Co-Simulation Framework for Design of Time-Triggered Cyber Physical Systems

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

Institute for Software Integrated Systems (ISIS)
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN, USA
{zhenkai.zhang, emeka.eyisi, xenofon.koutsoukos, joe.porter}@vanderbilt.edu

ABSTRACT

Designing cyber-physical systems (CPS) is challenging due to the tight interactions between software, network/platform, and physical components. A co-simulation method is valuable to enable early system evaluation. In this paper, a co-simulation framework that considers interacting CPS components for design of time-triggered (TT) CPS is proposed. Virtual prototyping of CPS is the core of the proposed framework. A network/platform model in SystemC forms the backbone of the virtual prototyping, which bridges control software and physical environment. The network/platform model consists of processing elements abstracted by real-time operating systems, communication systems, sensors, and actuators. The framework is also integrated with a model-based design tool to enable rapid prototyping. The framework is validated by comparing simulation results with the results from a hardware-in-the-loop automotive simulator.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Simulation; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design, Performance

Keywords

Co-simulation, Virtual prototyping, CPS, SystemC

1. INTRODUCTION

Cyber-physical systems (CPS) are complex systems that are characterized by the tight interactions between the physical dynamics, computational platforms, communication networks, and control software. Many CPS are safety-critical control systems, such as automotive vehicles, aircraft, and industrial processes. The complex cyber-physical interactions make the composability and predictability of these systems very challenging. Moreover, the economic factors, such as persistent effort for low production costs and tight time-to-market, further complicate the development.

Time-triggered architecture (TTA) has been proposed and widely used to address the composability and predictability challenges in CPS. As a computation platform, TTA provides precise timing and fault-tolerance guarantees for both control software and networked data communications [9]. In addition, there have been on-going efforts towards the standardization of communication systems based on time-triggered (TT) paradigm, especially in-vehicle networks (e.g. FlexRay and TTEthernet), with the overall goal of ensuring highly reliable, deterministic, and fault-tolerant system performance [16] [14].

When designing CPS, a practical approach is to consider three design layers, which include the physical layer, the network/platform layer, and the software layer, as shown in Fig. 1 [19]. The physical layer represents physical components and their interactions, whose behavior is governed by physical laws and is typically described in continuous time using ordinary differential equations. The network/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. The software layer represents the software components which are connected based on an input/output model implying a notion of causality.

This three-layer design approach can be easily applied to TT CPS design. We often start designing its control system using a high level modeling language such as MATLAB/Simulink. The model serves as an executable specification and the equivalent source code, usually in C, can be generated automatically from it. At later design stages, the generated source code is deployed on a platform to perform the required functionality. It may not be possible to achieve the required control performance if elements from the three CPS design layers are designed separately and then integrated later. Interactions between the layers are very tight, so late integration is very likely to result in large design gaps that will be costly to resolve. In order to reduce the efforts

ICCPS'13, April 8–11, 2013, Philadelphia, PA, USA.

and costs and shorten time-to-market, it is important to get realistic control performance feedback at early design stages. However, the platform prototype is usually not available at early design stages and even if it is available, testing at the very beginning presents safety and economical challenges.

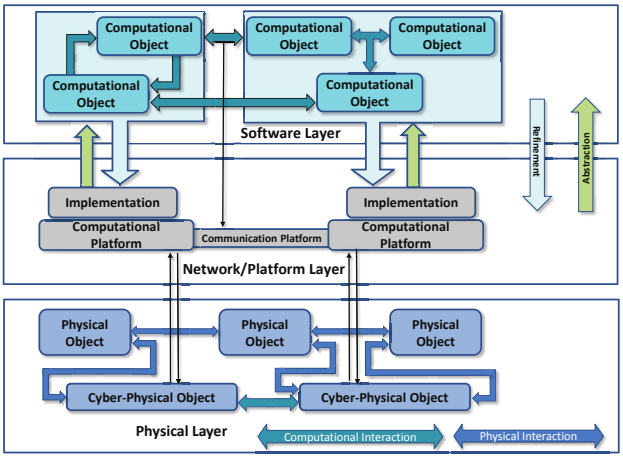

Figure 1: A Simplified View of Designing CPS: Three CPS Design Layers [19]

A cross-layer co-simulation framework that takes into account physical dynamics, control software, computational platforms, and communication networks becomes very crucial. The requirements for such a framework include: (1) it should contain models from three CPS design layers that can be integrated together; (2) the models should be at appropriate levels of abstraction, so that the simulation is efficient but accurate enough; (3) the scalability of the framework should allow simulation of large distributed CPS; (4) it should allow model-based rapid prototyping to improve the usability.

Co-simulation can be achieved by virtual prototyping. Virtual prototyping can take advantage of different modeling languages/tools and integrate them together to evaluate the whole CPS. Modeling cyber components in SystemC has begun to be dominant in the Electronic System-Level (ESL) design field. SystemC has become a *de facto* system-level design language for hardware/software (HW/SW) co-design and an IEEE standard [7]. By adding appropriate timing annotations, a SystemC model can reveal timing behavior of the corresponding HW/SW. SystemC also has a standardized library for realization of transaction level modeling (TLM) concepts. TLM focuses on what data is being transfered rather than how it is being transmitted, so a TLM model abstracts away certain communication details to speed up simulation while keeping sufficient accuracy.

This paper introduces a co-simulation framework that is used to facilitate CPS design. A virtual prototyping approach, which uses SystemC to model the cyber part and simulators of physical systems to model the physical part of a CPS, forms the core of the proposed framework. This paper mainly focuses on TT CPS. The main contributions of the paper include: (1) A co-simulation framework that centers on a detailed network/platform layer model in SystemC is proposed. The network/platform layer model, including processing elements (PEs) which are abstracted by real-time operating system (RTOS) models, TTEthernet communication systems, sensors, and actuators, enables TT computation and communication; (2) Rapid prototyping is realized by model transformations from a designed MATLAB/Simulink model to a front-end design environment model to the final virtual prototype; (3) TT communication in terms of its end-to-end transmission delay is validated against a real implementation of TTEthernet, and the framework is evaluated by using an automotive control system case study, which demonstrates the efficiency and accuracy of the approach.

The rest of paper is organized as follows: Section 2 describes the related work; Section 3 introduces the core of the framework, which is the virtual prototyping of CPS; Section 4 describes how to achieve rapid prototyping via a model-based design environment; Section 5 uses an automotive control case study to validate the framework and illustrate design space exploration; Section 6 concludes this work.

2. RELATED WORK

Co-simulation of CPS requires integrating different models of computation (MoCs). In [6], an operational behavioral semantics integrating discrete event MoC and continuous time MoC is proposed and illustrated by combining SystemC and MATLAB/Simulink. In [20], a similar behavioral semantics is proposed and demonstrated by integrating VDM++ and 20-sim. These papers present formal co-simulation frameworks, but they are not directly applicable to TT CPS design.

In [13], a methodology of virtual prototyping of CPS is proposed which combines SystemC, QEMU, and Open Dynamics Engine to achieve a holistic design view. In [8], a co-simulation environment based on a SH-2A CPU model is demonstrated by combining different design tools including CoMET, Saber, and MATLAB/Simulink. Again, these methods cannot be used for TT CPS design directly, and the approach in [8] does not support simulation of distributed CPS.

The TrueTime toolbox has been proposed and used in MATLAB/Simulink environment to enable CPS simulation [3]. The toolbox considers timing aspects introduced by computation and communication. However, it is difficult to integrate hardware models and support different abstraction levels. Further, preemption can only happen at points between segments which causes timing inaccuracy (e.g. interrupt handling), and the clock synchronization between computation and communication on a node is also implicit.

As a classical CPS domain, automotive has been gaining a lot of attention. In [12], a C-VHDL-MATLAB co-simulation approach for automotive control systems is proposed to deal with the joint design of software in C, hardware in VHDL, and mechanical components in MATLAB. However, the simulation of this approach is not efficient. In [10], using SystemC to help simulate and refine automotive software specified by AUTOSAR is given to deal with that timing simulation is not supported by AUTOSAR. Other work that introduces virtual prototyping in SystemC to co-simulate the automotive control systems is presented in [18] [22]. However, these approaches only consider the cyber part of the system and do not include a physical dynamics model. Hardware-in-the-loop (HIL) automotive simulators are also found in [4] and [5]. Compared to software-based simulation

frameworks, they are more expensive and usually not available at early design stages. Besides, their network/platform layers are often fixed which limit the initial development.

Our work focuses on TT CPS, and further we integrate the co-simulation framework with a model-based design tool for improving usability.

3. VIRTUAL PROTOTYPING OF CPS

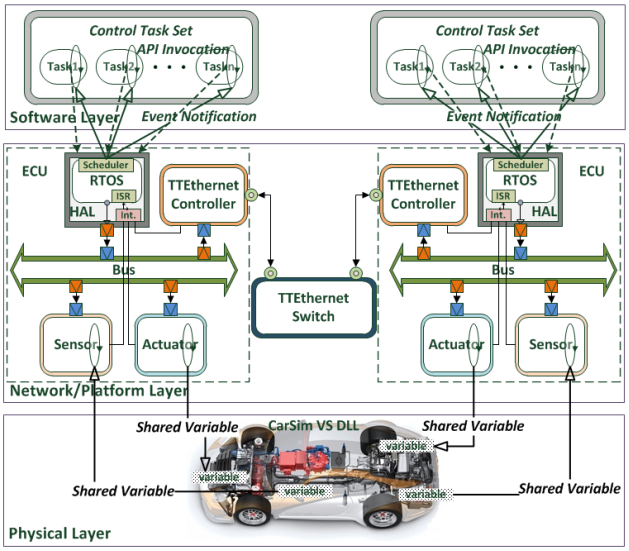

Figure 2: An Example: Virtual Prototyping of Automotive Control System by Three CPS Design Layers

The core of the co-simulation framework is the virtual prototyping of CPS, which is achieved by modeling each CPS design layer and exposing interfaces for integration. This section describes the virtual prototyping in detail, and illustrates the approach using an automotive vehicle example. Fig. 2 shows the virtual prototyping architecture used in an automotive control system and the interactions between three design layers.

3.1 Network/Platform Layer

As the backbone of the virtual prototyping of CPS, the network/platform layer bridges the software layer and the physical layer. The network/platform layer includes the network architecture and hardware platforms 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.

The behavior of this layer is captured by several models in SystemC: (1) a PE model for TT computation, (2) a clock model for synchronization, (3) a network model compliant with the TTEthernet protocol for TT communication between different nodes, and (4) sensor and actuator models for interaction with the physical environment. There are various network communication systems that can be used in the TT CPS design, such as TTP/C, FlexRay, and TTEthernet. In this paper, we choose TTEthernet to illustrate the framework, since it has been deployed in many CPS domains, such as automotive, aerospace, and industrial process

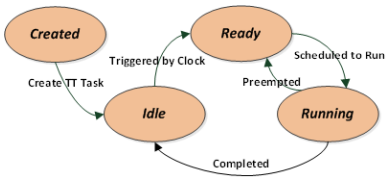

Figure 3: TT Task State Transitions

control.

In Fig. 2, there are two nodes (ECUs) connected by a TTEthernet switch forming the network/platform layer. In each node, the PE, TTEthernet controller, sensors, and actuators are connected through a bus.

3.1.1 Processing Element Modeling

Although an instruction set simulator can accurately mimic the behavior of a program running on a specific processor so as to give cycle accurate execution results, many drawbacks impede its use during early CPS design stages, including its low simulation speed for multi-processor simulation and the need to have the final target binary available. In order to accelerate the simulation while preserving accuracy, modeling the PE at higher abstractions levels is needed. An abstract RTOS model with accurate interrupt handling can serve as an efficient and effective model of the PE [21].

The abstract RTOS model in SystemC provides basic services to the software layer, which include task management, scheduling, interrupt handling, and inter-node communication, etc. It also exposes a set of primitive APIs to the software layer to facilitate the use of the model.

TT computation: In this framework, we interpret TT computation as follows: TT tasks are activated by the TT activator of the RTOS at the predefined times and put into the ready queue for scheduling. A TT task can be preempted and put back to the ready queue again, but it should not be blocked on any events (Fig. 3 shows the TT task state transitions). This mechanism can allow a more urgent system service program, such as an interrupt service routine (ISR), to preempt the execution of a TT task, and also allow the design of mixed time-/event-triggered systems. When using an off-line scheduling tool, the worst case preemption time should also be considered.

Scheduling: The scheduler is the heart of the abstract RTOS model, whose behavior depends on a specific scheduling algorithm. The scheduling algorithm of the RTOS can use rate monotonic, earliest deadline first, or other real-time scheduling algorithms to schedule the ready queue of the RTOS model. The ready queue consists of TT tasks and ISRs. As stated above, there is a TT activator that statically activates the tasks according to an *a priori* schedule table generated by an off-line scheduling tool. The timing properties of the scheduler mainly have two parameters which are scheduling overhead and context switching overhead. These timing properties are RTOS- and hardware platform-specific. Since it is not the focus of this paper, we assume the parameters are already available to system designers.

Interrupt handling: SystemC has some disadvantages for RTOS modeling, which can be summarized as non-interruptible *wait-for-delay* time advance and non-preemptive simulation processes. When an interrupt happens, it re-

quires the real-time system to react and handle it in a timely manner. Modeling an accurate preemption mechanism plays an important role in accurate PE modeling. We adopt the method from [21] which makes task use *wait-for-event* other than *wait-for-delay* to advance its execution time. A system call of the RTOS model taking execution time as its argument makes the task wait on a *sc_event* object which will be notified after the given execution time elapses if no preemption happens. When an interrupt happens and its corresponding ISR preempts the execution of the task, the notification of the *sc_event* object will be canceled and a new notification time will be calculated according to how much time the preemption took and how much execution time already passed.

Inter-task communication: Within a PE, the communication between TT tasks is through shared memory, since it can be accessed without race-condition. The communication between tasks running on different PEs is achieved by invoking send/receive APIs of the underlying abstract RTOS model. The corresponding messages will be delivered by the underlying TT communication system. State messages are used to prevent the TT tasks from blocking on reading.

PE Integration: In order to integrate the abstract RTOS model as a PE with other models on the network/platform layer through a bus, an additional Hardware Abstraction Layer (HAL) model is added to wrap the abstract RTOS model. The HAL model has a multi-port *sc_port* object to collect all the interrupt requests (IRQs) from peripherals in the node, and it is also a hierarchical SystemC channel which implements the pure virtual functions of a HAL interface class. The abstract RTOS model is connected to the HAL model through a *sc_port* object parameterized with the HAL interface class. When the abstract RTOS model communicates with other models, it will send/receive the data via the port by invoking the functions implemented by the HAL model, and the HAL model will initiate a bus transaction.

Clock synchronization: In this framework, the clock of a PE can be synchronized with the communication controller or be independent according to the configuration. As discussed in [11], if the clock is synchronized with the TTEthernet controller, all the operations are based on a global time base, and the control delay, δ, only depends on the offset and execution time of the actuation in a control period without variation. If the PE model and the network model do not share a global time base, the control delay will have a large variation which will be the sum of the periods of the computation and communication.

3.1.2 Clock Modeling

In TTA, time is the driving force for all TT operations. A TT communication system has its own synchronized global time base for correct operation. Computation can be synchronized with the communication or it can be driven by its own independent clock. Since time is the most important notion in TTA, modeling the independent clock and its synchronization service in SystemC becomes necessary. However, SystemC uses a discrete event simulation kernel which maintains global simulation time. If we simulate every tick of a clock with a drift, the simulation overhead will be too large, which can seriously slow down the simulation. Instead, we model the clock as follows: A random ppm value is assign to each clock in the interval [-MAX_PPM, -MIN_PPM] $\cup$ [MIN_PPM, MAX_PPM] (MAX_PPM and MIN_PPM are set by the user). According to the time-triggered schedule, the duration in clock ticks from the current time to the time when the next time-triggered action needs to take place is calculated. After that, we can get the duration in simulation time by taking into account its clock drift: $duration in simulation\, time = duration in clock ticks \times (tick\, time \pm drift)$, and then we can arrange a clock event with this amount of time by using the notification mechanism of *sc_event* in SystemC.

Because the clock will be adjusted periodically by the synchronization service, the arranged clock event will be affected (its occurrence in simulation time becomes sooner or later). In order to simulate this properly, the arranged clock event and its occurring time in clock ticks is stored in a linked list in order of occurrence. When a clock event occurs or its time has passed due to clock adjustment, it will be deleted from the linked list and processes pending on it will be resumed. When the clock is corrected, notifications of the arranged clock events are canceled and new simulation times for the notifications of the events are recalculated based on the corrected clock.

3.1.3 Communication System Modeling

There are two communication system models which are used in our framework for intra-node communication and inter-node communication respectively.

As shown in Fig. 2, in a node, the PE model, the TTEthernet controller model, and the sensor/actuator models communicate via a bus which is modeled in TLM-2.0 using its convenience sockets. Since the bus is an interconnect component, it uses a multi-initiator and multi-target sockets to support multiple connections. Other designed hardware models can also be connected to the bus as long as they provide interfaces that are compliant with TLM-2.0.

We model a concrete network protocol, TTEthernet, for inter-node communication. Three traffic classes, which are time-triggered (TT), rate-constrained (RC), and best-effort (BE), are supported in this protocol as well as a transparent traffic called protocol control frame (PCF) that is used for its synchronization service. There are two TTEthernet device types: the TTEthernet controller and the TTEthernet switch. Each node has at least one TTEthernet controller that can be connected by intermediate TTEthernet switch(es). The network topology is star or cascaded star so that the collision domain is segmented and only two TTEthernet devices which are directly connected may contend for the use of the medium.

The model is compliant with the TTEthernet standard [17]. Since the TTEthernet controller and switch have several common functions/services, we extract all the common functions and implement them in a class derived from the *sc_module* class. This class serves as the abstract base class of the TTEthernet controller and switch. It has pure virtual functions that need to be implemented by the controller or switch to define different behaviors of these two different devices. We use *SC_THREAD* processes to model the TT communication behavior and protocol state machines (PSMs) of TTEthernet, and also model its two-step synchronization mechanism that is used to establish the synchronized time base.

The main processes and their functions are listed in Tab. 1. The TT communication is realized by a scheduler pro-

Table 1: Processes in TTEthernet Model

Name	Main Function
send() & recv()	send/receive Ethernet frames
execSched()	signal TT frame transmission
releaseET()	arrange ET frame transmission
sync()	calculate clk. correction & adjust clk.
processPCF()	execute permanence function
compression()	compress PCFs
detectCliqueSync()	detect synchronous cliques
detectCliqueAsync()	detect asynchronous cliques
psmSM()	execute sync master PSM
psmCM()	execute compression master PSM
psmSC()	execute sync client PSM

cess (*execSched()*) which is responsible for signaling the send process (*send()*) to start a TT frame transmission according to a static schedule that relies on synchronized global time. The static schedule guarantees two TT frames never contend for transmission and is used by the TTEthernet device through a configuration file. Each TTEthernet device executes exactly one of the PSMs to maintain its role for synchronization, which are formulated in [17]. All TTEthernet devices can be classified into three different roles: synchronization masters (SMs), compression masters (CMs), and synchronization clients (SCs). Startup service of the PSMs tries to establish an initial synchronized global time to make devices operate in synchronized mode. When a device detects there is a synchronous/asynchronous clique scenario (*detectCliqueSync()*/*detectCliqueAsync()*), the restart service of PSMs will try to resynchronize itself. When operating in synchronized mode, TTEthernet uses a two-step synchronization mechanism: SMs dispatch PCFs to CMs, and CMs calculate the global time from the PCFs (i.e. "compress") and dispatch "compressed" PCFs to SMs and SCs. SMs and SCs receive "compressed" PCFs and adjust their clocks to integrate into the synchronized time base. When a PCF arrives, a dynamic PCF handler process (*processPCF()*) will be spawned to cope with this PCF. If the TTEthernet device is a CM, a dynamic compression process (*compression()*) will be spawned if there is no process handling corresponding integration cycle of the PCF. After receiving scheduled PCFs, the synchronization process (*sync()*) will be resumed to calculate the clock correction from the PCFs that are in-schedule, and after a fixed delay the clock will be adjusted by the calculated correction value.

The TTEthernet controller model acts as a TLM-2.0 target which receives transactions containing Ethernet frames from the PE model via a target socket. Generic payload extensions are added to show which traffic class the Ethernet frame belongs to. The TTEthernet switch model stores and forwards different traffic class frames using different mechanisms. Since TLM-2.0 of SystemC is mainly for modeling memory-mapped buses, modeling TTEthernet requires some extensions: An Ethernet socket is introduced by deriving from both tagged initiator and target sockets of TLM-2.0 in order to simulate the bidirectional communication link between two ports of TTEthernet devices. Binding and accessing methods of the socket are implemented and new payload type for Ethernet is also added.

3.1.4 Sensor and Actuator Modeling

The cyber components interact with the physical system through sensors and actuators. In our model each sensor/actuator has a *SC_THREAD* process that is responsible for updating the sensing/actuation values. The sensors are modeled as active devices, and the actuators are modeled as passive devices.

As an active device, a sensor will periodically use an IRQ line to inform the PE model to fetch the values through a bus transaction. According to the configuration, the active sensors can use their independent clocks or they can be synchronized with the PE model. On the contrary, the values used by an actuator are fed by the PE model periodically or sporadically.

The interactions between the sensors/actuators and the physical model are simply through shared variables. The pointers to these shared variables are taken by the sensors/actuators. When there is an update, the value will be read/written from/to the physical model by dereferencing the corresponding pointer.

3.2 Software Layer

The software layer comprises the software components with behavior expressed in logical time. Each software component takes the corresponding generated C code from MATLAB/Simulink model to realize its functionality.

All the software components belonging to the same PE are grouped into one task set class which is derived from *sc_module* class. When integrating all the models, the task set will be instantiated and registered to the RTOS model of the corresponding PE, and an off-line defined schedule table for TT activations is also registered to the RTOS model. Each software component is wrapped into a *SC_THREAD* SystemC process as a task which will be scheduled by the RTOS model. Each task has an *sc_event* object. The execution of a task is pending on its own *sc_event* object which will be notified by the scheduler when the task is scheduled to run. The worst case execution time (WCET) of a software component is needed for the off-line TT paradigm scheduling tool and is annotated to the task. Although the execution of a piece of C code will take zero logical execution time, the task will invoke an RTOS API to delay itself for at least the WCET to generate the outputs.

As shown in Fig. 2, all the control software tasks constitute the software layer and are distributed over two ECUs. The interactions between the software and network/platform layers are: the scheduler of the RTOS schedules the tasks and informs a task to run by using *sc_event* notification; the tasks acquire RTOS services, such as inter-node communication, via system calls.

3.3 Physical Layer

In order to integrate the physical layer model with the network/platform layer model in SystemC, the physical model has to provide input/output interfaces through which we can access the variables representing its dynamics. It also should have an interface for simulation time synchronization. A wrapper module can take advantage of these interfaces and integrate them with the network/platform layer.

For instance, in the automotive example shown in Fig. 2, we integrate CarSim into the co-simulation framework to act as the physical layer. CarSim is a commercial parameter-based vehicle dynamics modeling software [2]. CarSim has a program called VehicleSim (VS) solver used to read and write files, calculate dynamics, and communicate with other software. It has an internal mathematical model that pre-

dicts the behavior of vehicles in response to control signals. The solver is in the form of a dynamically linked library (DLL) file with a set of API functions. Integrating CarSim into the co-simulation framework is achieved by a wrapper module that takes charge of synchronization between the SystemC simulation kernel and CarSim VS solver. The solver DLL has a set of time-stepping API functions, and the time of the physical model will be increased with the configured time step by each time-stepping API function call from the wrapper module. The VS solver will solve the differential equations according to the current time and update the internal mathematical model. Different dynamics variables reference to the variables in the internal mathematical model through VS APIs. These dynamics variables are revealed by the wrapper module to the sensors/actuators of the network/platform layer through shared variables.

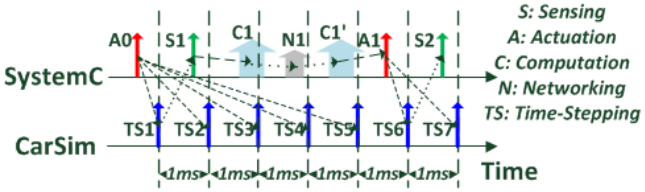

Figure 4: SystemC-CarSim Integration in Time Domain

Due to using the fixed-step solver in CarSim, the interval between two successive mathematical model updates is fixed (1 ms in our example). The wrapper module has to call the CarSim time-stepping function every fixed-step. SystemC uses a discrete event simulator which can process sensing and actuation events at the network/platform layer between an interval. The sensing period should be greater than the simulation fixed-step; otherwise two successive sensing events within a step will acquire the same dynamics variable evaluations. Similarly for the actuation, whose timing depends on the upper layer computation since it passively receives the actuation data, the actuation period should also be greater than the fixed-step. Fig. 4 shows how the cyber part modeled in SystemC is integrated with the physical layer modeled by CarSim in the time domain (the dashed lines represent the data flow). In this example, the control sampling period is 5ms. S1 acquires the dynamics variable values updated by TS1, and the computation and communication (C1 and N1) take some time before the actuation variables are changed by A1. The changed actuation variables affect dynamics variables updating TS6 and TS7, and then another control period begins.

4. RAPID PROTOTYPING DESIGN FLOW

The virtual prototype of a specific system can be generated automatically by using the Embedded Systems Modeling Language (ESMoL) environment. ESMoL is a suite of domain-specific modeling languages, providing a single multi-aspect design environment and a set of tools to facilitate the design of embedded real-time control systems [15]. The rapid prototyping design flow is shown in Fig. 5.

The first four steps belong to using ESMoL to facilitate high-confidence control software design. Step 1 is to specify the control functionality in the MATLAB/Simulink environment and configure/establish the physical dynamics model.

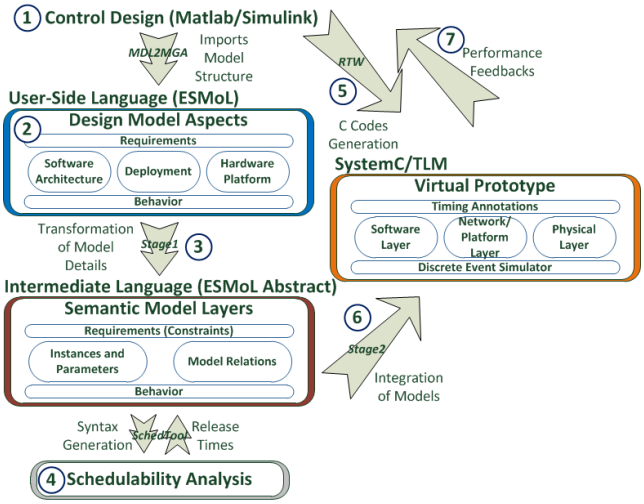

Figure 5: Rapid Prototyping Design Flow Supported by the ESMoL Language and Virtual Prototype

The Simulink model will be imported into the ESMoL automatically to become the functional specification for instances of software components. Step 2 is to specify the non-functional parts of the system in ESMoL which includes: (1) specifying the logical software architecture which captures data dependencies between software component instances independent of their distribution over different nodes; (2) defining hardware platforms hierarchically as nodes with communication ports interconnected by networks; (3) setting up a deployment model by mapping software components to nodes, and data messages to communication ports; (4) establishing a timing model by attaching timing parameter blocks to components and messages. Step 3 translates the ESMoL model into the simpler ESMoL_Abstract model using the *Stage1* interpreter of ESMoL. The model in this intermediate language is flattened and the relationships implied by structures in ESMoL are represented by explicit relation objects in ESMoL_Abstract. Step 4 is to take the scheduling problem specification generated from ESMoL_Abstract model and use a tool of ESMoL called *SchedTool* to solve the scheduling problem. The results are imported back into ESMoL model and written to the appropriate objects. More details of these four steps can be found in [15].

In order to integrate the co-simulation framework with ESMoL, we extend the ESMoL design flow. Step 5 is to generate C code from MATLAB/Simulink model using Real Time Workshop (RTW) toolbox. Step 6 uses *Stage2* interpreter of ESMoL to generate the virtual prototype. For each model of the cyber part, there is a corresponding configuration template which can be parameterized by using Google Ctemplate. The interpreter uses the UDM model navigation APIs to traverse the ESMoL_Abstract model to assemble the C code generated by RTW into tasks and parameterize the configuration templates. The template for a task is organized as follows: in an infinite loop it first waits on its own *sc_event* object; if the task is a receiver of a remote message, it invokes the read message API with corresponding arguments; then it invokes its generated C function to compute

in zero logic execution time; its execution time is enforced by calling the timing annotation API to pass its WCET to the RTOS it belongs to; at last if the task sends a message to the network, it calls the write message API with corresponding arguments; otherwise, it updates the shared memory. All the tasks running on the same node are grouped into one task set class which is derived from *sc_module* class. For each node, the PE, bus, TTEthernet controller, sensors, and actuators are instantiated, connected, and configured in the *sc_main()* function which is the top level of a SystemC program. The configuration files for the model instances are generated according to the specified attributes in the ESMoL model, such as the schedule tables for the RTOSes. The task set class of the node is also instantiated and registered to the RTOS of the PE model. TTEthernet switches are also instantiated and configured. According to the topology defined in the hardware model of ESMoL, all the nodes are connected. The physical model is instantiated and configured. All the pointers to the shared variables of the physical model are passed to the corresponding sensors/actuators. Finally, the co-simulation results of the holistic system provide performance feedback for engineers to revise their designs, which is the Step 7.

5. VALIDATION AND EVALUATION

The proposed framework is implemented in C++/SystemC/TLM with about 10,000 lines of code. In this section, we first validate the TT communication model by comparing the delays obtained from the network model and a real implementation of TTEthernet. Then, we use an adaptive cruise controller (ACC) case study to validate the co-simulation framework by comparing with the results obtained from a hardware-in-the-loop (HIL) automotive simulator. We also explore the design space of the communication systems to help designers make design decisions. In terms of measuring control performance of ACC, we use the tracking ability of the controller as the performance metric.

5.1 TT Communication Validation

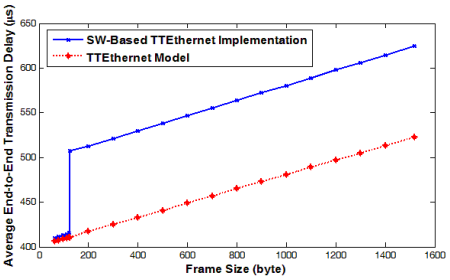

Figure 6: Average End-to-End Transmission Delay of Different Frame Sizes

We validate the TTEthernet model by comparing the average end-to-end transmission delay of the TT traffic in a software-based implementation of TTEthernet and the model under the same experimental scenario. The method of measuring end-to-end transmission delay of software-based TTEthernet implementation is presented in [1], which utilizes two ports on a single box. We measure the average delay for different TT frame sizes under full link utilization of BE traffic. Fig. 6 shows the results. From the results we can observe there is a latency gap (90μs) between frame size of 123 and 124 bytes which is actually caused by the BE-device driver configuration according to [1]. This gap is due to measurement approach limits and will not appear when using the TT communication.

5.2 ACC Case Study

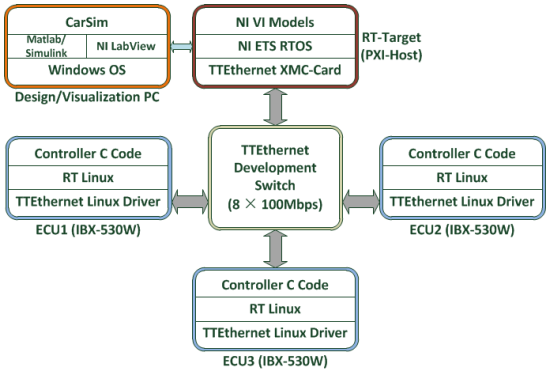

Figure 7: System Architecture of HIL Automotive Simulator

In order to test the automotive control system in a more realistic way, we use a HIL automotive simulator. The architecture of the HIL simulator is shown in Fig. 7. The physical dynamics modeled in CarSim is deployed on an RT-Target in a sense that it acts as the real automotive vehicle. The RT-target is also integrated with a TTTech PCIe-XMC card which enables the seamless integration and communication with ECUs on the time-triggered network. The network/platform layer of the HIL simulator is composed of three ECUs which are connected to an 8-port 100Mbps TTEthernet development switch from TTTech. Each ECU is an IBX-530W box with an Intel Atom processor running a RT-Linux operating system. Each ECU is integrated with a TTEthernet Linux driver which is a software-based implementation of TTEthernet protocol to enable communication with other end systems in a TTEthernet network. Automotive control software is distributed over the ECUs and the tasks execute in the kernel space of RT-Linux which can utilize the synchronized time base of the TTEthernet communication.

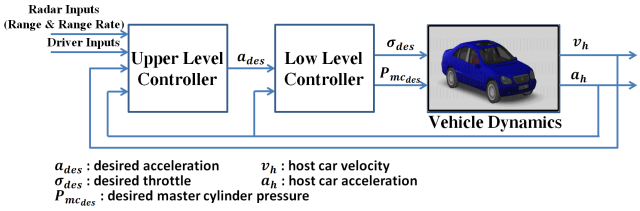

Figure 8: Adaptive Cruise Control System [5]

The control algorithm of ACC is designed in MATLAB/-Simulink. Fig. 8 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. 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. The main objective of the low controller is two-fold: first, using the desired acceleration command from the upper level controller, the lower level controller determines whether to apply braking control or throttle control; second, the required control command is applied to the vehicle in order to achieve the desired acceleration. Details about the ACC can be found in [5].

The model is imported into ESMoL environment. The four different aspects of the design in ESMoL are shown in Fig. 9. The topology of the network/platform layer is based on the HIL simulator which is shown in Fig. 9 (a). Fig. 9 (b) shows the software logical architecture that depicts the logical interconnections of four ACC tasks, which are *InstrClstrSens*, *UpperLevelController*, *LowLevelController*, and *InstrClstrAct*, and two sensing/actuation tasks, which are *InputHandler* and *OutputHandler*. The deployment of the ACC control software is shown in 9 (c) in which the dashed arrows represent assignment of tasks to their respective ECUs and solid lines represent assignment of message instances to communication channels on the ECU. Finally, the timing and execution model for tasks and message transfers of the ACC control system is shown in 9 (d).

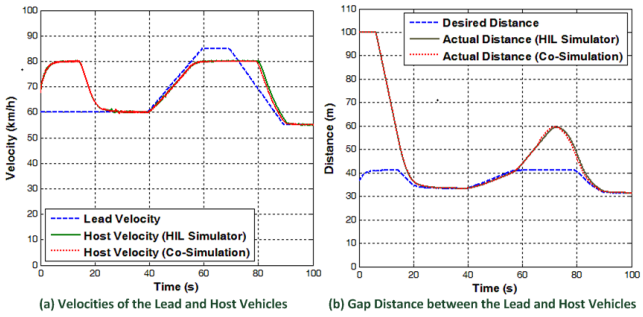

Figure 10: Velocities and Gap Distance from HIL Simulator and Co-Simulation Framework

The sampling period on the HIL simulator is 10ms which is limited by the software-based TTEthernet implementation. In this case study, the velocity of the leading vehicle starts at an initial value of 60km/h. The host vehicle radar range is 100m. The initial global longitudinal positions of the leading vehicle and the host vehicle are 130m and 0m respectively, which means the host vehicle radar is initially out of range. The host vehicle starts at an initial velocity of 65km/h with a driver set target velocity of 80km/h. The expected driving behavior of the host vehicle should be: (1) before the host vehicle detects the leading one, it will speed up to and maintain at most at 80km/h; (2) when the radar detects a slower leading vehicle, the ACC will control the distance between the two vehicles to a driver set time gap, and the desired gap distance is attained when two vehicles travel at the same velocity; (3) when the leading vehicle begins to speed up, the velocity of the host vehicle will also increase in order to achieve a desired velocity (the host maintains at most at 80km/h, even when the leading vehicle exceeds this speed); (4) when the leading vehicle slows down, the host also starts to decrease its velocity in order to maintain the desired space between the vehicles.

The results of the designed ACC running on the HIL simulator and the proposed framework are given in Fig. 10

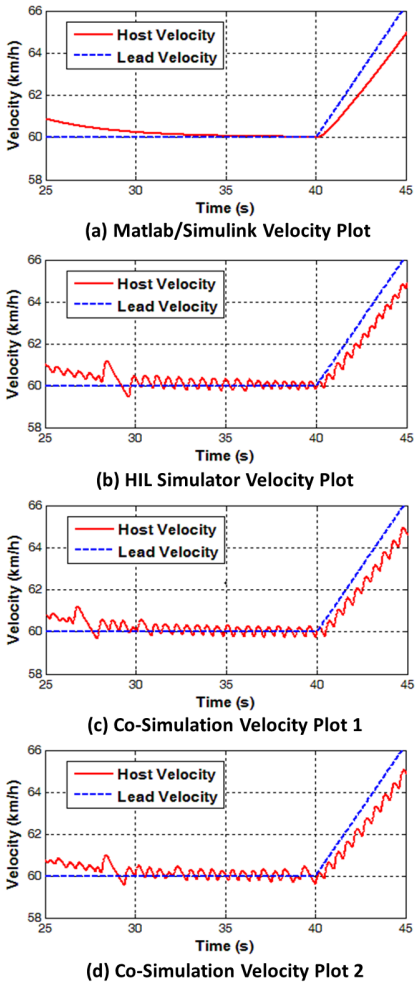

Figure 11: Comparison of Results from MATLAB/Simulink, HIL Simulator, and Co-Simulation Framework

(since the results are very similar, most of the curves are overlapped). We zoom in on the velocity plots between 25s and 45s, and also use the result from MATLAB/Simulink as a reference which is shown in Fig. 11 (a). From Fig. 11 (b), we can observe the velocity of the HIL simulator suffers from some oscillations which have a highest value about 0.7km/h $\approx$ 0.2m/s. The co-simulation results also shows these oscillations as shown in Fig. 11 (c) and (d). Due to different randomly assigned clock drifts, the results cannot be exactly the same; yet, the figures show very similar results, especially compared to the result from MATLAB/Simulink. For example, similar but not identical results are given in Fig. 11 (c) and (d) (e.g. the peak of the oscillations is shifted more towards 30s in (d)).

Our HIL simulator has an implementation limitation that does not allow the computation on the RT-Target to be synchronized with the TTEthernet communication. We can conjecture that the oscillations are mainly due to the delays caused by the non-synchronized computation with the TT communication on the RT-Target. This conjecture can be proved by comparing the co-simulation result of the synchronized setting with the one of the non-synchronized setting of

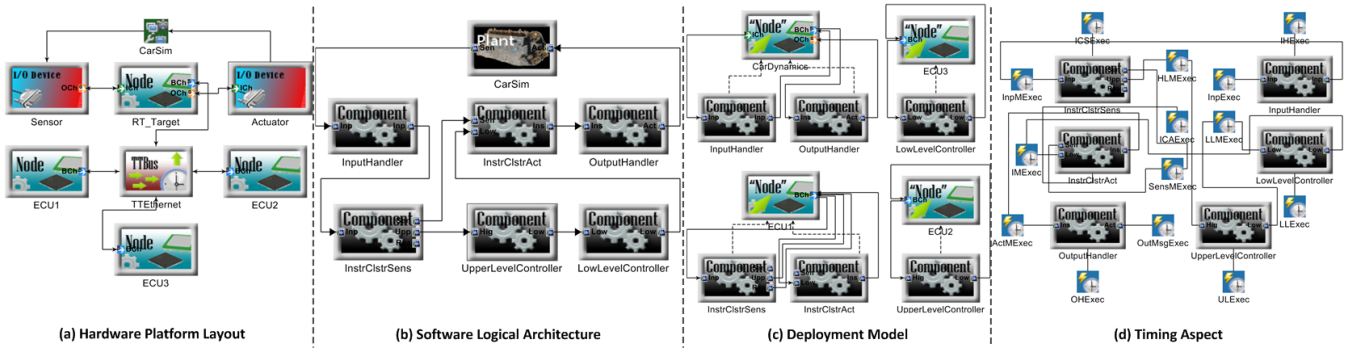

Figure 9: ESMoL Design Models of ACC

the RT-Target. Fig. 12 shows the co-simulation result of the synchronized setting of the RT-Target, from which we can observe the oscillations are apparently reduced (the highest amplitude is about 0.09km/h ≈ 0.025m/s).

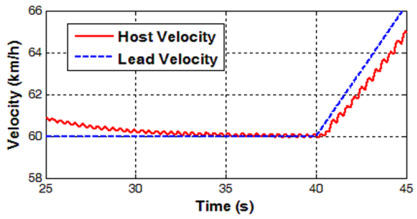

Figure 12: Co-Simulation Velocity Plot by Using Synchronized Setting of RT-Target

5.3 Design Space Exploration

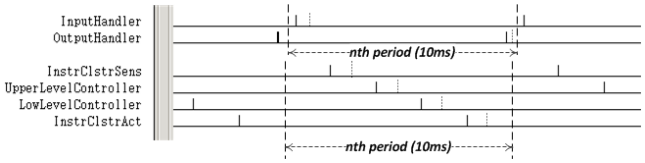

Figure 13: Timing Diagram of ACC Tasks

The ACC software execution on the HIL simulator is not computationally intense. The timing diagram generated by the co-simulation framework (Fig. 13) shows that every task meets its deadline which is represented by the dotted line (due to implementation limitations, on RT-Target the computation is not synchronized with the communication). If the physical layer is not included, like the tools introduced in [18] and [22], the system is perfectly designed. However, when the car dynamics model begins to execute in this simulation, the oscillations of the vehicle velocity can be observed. In order to improve performance, we can increase the sampling rate. The computation of the system is negligible, but the communication system that uses the software-based implementation of TTEthernet becomes an obstacle which limits the fastest reasonable sampling period to 10ms. In order to reduce the sampling period, we need to consider other design alternatives in the design space.

By employing the hardware-based implementation of TTEthernet which has a 1Gbits/s bandwidth and more precise

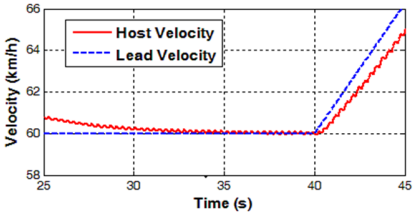

Figure 14: Co-Simulation Velocity Plot of 5ms Sampling Period

clock synchronization, we can achieve the sampling period reduction. A 5ms sampling period is used in the new design and gains in the ACC controller are tuned. The zoomed-in velocity result of the co-simulation is given in Fig. 14, from which we can see the performance of the ACC is much better than the previous one: the oscillations have a highest amplitude about 0.05km/h ≈ 0.015m/s.

To illustrate the efficiency of this framework, we provide the CPU time for 100s simulation time of the ACC under a machine with dual cores of 3.40GHz and 8GB memory. For the first case (10ms sampling period), the consumed CPU time is about 102s, and for the second case (5ms sampling period), the consumed CPU time is about 194s.

6. CONCLUSION

In this paper, we propose a co-simulation framework that can facilitate TT CPS design. A simplified view of designing CPS is to consider three design layers, which include the physical layer, the network/platform layer, and the software layer. The proposed framework contains models from each of the three CPS design layers. SystemC is used to model the cyber part and simulators of physical systems are used to model the physical part of a CPS. Since the network/platform layer is the intermediate layer between the other two layers, it plays an important role in CPS integration and becomes the backbone of the framework models. The models can be configured and integrated to become a virtual prototype of a TT CPS to provide realistic feedback at early design stages. The framework is also integrated with a model-based design tool called ESMoL to enable rapid prototyping. The TT communication model is validated against a real implementation in term of its end-to-end transmission delay. In order to evaluate the framework, we focus on the automotive domain and use CarSim to model the phys-

ical layer. An ACC case study is provided to illustrate the framework. The case study shows that the co-simulation framework provides similar results to a HIL simulator with good efficiency.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (CNS-1035655). The authors would like to thank the anonymous reviewers for their comments and suggestions which greatly help us improve the quality of the paper.

7. REFERENCES

[1] F. Bartols, T. Steinbach, F. Korf, and T. C. Schmidt. Performance analysis of time-triggered ether-networks using off-the-shelf-components. In *Proceedings of the 2011 14th IEEE International Symposium on Object/Component/Service- Oriented Real-Time Distributed Computing Workshops*, ISORCW '11, pages 49–56, 2011.

[2] CarSim. http://www.carsim.com/.

[3] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3):16–30, Jun 2003.

[4] U. Drolia, Z. Wang, Y. Pant, and R. Mangharam. Autoplug: An automotive test-bed for electronic controller unit testing and verification. In *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*, pages 1187 –1192, oct. 2011.

[5] E. Eyisi, Z. Zhang, X. Koutsoukos, J. Porter, G. Karsai, and J. Sztipanovits. Model-based control design and integration of cyber-physical systems: An adaptive cruise control case study. *Journal of Control Science and Engineering*, 2013.

[6] L. Gheorghe, F. Bouchhima, G. Nicolescu, and H. Boucheneb. Formal definitions of simulation interfaces in a continuous/discrete co-simulation tool. In *IEEE International Workshop on Rapid System Prototyping*, pages 186–192, 2006.

[7] IEEE. IEEE Standard SystemC Language Reference Manual, 2011.

[8] M. Ishikawa, D. J. McCune, G. Saikalis, and S. Oho. Cpu model-based hardware/software co-design, co-simulation and analysis technology for real-time embedded control systems. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, RTAS '07, pages 3–11, 2007.

[9] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[10] M. Krause, O. Bringmann, A. Hergenhan, G. Tabanoglu, and W. Rosentiel. Timing simulation of interconnected autosar software-components. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 474–479, 2007.

[11] H. Lonn and J. Axelsson. A comparison of fixed-priority and static cyclic scheduling for distributed automotive control applications. In *ECRTS*, pages 142–149, 1999.

[12] P. L. Marrec, C. A. Valderrama, F. Hessel, A. A. Jerraya, M. Attia, and O. Cayrol. Hardware, software and mechanical cosimulation for automotive applications. In *Proceedings of the Ninth IEEE International Workshop on Rapid System Prototyping*, RSP '98, pages 202–, 1998.

[13] W. Müller, M. Becker, A. Elfeky, and A. DiPasquale. Virtual prototyping of cyber-physical systems. In *ASP-DAC*, pages 219–226, 2012.

[14] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems. *Proceedings of the IEEE*, 93(6):1204–1223, 2005.

[15] J. Porter, G. Hemingway, H. Nine, C. vanBuskirk, N. Kottenstette, G. Karsai, and J. Sztipanovits. The esmol language and tools for high-confi dence distributed control systems design. part 1: Language, framework, and analysis. Technical report, Vanderbilt University, Sep 2010.

[16] J. M. Rushby. Bus architectures for safety-critical embedded systems. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 306–323, 2001.

[17] SAE Standard AS 6802. Time-Triggered Ethernet, 2011.

[18] M. Streubühr, M. Jäntsch, C. Haubelt, and J. Teich. From Model-based Design to Virtual Prototypes for Automotive Applications. In *Proceedings of the Embedded World Conference*, pages 1–10, Nuremberg, Germany, Mar. 2009.

[19] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and S. Wang. Toward a science of Cyber-Physical system integration. *Proceedings of the IEEE*, 100(1):29–44, Jan. 2012.

[20] M. Verhoef, P. Visser, J. Hooman, and J. Broenink. Co-simulation of distributed embedded real-time control systems. In *Proceedings of the 6th international conference on Integrated formal methods*, IFM'07, pages 639–658, 2007.

[21] H. Zabel, W. Müller, and A. Gerstlauer. Accurate RTOS modeling and analysis with SystemC. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware-dependent Software*, chapter 9, pages 233–260. Springer Netherlands, 2009.

[22] M. Zeller, G. Weiss, D. Eilers, and R. Knorr. Co-simulation of self-adaptive automotive embedded systems. In *Proceedings of the 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, EUC '10, pages 73–80, 2010.