecting and diagnosing network faults. In [15], the authors used a technique called Instrumentation-Based Verification (IBV) to design automated tests in order to check whether models developed in Matlab/Simulink satisfy specified requirements; while the work addresses an important issue of verifying requirements with control design, it does not go further into the actual software deployment and evaluation on a test platform.

The design of the adaptive cruise control (ACC) has been extensively studied, and there are numerous design techniques for deriving the corresponding control laws. Some of the most common approaches are sliding-mode design techniques [16, 17], optimal control techniques [18, 19], fuzzy logic [20], neural networks, and proportional derivative (PD) type control law [21, 22]. In this work, our main focus in the case study is on the design flow from the high level design of a vehicle control system such as the ACC using model-based tools such as Matlab/Simulink to the actual deployment and testing on an automotive testbed capable of mimicking real world scenarios. In [23], the authors describe a model-based approach for the modeling, design,dere and implementation of an intelligent cruise control. In contrast to their approach, our development approach provides a simpler graphical language that clearly defines the integration of the control software. In addition, we adopt the time-triggered architecture [10] which essentially ensures predictability, determinism, and guaranteed latencies, hence, allowing for a certain level of decoupling in system design. The work in [24] describes a FlexRay-based distributed networked system for automotive applications mainly focusing on the challenges in regards to the paradigm shift from the Controller-Area-Network-(CAN-) based even-triggered communication technologies to the introduction of time-triggered communication scheme while assuming an existing software environment.

Our work differs from the existing works, due to the fact that we consider an end-to-end design flow in the development of control software with well-defined components that are necessary for the efficient and reliable integration of design layers of an automotive CPS. By clearly defining these components which include timing and deployment, we restrict the possible interactions that can potentially make the overall behavior of the system unpredictable.

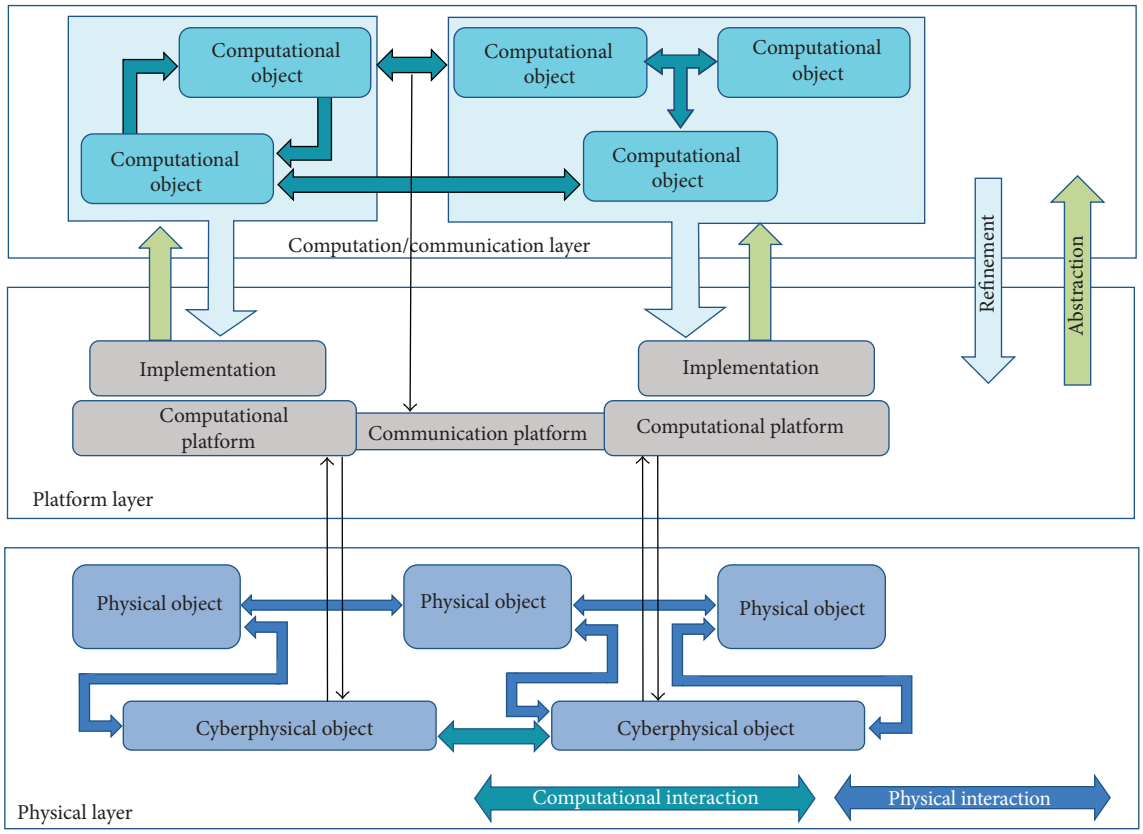


Figure 1: Design flow in CPS design layers [25].

3. Problem Formulation

Figure 1 shows the three fundamental design layers of CPS, such as an automotive vehicle [25].

(1) *The physical layer* represents physical components and their interactions. The behavior of the physical components is governed by physical laws and is typically described in continuous (physical) time using, for example, ordinary differential equations (ODEs). Physical objects are interconnected by physical components (e.g., steering wheel) or cyberphysical objects (e.g., steer by wire).

(2) *The platform layer* represents the hardware side of CPS and includes the network architecture and computation platform that interact with the physical components through sensors and actuators. While executing the software components on processors and transferring data on communication links, their abstract behavior is "translated" into physical behavior.

(3) *The computation/communication layer* represents the software components with behavior expressed in logical time. Interconnections are modeled using various Models of Computations (MoCs) [26]. Software components are connected using an input/output model with an implied notion of causality.

In view of this CPS architecture, for an automotive application, the vehicle chassis together with the engine, transmission, brakes and tires, cyberphysical objects (e.g., steer by wire), and the initial controller design comprise the physical layer. The electronic control units (ECUs) on which the control software applications are deployed, together with the communication network over which the ECUs send and receive data, comprise the platform layer.

In this work, we assume that the components of the physical layer are specified by a given physical vehicle dynamic model. Also, we assume that the platform layer is specified based on a given set of computational nodes and communication network. The main research problem we address is handling the complex interactions in the computation/communication layer of automotive CPS which manifest due to the lack of a clear and well-defined systematic integration of control design and scheduling. This problem involves the need for a "correct-by-construction" end-to-end design methodology for the modeling, designing, analysis, deployment, and testing of automotive control applications. In order for such a development process to be beneficial, it should be systematic and efficient. Additionally, such an

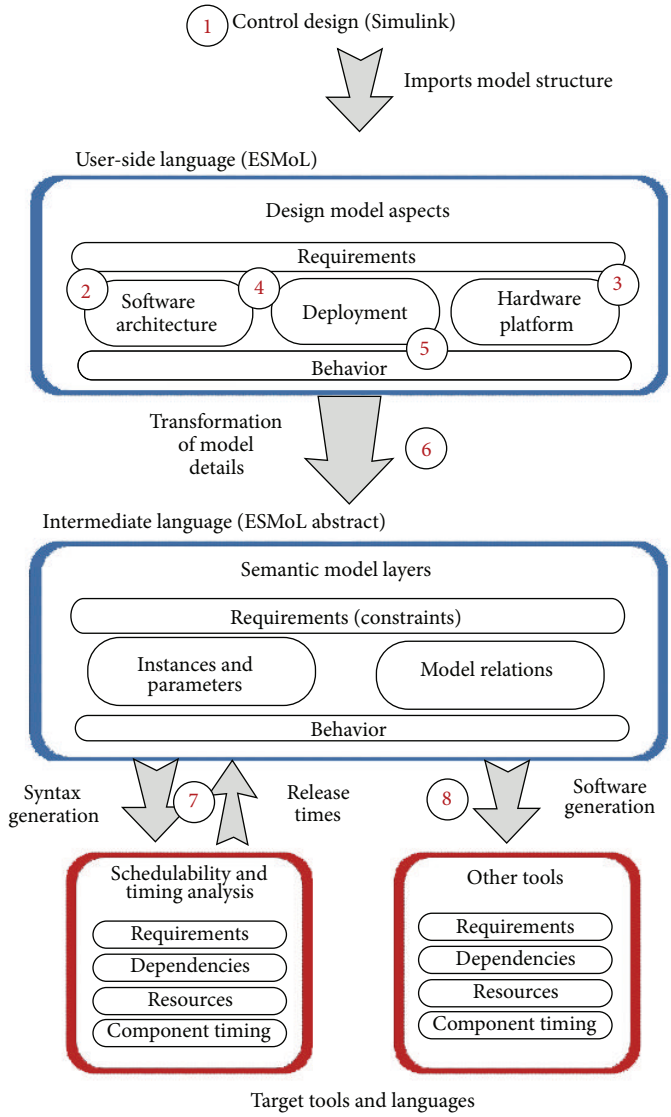


Figure 2: Automotive embedded control software design flow supported by the ESMoL language and modeling tools.

approach will require an experimental platform that is able to model realistic scenarios that can mimic real-world cases.

4. Automotive Control Software Design Process

The proposed approach uses formal models and design concepts integrated in the model-based tool chain ESMoL, to restrict component interactions by specifying attributes in various refinements of a single design model of an application. These refinements, which include component architecture, deployment, and timing/execution models, represent different aspects of the system as the process progresses towards implementation. The combined information expressed in the aspects constitutes a suitable complete model for the actual deployment and implementation of the system. We provide the design flow of the development process.

4.1. Automotive Control Software Design Flow. Figure 2 shows the design flow for the proposed software development process. The design process involves eight main steps numbered in a top-down manner, starting from the first step which involves the importation of a control design model into ESMoL to the eighth and final step of software generation, as shown in Figure 2. The design steps are described as in Figure 2.

4.1.1. Controller Design and Importation. The controller for a specific system functionality is typically designed and validated in Matlab/Simulink using simulations. Real-Time Workshop (RTW) [27], an automatic code generator in Simulink, can subsequently be used to generate the equivalent C code of the designed controller. The controller is typically designed in floating-point math, while the actual control software executes on ECUs with potentially limited number of bits and fixed-point implementation. In order to convert

controller models from floating-point to fixed-point, fixed-point Toolbox [28] is used to aid code generation.

Although Matlab/Simulink provides the environment for modeling and code generation of controllers, it lacks the adequate and important refinement for controllers that provide implementation details, such as real-time operating system (RTOS) environment, timing, hardware platform specifications, and network considerations. Thus, after the controller validation and generation of equivalent C code, in the first step of our design process, the controller's Matlab/Simulink model is automatically imported into the ESMoL environment by using the MDL2MGA tool in ESMoL. The MDL2MGA tool is a model interpreter that creates a structural replica of the Matlab/Simulink model in the ESMoL modeling environment. The replica of the Matlab/Simulink model is represented as a synchronous data flow model (SDF), and each subsystem in the replica becomes an actor in the SDF. The ESMoL model's references to the imported Simulink blocks become the functional specifications for the instances of software components in a logical SDF model. C code fragments may also be used to specify the component functionality. Component ports represent instances of data message types. These types are defined as structures with individual data fields to which Simulink data ports can be mapped. These relations describe the marshaling, demarshaling, and transfer of data between software components.

4.1.2. Logical Software Architecture. The second step in the design process, denoted as ② in Figure 2, involves the specification of the logical software architecture. The logical software architecture model describes the interconnection of component instances representing the functional blocks. The logical software architecture captures the data dependencies between software components independent of their distribution over different processors. The semantics of the logical interconnections are defined by task-local synchronous function invocations as well as message transfers between tasks based on the time-triggered communication paradigm.

4.1.3. Hardware Platform. The third step in the design process, depicted as ③ in Figure 2, involves the definition of the hardware platform on which the controller software is deployed. In this step, by specifying the attributes of the hardware platform, we clearly define the components and interactions of the platform which significantly impacts the behavior of the overall control system. ESMoL's network/platform sublanguage has several components including processing nodes and communication networks for defining the computing nodes as well as the underlying communication networks. Several specific networks for automotive systems are defined in this sublanguage, such as CAN bus and TTEthernet. In this paper, we consider TTEthernet, which is based on the time-triggered paradigm. In ESMoL, hardware platforms are defined hierarchically as hardware units with ports for interconnections. The model attributes for hardware platform also capture timing resolution, overhead parameters for data transfers, task context switching times, and scheduling policies.

4.1.4. Deployment Model. The fourth step in the design process, denoted as ④ in Figure 2, involves the definition of the deployment model. The deployment model represents the mapping of software components to processing nodes and data messages to communication ports. This model captures the assignment of component instances as periodic tasks running on a particular processor. A well-defined specification of the deployment model is important as the overall behavior of the control application depends on the efficient mapping of software components to the designated platforms. In ESMoL, a task executes on a processing node at a single periodic rate, and all components within the task execute synchronously. Message ports on component instances are assigned to hardware interface ports in the model to define the media through which messages are transferred.

4.1.5. Timing Model. The fifth step, denoted as ⑤ in Figure 2, involves the specification of the timing behavior of the system. The timing model allows a designer to specify component execution constraints involving the timing behavior of the component. The specification of the timing model is very important in order to ensure the predictability of the overall system behavior. In ESMoL, the timing model of a control application is established by attaching timing parameter blocks to components and messages. There are three types of timing parameter blocks in ESMoL to represent three different execution modes. Time-triggered execution information (*TTExecInfo*) is used to specify the timing for a task or a message transfer that executes based on a synchronized time base, such as in time-triggered distributed system. If the synchronized time base is not available or used, an event-triggered system needs to be specified. In this case, asynchronous periodic execution information (*AsyncPeriodicExecInfo*) is used for periodic execution, while sporadic execution information (*SporadicExecInfo*) is used for aperiodic execution with a minimum period. The model also indicates which components and messages that will be scheduled independently, and those that should be grouped into a single task or message object. In the case of processor-local message transfers, transfer time is neglected as reads and writes occur in locally shared memory.

4.1.6. Model Transformation. In order to integrate analysis tools and other code generators into ESMoL, rather than directly attaching translators directly to the user language, ESMoL defines a simpler abstract intermediate language whose elements are similar to those of the user language. The sixth step in the design process, depicted as ⑥ in Figure 2, involves this model transformation. An ESMoL interpreter called Stage 1 is used to perform the transformation from the originally defined ESMoL model into an abstract intermediate language that contains explicit relation objects that represent relationships implied by structures in ESMoL. This translation is similar to the way a compiler translates concrete syntax first to an abstract syntax tree and then to intermediate semantic representations suitable for optimization. Stage 1 is implemented using the Universal Data Model (UDM) navigation application interface [29]. The ESMoL_Abstract

target model is a flattened ESMoL model and the source for the transformations for further analysis and software component generations.

4.1.7. Network/Task Scheduling. This step in the design process, depicted as ⑦ in Figure 2, involves network and task scheduling. A scheduler provided by TTTech [30] is used for network scheduling. This scheduler requires a configuration script of the network/hardware platform in order to perform analysis. An ESMoL_Abstract interpreter called Stage 2 is integrated into the ESMoL's abstract intermediate language to generate the TTEthernet configuration script for network scheduling. This interpreter takes the parameters specified in the TTEthernet components of the network/platform model and combines them with message specifications generated for interprocessor message transfers, which can be deduced from the software architecture model and the deployment model. The desired offset fields of the TT messages are obtained from the timing model.

For the task scheduling, we use the bottom-level-based heuristic scheduling algorithm [31]. The algorithm establishes the critical path of the task graph, which needs at least the execution time of any other path in the task graph. In order to distinguish the tasks on the critical path, the notion of bottom-level of a task is used, which is the length of the longest path starting with this task. Because the bottom-level bounds the start time of a task, as-late-as-possible start time of a task can be used to generate the task schedule.

4.1.8. Software Implementation. The final step in the design process, depicted as ⑧ in Figure 2, is the software implementation of the control software. The network schedule from the previous design step is used by a tool called ^{TTE}Build from TTTech to generate the binary configuration files and C code configurations required for the implementation of communication on the platform. An integrated interpreter in ESMoL's abstract intermediate language assembles the C code files generated by RTW and ^{TTE}Build with glue code files and automatically generates a Makefile. After compilation, the executables are deployed onto the respective ECUs.

The proposed software development process and toolchain provides flexibility and convenience in the rapid prototyping of automotive control software while at the same time ensuring that the models are correct at the different stages of design. In the design of automotive control application, it is typical to test multiple configurations and refinements in multiple iteration. Hence, the model-based approach provides the ability to quickly modify models and parameters to reflect changes and subsequently generate deployable software components for testing on the platform.

5. System Architecture

Figure 3 shows the system architecture for the experimental platform used in the evaluation of the automotive software development process.

5.1. Physical Layer. The physical layer modeling the dynamics of the automotive system encompasses two main components as described as follows.

5.1.1. Design/Visualization PC. The design/visualization PC represents the computing platform, running Windows operating system, for the dynamic modeling of a vehicle using CarSim as well as the initial control design and testing using Matlab/Simulink. CarSim is a commercially available parameter-based vehicle dynamics modeling software. It facilitates the efficient simulation and analysis of the behavior of four-wheeled vehicles in response to various inputs such as steering, braking, and acceleration [32]. The design/visualization PC is also used for the visualization and reporting of results from various experiments.

5.1.2. Target PC. The Target PC is a National Instruments LabVIEW Real-Time target running NI's Real-Time Module which provides a complete solution for creating reliable, stand-alone real-time systems [33]. In the experimental platform, the vehicle's physical dynamics modeled in CarSim is deployed on the Target PC during experiments. The Target PC is also integrated with a TTTech PCIe-XMC card [30] which enables the seamless integration and communication with ECUs on the time-triggered network supported by the TTEthernet switch.

5.2. Platform Layer. The platform layer is modeled by the TTEthernet switch and ECUs. These components are described as follows.

5.2.1. TTEthernet Development Switch. The TTEthernet Development Switch is an 8-port 100 Mbps system which supports 100 Base-TX Ethernet and enables hard real-time communication based on the TTEthernet protocol. It supports a star network topology. In Figure 3, the end systems comprised of the ECUs and the XMC card communicate with each other through the switch. The switch operates based on user defined configurations based on an experimental scenario. The configurations are specified in our model-based tool, ESMoL.

5.2.2. Electronic Control Units (ECUs). In Figure 3, the network depicts four ECUs, but there could possibly be more or fewer number of ECUs connected at a time based on a specific configuration. In our framework, an ECU is an IBX-530W-ATOM box with an Intel Atom CPU running a Real-Time Linux (RT-Linux) operating system. Each ECU is integrated with a TTEthernet Linux driver using an implementation of the TTEthernet protocol to enable the communication with other end systems in the TTEthernet network. Controller software components are deployed on the ECUs for the execution of automotive control applications. The controller software components that are deployed on each ECU are generated from the software design process for the controller specified in ESMoL. These software components execute in kernel space of the RT-Linux running on each of the ECUs and utilizes the synchronized time base of TTEthernet.

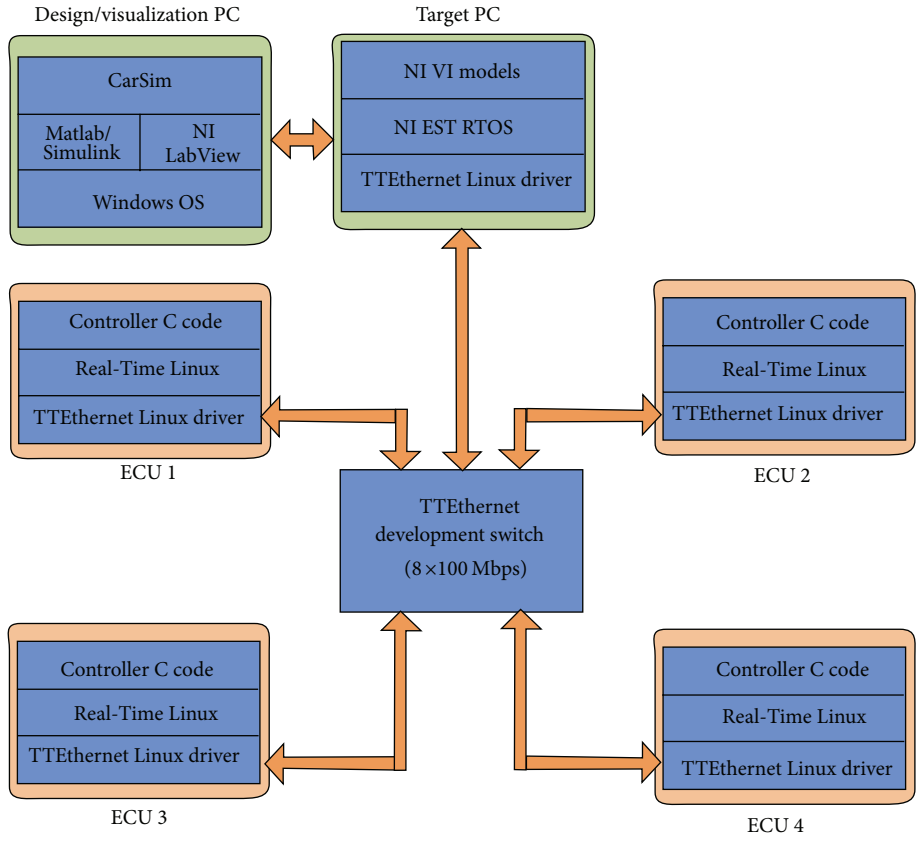


Figure 3: System Architecture for the Experimental Platform.

6. Control Design

In this section, we describe the controller design for the ACC. The operation of an ACC involves the use of a radar system which is attached to the front of the vehicle in order to detect when a vehicle is in the ACC-equipped vehicle's detectable view. When a vehicle is detected by the radar, the ACC system will control the distance between the ACC-enabled vehicle and the leading vehicle. In the absence of a leading vehicle in the ACC-enabled vehicle's path, the ACC system controls the vehicle to maintain a driver set velocity, essentially behaving like the conventional cruise control system.

The vehicle model used for the ACC design only considers the longitudinal motion of the vehicle.

6.1. Longitudinal Vehicle Model. The longitudinal vehicle model is typically based on the following assumptions [17].

A1: The torque converter is locked which implies that the engine speed is algebraically proportional to the vehicle speed via the gear ratios. *A2*: The tire slip is negligible.

The longitudinal dynamics of a vehicle can be described by the following equation provided that assumptions *A1* and *A2* hold:

$$T_e - R_g\left(T_b + M_{\mathrm{rr}} + hF_a + mgh\sin\theta\right) = \beta a, \tag{1}$$

where

$$\beta = \frac{\left[J_e + R_g^2\left(J_{wr} + Jwf + mh^2\right)\right]}{R_g h},$$
$$F_a = C_a v^2. \tag{2}$$

Table 1 provides a summary of the parameter definitions. Figure 4 shows a block diagram of the ACC system. The ACC is hierarchically divided into two levels of control: the upper level controller and the low level controller.

6.1.1. Upper Level Controller. The main functionality of the upper level controller is to compute the desired acceleration for the ACC-equipped vehicle that achieves the desired spacing or velocity. As depicted in Figure 4, the upper level controller, using the driver inputs, the radar measurements, and the current distance and velocity of the ACC-equipped vehicle relative to a leading vehicle, computes the desired acceleration that is required to achieve the desired spacing or velocity. The computed acceleration command is sent to the lower level controller to compute and implement the corresponding actuation commands as needed. The upper level controller can operate in two main control modes.

(a) *Velocity Control*: In this mode, the radar does not detect any vehicle in the path of the ACC-equipped vehicle. In this mode, the ACC essentially acts like the

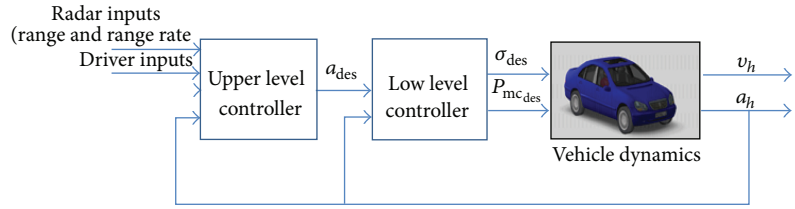


Figure 4: Adaptive cruise control system.

Table 1: Parameters for the longitudinal vehicle dynamics.

Parameter	Definition
β	Lumped inertia
F_a	Aerodynamic drag force
T_e	Net engine torque
T_b	Brake torque
R_g	Gear ratio
M_{rr}	Rolling resistance moments
h	Effective wheel radius
m	Total curb mass of the vehicle
J_e	Inertia of engine
J_{wr}	Inertia of rear axle
J_{wf}	Inertia of front axle
C_a	Aerodynamic drag coefficient
v	Velocity of the vehicle
a	Acceleration of vehicle
θ	Inclination angle of the road
g	Gravitational acceleration

conventional cruise controller. Therefore, the ACC-equipped vehicle's velocity is maintained at the target velocity set by the driver. The control law for computing the acceleration command is a proportional controller defined as.

$$a_{\text{des}} = K_1 * (v_d - v_h), \tag{3}$$

where K_1 is a control gain, v_d is the user-set velocity, and v_h is the velocity of the host or ACC-equipped vehicle.

(b) *Spacing Control*: The spacing control mode is entered when the radar detects a leading vehicle in the ACC-equipped vehicle's path, and the ACC system controls the vehicle to maintain a desired distance based on the velocity of the host vehicle and a user-specified time gap. This desired distance, S_d, can be defined as

$$S_d = \Delta + (v_h * t_{\text{gap}}), \tag{4}$$

where Δ is the desired distance to be maintained in the case where the leading vehicle comes to a complete stop, and t_{gap} is the user-specified time gap with typical values in the range of about 0.7–1.8 seconds.

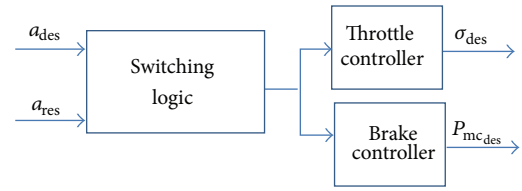


Figure 5: Low level controller.

The control law used in computing the desired acceleration in this mode is

$$a_{\text{des}} = \min(a_1, a_2), \tag{5}$$

where a_1 is computed similar to the desired acceleration in (3), and a_2 is computed as follows

$$a_2 = K_2 * (v_l - v) + K_3 (S_d - S_a), \tag{6}$$

where S_a is the gap distance measured by the radar, v_l is the velocity of the leading vehicle, K_2, and K_3 are control gains.

6.1.2. Low Level Controller. The main objective of the low level controller is twofold. First, using the desired acceleration command from the upper level controller, the lower level controller determines whether to apply braking control or throttle control. Secondly, the required control command is applied to the vehicle in order to achieve the desired acceleration. The applied control command is either throttle angle command, σ_{des}, or master cylinder pressure command, $P_{mc\,\text{des}}$.

(a) *Switching Logic*: The switching logic component shown in Figure 5 determines, based on the desired acceleration from the upper level controller, whether a brake torque or engine torque is required to achieve the desired acceleration. Typically, it is common to assume that a simple logic for choosing between brake and engine control can be based on the sign of the desired acceleration; that is, if the acceleration is greater than or equal to zero, then engine control should be applied, otherwise the brake control should be applied. This approach neglects the fact that with no control inputs, the engine torque is not necessarily zero. Thus, a better alternative is to consider the residual acceleration, a_{res}, due to the presence of engine torque when no control inputs are

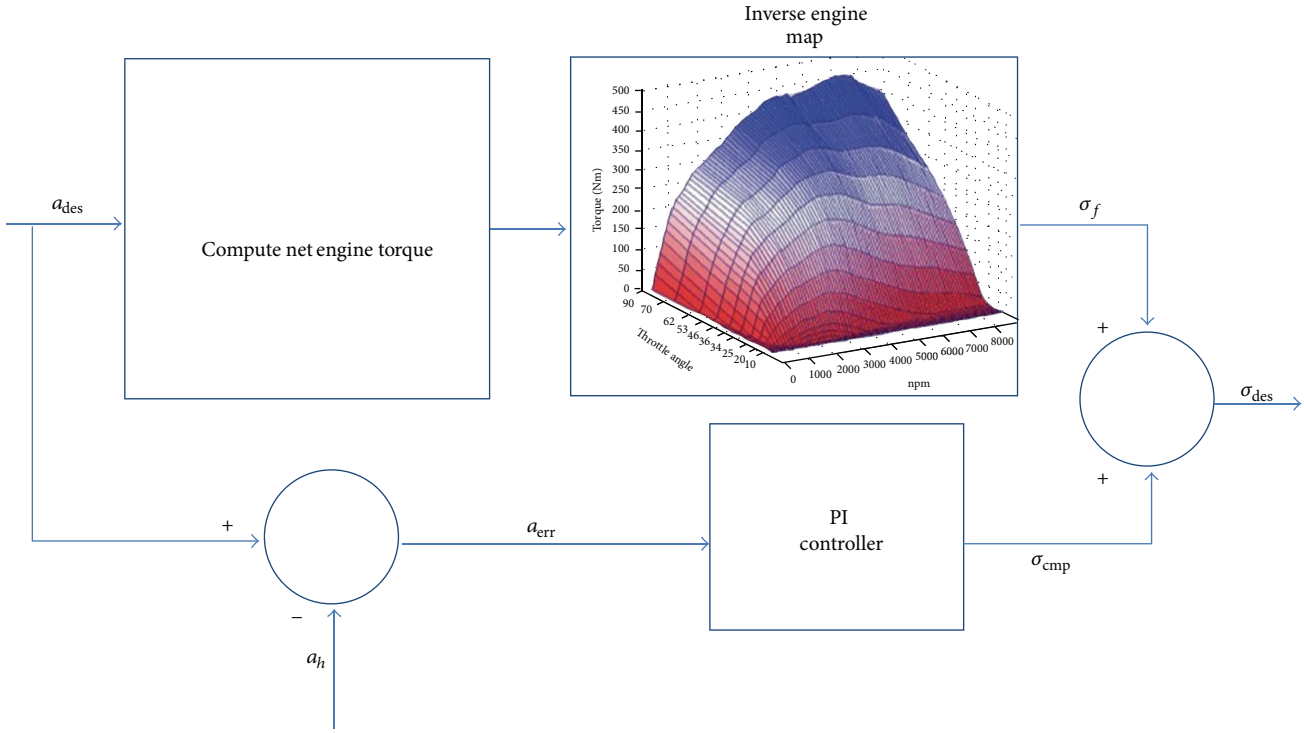


Figure 6: Throttle controller.

introduced [16]. Based on this approach, the engine torque can be subdivided into two parts: minimum or closed throttle torque, T_{ect}, and the portion subject to control, T_{ec}. Substituting these two components for T_e in (1) we have

$$T_{\text{ec}} + T_{\text{ect}} - R_g\left(T_b + M_{\text{rr}} + hF_a + mgh\sin\theta\right) = \beta a. \tag{7}$$

In the absence of control inputs $T_{\text{ec}} = T_b = 0$, the residual acceleration, a_{res}, as a result of closed-throttle torque, can be obtained as

$$a_{\text{res}} = \frac{1}{\beta}\left[T_{\text{ect}} - R_g\left(T_b + M_{\text{rr}} + hF_a + mgh\sin\theta\right)\right]. \tag{8}$$

Once the residual acceleration is calculated, the switching law uses the following criteria to determine whether engine or braking is required:

$$\begin{aligned} a_{\text{des}} \geq a_{\text{res}} &\Longrightarrow \text{throttle control},\\ a_{\text{des}} < a_{\text{res}} &\Longrightarrow \text{brake control}. \end{aligned} \tag{9}$$

In order to prevent rapid chattering between the engine control and brake control models, a small hysteresis, h_{yst}, is introduced. This results in the following switching law:

$$\begin{aligned} a_{\text{des}} \geq a_{\text{res}} + h_{\text{yst}} &\Longrightarrow \text{throttle control},\\ a_{\text{des}} < a_{\text{res}} - h_{\text{yst}} &\Longrightarrow \text{brake control}. \end{aligned} \tag{10}$$

Once the decision of the control mode is determined, the corresponding controller converts the desired acceleration into the appropriate input to the vehicle.

(b) *Throttle Control*: When engine control torque is required, the throttle controller converts the computed desired acceleration into a throttle command that is required to achieve the acceleration. Figure 6 shows the block diagram for the throttle control law. The controller first converts desired acceleration into a desired engine net torque. The desired net torque can be computed based on (1) with $T_b = 0$ as follows:

$$T_{e\,\text{des}} = \beta a_{\text{des}} + R_g\left(T_b + M_{\text{rr}} + hF_a + mgh\sin\theta\right). \tag{11}$$

The computed desired torque is converted into a throttle angle command by using an inverse engine map for the vehicle based on the current engine speed, w_e. This is performed by interpolating the data from an experimentally determined engine map lookup table for the vehicle. Consider

$$T_{e\,\text{des}} \Longrightarrow \text{inverse engine map} \Longrightarrow \sigma_f. \tag{12}$$

Due to the potential inaccuracies from the obtained engine map, an additional proportional-integral (PI) controller is integrated in the throttle controller to ensure that the desired acceleration is achieved.

(c) *Brake Control*: Figure 7 shows the brake controller. When braking control torque is required, the brake controller converts the desired acceleration to an

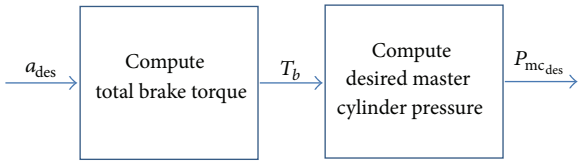


Figure 7: Brake controller.

appropriate brake command. The controller first computes the desired brake torque from the desired acceleration by using (7) and with $T_{\text{ec}} = 0$, which results in the following equation:

$$T_{\text{ect}} - R_g \left(T_{b\,\text{des}} + M_{\text{rr}} + hF_a + mgh \sin\theta\right) = \beta a_{\text{des}}. \tag{13}$$

The computed desired brake torque is then converted to an equivalent master cylinder pressure which is applied as the control input to the vehicle. The master cylinder pressure, P_{mc}, can be related to the brake torque by the following equation:

$$T_b = K_b \left(P_{\text{mc}} - P_{\text{po}}\right), \tag{14}$$

where K_b is a brake gain, and P_{po} is the push-out pressure required to engage brake.

7. ACC Controller Software Design and Implementation

This section describes the ACC controller software design and implementation based on the proposed development process. The initial ACC controller design is performed in Matlab/Simulink. Subsequently, following the automated software development process described in Section 4, the Matlab/Simulink model is imported into the ESMoL environment.

Figure 8 shows the logical software architecture depicting the logical interconnections of ACC controller components. In this model, each component represents a task. The ACC controller has four tasks, Instrument Cluster Sense (*InstrClstrSens*), Instrument Cluster Actuate (*InstrClstrAct*), Upper Level Controller (*UpperLevelController*), and Low Level Controller (*LowLevelController*). The *InstrClstrSens* and *InstrClstrAct* correspond to the tasks for processing the inputs and outputs of the ACC controller, respectively, while *UpperLevelController* and *LowLevelController* tasks implement the functionality of the controller designed in Section 6. Two tasks, *InputHandler* and *OutputHandler*, are used to represent the sensing and actuation for the vehicle dynamics developed in CarSim. In addition to the ACC controller, we also introduced another task, *RearViewMonitor*, in order to emulate the execution of other tasks on the ECUs.

In Figure 9, the network/platform configurations are explicitly modeled in the ESMoL. Three ECUs are specified and are denoted as ECU1, ECU2, and ECU3. The RT-Target node represents the Target-PC or computing node where the CarSim vehicle dynamics is executed. In order to represent the sensors and actuators of a vehicle, two virtual I/O devices are used. For the networked system, specific parameters for TTEthernet need to be defined, such as hyperperiod, bandwidth, time slot size, clock synchronization cycle, and synchronization precision. These specified parameters are used to generate the TTEthernet configuration script in ESMoL.

Figure 10 shows the deployment model of ACC control software. Dashed arrows represent assignment of components to their respective processors, and solid lines represent assignment of message instances (component ports) to communication channels (port objects) on the processor. We manually deploy *InstrClstrSens* and *InstrClstrAct* on ECU1, *UpperLevelController* on ECU2, and *LowLevelController* and *RearViewMonitor* on ECU3.

In Figure 11, the timing and execution model for tasks and message transfers of the ACC control system are shown. The ACC controller runs at a period of 10 ms. Since the TTEthernet provides a synchronized time base for communication, all the message transfers are attached with *TTExecInfo* to indicate time-triggered communication. For example, *ULMExec* is used to specify the timing for the message transfer from *InstrClstrSens* to *UpperLevelController*. We set the following parameters: the execution period which is the hyperperiod of the TTEthernet configuration, the desired offset which is used to specify the *initstart_ns* field of TT message in the generated TTEthernet configuration script, and the worst case duration which is the worst case communication time of the TTEthernet. The TTEthernet driver on each ECU has a scheduler that utilizes the synchronized time base, which can invoke the tasks according to a static schedule. Thus, all the tasks are executed according to the time-triggered paradigm. We specify the *TTExecInfo* for each task. For instance, *ULExec* specifies the execution time of *UpperLevelController* in the 10 ms period. Before scheduling, we only need to provide the execution period and the task's worst case execution time, which is determined empirically.

Using the Stage 1 interpreter as described in Section 4, the ESMoL model is transformed to an ESMoL_Abstract model in the form of XML file. A Stage 2 interpreter is used to generate the TTEthernet network configuration for scheduling and task scheduling. In this case of task scheduling, the critical path is simple as follows: *InputHandler* $\mapsto$ *InstrClstrSens* $\mapsto$ *UpperLevelController* $\mapsto$ *LowLevelController* $\mapsto$ *InstrClstrAct*. After network/task scheduling, the schedule information is updated into the ESMoL and ESMoL_Abstract models automatically. The integrated interpreter then uses the updated ESMoL_Abstract model to assemble all the codes for compilation.

8. Experimental Evaluation

In this section, we present the experimental results from the testing of the ACC system on the experimental platform. The experiments consist of two vehicles, a leading vehicle and an ACC-equipped vehicle which we refer to as the host vehicle. When the ACC feature is enabled and engaged, the host vehicle starts in the velocity control mode and

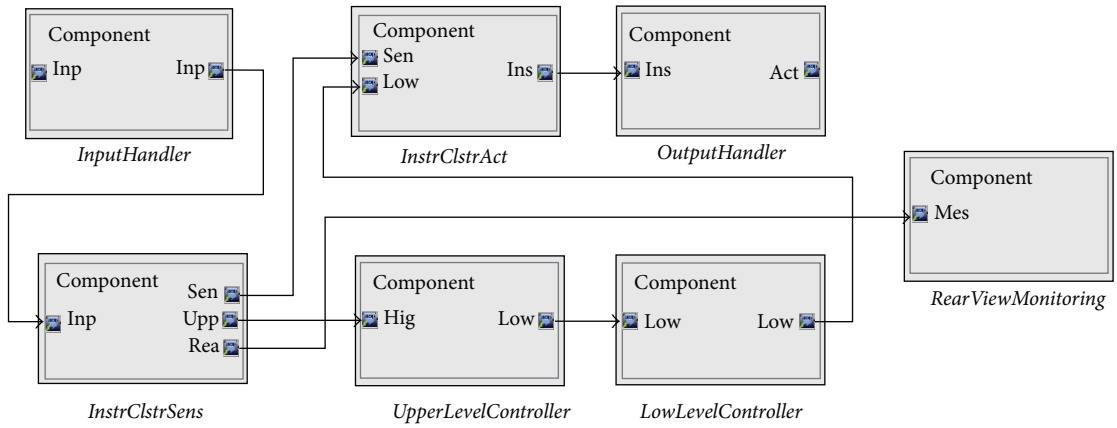


Figure 8: Logical software architecture of ACC controller.

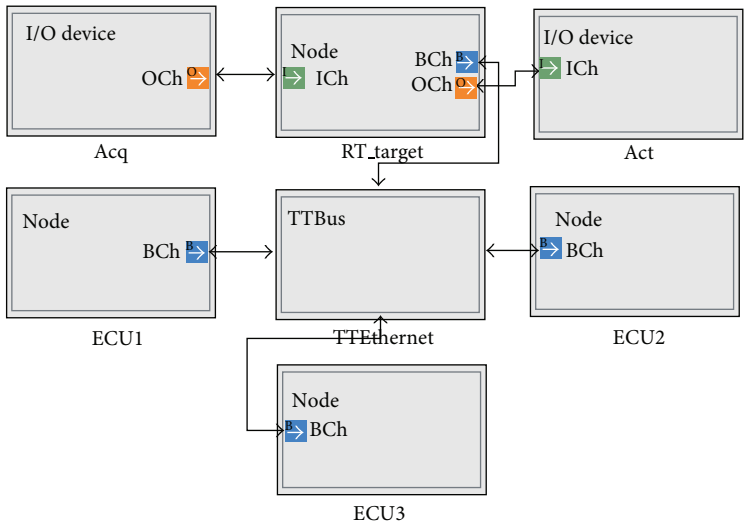


Figure 9: Hardware platform representation of the experimental platform.

Table 2: Parameters for the ACC experiments.

Parameter	Value
K_1	0.5
K_2	2
K_3	0.7
Range of ACC radar	100 m
t_{gap}	1.5 s
Δ	10 m
θ	0 rad
Sampling period	0.01 s
m	1650 kg

maintains a driver-set velocity, and when a leading vehicle is detected, the ACC transitions into the spacing control mode in order to maintain a desired distance based on the driver-set time gap and host velocity as described in Section 6. The parameters for the experiments are provided in Table 2.

The experimental setup is based on the system architecture described in Section 5. The generated software components for the ACC are distributed over three ECUs as described in Section 4. In this experiment, the velocity of the leading vehicle starts at an initial value of 60 km/h. The initial global longitudinal positions of the leading vehicle and the host vehicle are 130 m and 0 m, respectively, which means that the host vehicle radar is initially out of range. The host vehicle initially starts at an initial velocity of 65 km/h with a driver set target velocity of 80 km/h. We present four scenarios based on the driving behavior of the leading vehicle. In addition, we compare the results obtained during the control design stage using Matlab/Simulink with those obtained from the experimental platform. The modeled scenarios are described as follows.

(1) *Scenario 1*: *Velocity Control*. In this scenario, there is no leading vehicle in front of the host vehicle within the range of the radar, and, hence, the host vehicle is under the velocity control mode or the conventional

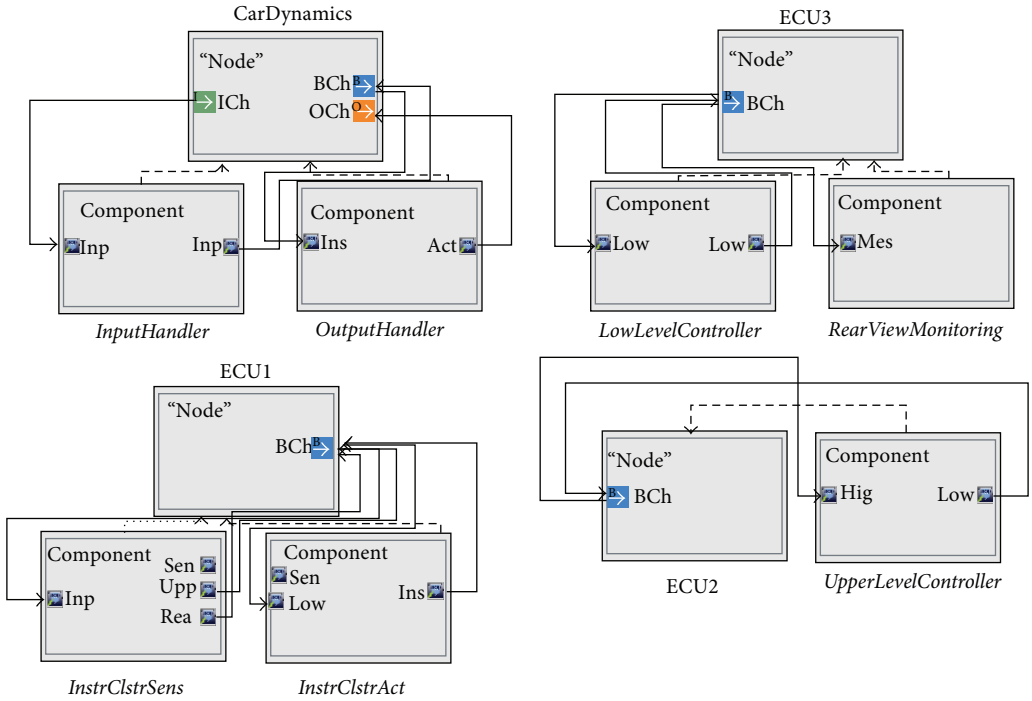


Figure 10: Platform deployment aspect of ACC controller.

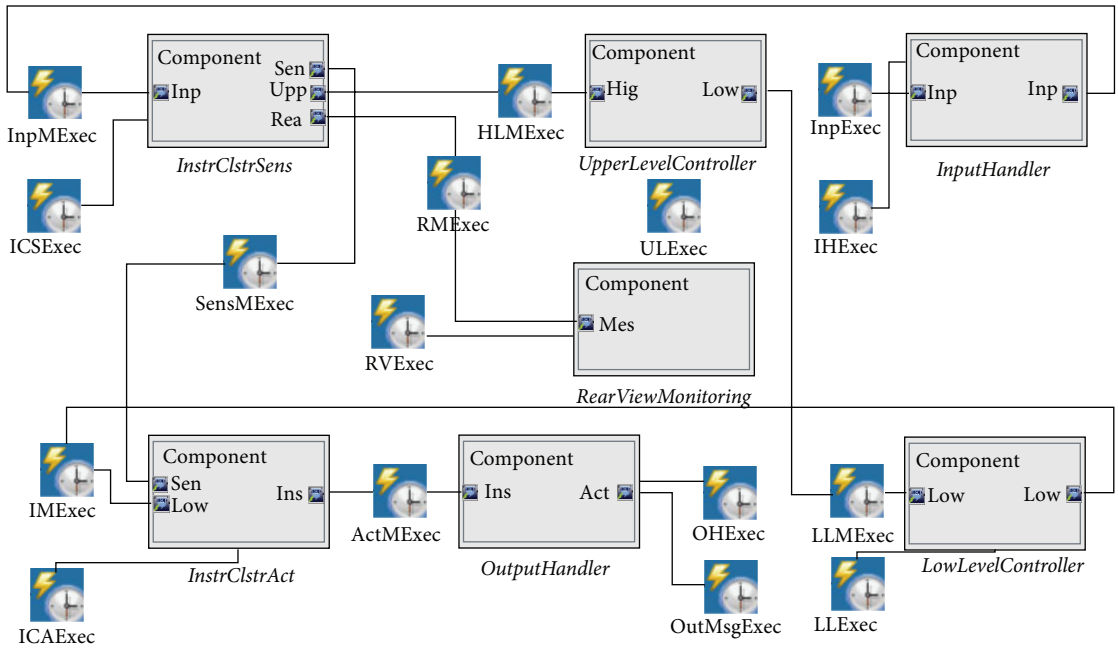
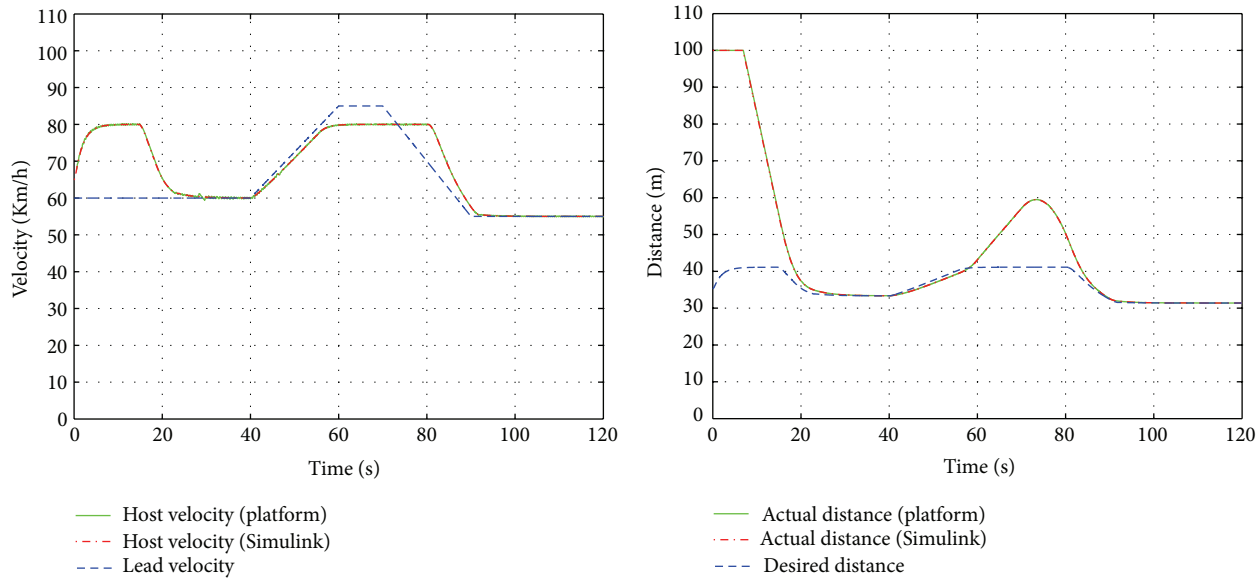


Figure 11: Timing/execution model of ACC controller.

cruise control. Figure 12 shows the results from the control design phase using Matlab/Simulink and the results from the execution on the experimental platform, respectively. This scenario can be observed between the time segment of 0–10 s.

(2) *Scenario 2*: *Spacing Control*. In this scenario, the radar detects a slower leading vehicle and transitions to the spacing control mode to control the distance between the two vehicles to a driver-set time gap. The desired gap distance is attained when the two vehicles travel at the same velocity. This scenario can be observed between the time segment 10–40 s in Figure 12.

(3) *Scenario 3*: *Leading Vehicle Speeds Up*. In this scenario, while in spacing control mode, the leading vehicle



(a) Velocity plots

(b) Gap distance plots

Figure 12: Gap Distance and Velocities.

begins to speed up. As a result, the velocity of the host vehicle also increases in order to maintain a desired velocity. This scenario can be observed between the time segment 40–60 s in Figure 12. From the plots, when the leading vehicle reaches and a maintains a velocity of 85 km/h after 60 s, the host vehicle maintains it is velocity at the driver-set velocity of 80 km/h, since the driver-set velocity is the maximum achievable velocity of the host vehicle based on the ACC algorithm. It can be seen from the distance plots that the distance between the two vehicles increases due to the difference in velocity of the vehicles.

(4) *Scenario 4*: *Leading Vehicle Slows Down*. In this scenario, the leading vehicle slows down, and as a result, the host vehicle also starts to decrease its velocity in order to maintain the desired spacing between the vehicles. This scenario can be observed between the time segment 70–90 s in Figure 12. At approximately 105 s, the two vehicles starts to travel at the same velocity again.

To highlight the importance of the experimental platform for the early assessment for control software before actual deployment on a real vehicle, we compared the results obtained from running the scenarios in Matlab/Simulink to those obtained from running the scenarios on experimental platform. This comparison provides insight to potential impacts of the deployment of the control software on a prototype platform. Figure 13 shows the velocity plots from the simulation in Matlab/Simulink from the control design stage and the results obtained from deploying the resulting software components on the platform. By zooming in on the velocity plots between 25 and 45 s, we notice that compared

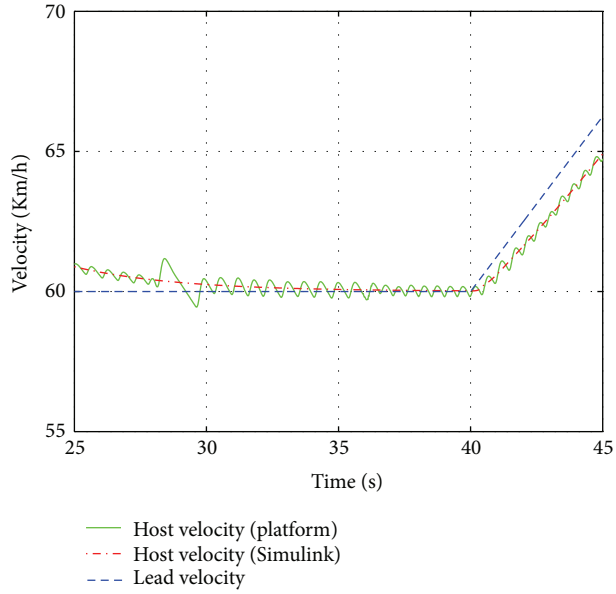


Figure 13: Comparison of Simulink simulation results and results from the experimental platform.

to the Simulink results, the results from the deployed software exhibit some oscillations with an amplitude 0.6 km/h. Although this is barely noticeable, it is important to note the difference in the results which can be attributed to platform effects as a result of deploying the controller on the platform. This implementation limitation is due to the fact that the computation on the RT-Target is not synchronized with the communication.

9. Discussion and Conclusion

Our proposed framework addresses the complex interactions that emerge as a result of integrating the various design layers of CPS. The proposed approach provides an end-to-end methodology and a toolchain for the development of automotive control software. The toolchain is based on simple and visually intuitive formal languages that restrict component interactions in a well-defined manner in order to ensure a "correct-by-construction" design of automotive control software. We employ an experimental platform based on the time-triggered paradigm in order to facilitate the design, deployment, and hardware-in-the-loop simulation of automotive control software. We applied the design methodology to a case study on the adaptive cruise control and presented the experimental results based on realistic scenarios.

In regards to future work, we would like to study the possible interactions between multiple automotive control systems deployed together and formally define their interacting behavior as well as the impact on the overall system. This interacting behavior is important because there are possible cases where these systems can have conflicting objectives, and, hence, a clear understanding of their underlying interaction is crucial.

Conflict of Interests

The authors do not have any conflict of interests with any of the commercial identities mentioned in this paper.

Acknowledgments

The authors would like to thank Shige Wang for his insightful suggestions and discussions. This work is supported in part by the National Science Foundation (CNS-1035655 and CCF-0820088), US Army Research Office (ARO W911NF-10-1-0005), and Lockheed Martin. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Government.

References

[1] J. Mossinger, "An insight into the hardware and software complexity of ecus in vehicles," in *Advances in Computing and Information Technology*, vol. 198 of *Communications in Computer and Information Science*, pp. 99–106, 2011.

[2] J. Mossinger, "Software in automotive systems," *IEEE Software*, vol. 27, no. 2, pp. 92–94, 2010.

[3] A. Michailidis, U. Spieth, T. Ringler, B. Hedenetz, and S. Kowalewski, "Test front loading in early stages of automotive software development based on AUTOSAR," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '10)*, pp. 435–440, March 2010.

[4] A. Sangiovanni-Vincentelli, "Electronic-system design in the automobile industry," *IEEE Micro*, vol. 23, no. 3, pp. 8–18, 2003.

[5] J. A. Cook, I. V. Kolmanovsky, D. McNamara, E. C. Nelson, and K. V. Prasad, "Control, computing and communications: technologies for the twenty-first century model T," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 334–355, 2007.

[6] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded system design for automotive applications," *Computer*, vol. 40, no. 10, pp. 42–51, 2007.

[7] T. Stahl and M. Volter, *Model-Driven Software Development*, John Wiley and Sons, 2006.

[8] J. Porter, G. Hemingway, H. Nine et al., "The esmol language and tools for high-confidence distributed control systems design—part 1: language, framework, and analysis," Tech. Rep. ISIS-10-109, Vanderbilt University, 2010.

[9] A. Ledeczi, M. Maroti, A. Bakay et al., "The generic modeling environment," in *Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP '01)*, May 2001.

[10] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[11] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1222, 2005.

[12] M. Broy, S. Chakraborty, D. Goswami et al., "Cross-layer analysis, testing and verification of automotive control software," in *Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT '11)*, pp. 263–272, 2011.

[13] U. Drolia, Z. Wang, Y. Pant, and R. Mangharam, "AutoPlug: an automotive test-bed for electronic controller unit testing and verification," in *Proceedings of the 14th IEEE International Conference on Intelligent Transportation Systems (ITSC '11)*, pp. 1187–1192, 2011.

[14] W. Hu, M. Wang, and Y. Lin, "On the software-based development and verification of automotive control systems," in *Proceedings of the 33rd IEEE Annual Conference of the Industrial Electronics Society (IECON '07)*, pp. 857–862, 2007.

[15] A. Ray, I. Morschhaeuser, C. Ackermann, R. Cleaveland, C. Shelton, and C. Martin, "Validating automotive control software using instrumentation-based verification," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, pp. 15–25, November 2009.

[16] J. C. Gerdes and J. K. Hedrick, "Vehicle speed and spacing control via coordinated throttle and brake actuation," *Control Engineering Practice*, vol. 5, no. 11, pp. 1607–1614, 1997.

[17] J. K. Hedrick and P. P. Yip, "Multiple sliding surface control: theory and application," *Journal of Dynamic Systems, Measurement, and Control*, vol. 122, no. 4, pp. 586–593, 2000.

[18] P. A. Ioannou and C. C. Chien, "Autonomous intelligent cruise control," *IEEE Transactions on Vehicular Technology*, vol. 42, no. 4, pp. 657–672, 1993.

[19] K. Yi, S. Lee, and Y. D. Kwon, "An investigation of intelligent cruise control laws for passenger vehicles," *Journal of Automobile Engineering*, vol. 215, no. 2, pp. 159–169, 2001.

[20] P. S. Fancher, H. Peng, and Z. Bareket, "Comparative analyses of three types of headway control systems for heavy commercial vehicles," *Vehicle System Dynamics*, vol. 25, no. 1, pp. 139–151, 1996.

[21] B. A. Güvenç and E. Kural, "Adaptive cruise control simulator: a low-cost, multiple-driver-in-the-loop simulator," *IEEE Control Systems Magazine*, vol. 26, no. 3, pp. 42–55, 2006.

[22] K. Breuer and M. Weilkes, "A versatile test vehicle for acc-systems and components," Tech. Rep., 1999.

[23] A. R. Girard, S. Spry, and J. K. Hedrick, "Intelligent cruise-control applications: real-time, embedded hybrid control software," *IEEE Robotics and Automation Magazine*, vol. 12, no. 1, pp. 22–28, 2005.

[24] E. Armengaud, A. Tengg, M. Driussi, M. Karner, C. Steger, and R. Weiß, "Automotive software architecture: migration challenges from an event-triggered to a time-triggered communication scheme," in *Proceedings of the 7th Workshop on Intelligent Solutions in Embedded Systems (WISES '09)*, pp. 95–103, June 2009.

[25] J. Sztipanovits, X. Koutsoukos, G. Karsai et al., "Toward a science of cyber-physical system integration," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 29–44, 2012.

[26] J. Eker, J. W. Janneck, E. A. Lee et al., "Taming heterogeneity—the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–143, 2003.

[27] MathWorks—real-time workshop, http://www.mathworks.com/products/rtw/.

[28] B. Chou and T. Erkkinen, "Converting models from floating point to fixed point for production code generation," Matlab Digest, November 2008.

[29] E. Magyari, A. Bakay, A. Lang et al., "Udm: an infrastructure for implementing domain-specific modeling languages," in *Proceedings of the 3rd OOPSLA Workshop on Domain-Specific Modeling*, Anaheim, Calif, USA, 2003.

[30] Ttethernet, http://www.tttech.com/en/products/ttethernet/.

[31] O. Sinnen, *Task Scheduling for Parallel Systems*, Wiley-Interscience, New York, NY, USA, 2007.

[32] Carsim, http://www.carsim.com/.

[33] National Instruments, http://www.ni.com/.