Model-Based Development and Verification of Control Software for Electric Vehicles

Dip Goswami[1], Martin Lukasiewycz[2], Matthias Kauer[2], Sebastian Steinhorst[2],
Alejandro Masrur[1], Samarjit Chakraborty[1] and S. Ramesh[3]
[1]Institute for Real-Time Computer Systems, TU Munich, Germany
[2]TUM CREATE, Singapore
[3]General Motors Corp., USA

ABSTRACT

Most innovations in the automotive domain are realized by electronics and software. Modern cars have up to 100 Electronic Control Units (ECUs) that implement a variety of control applications in a distributed fashion. The tasks are mapped onto different ECUs, communicating via a heterogeneous network, comprising communication buses like CAN, FlexRay, and Ethernet. For electric vehicles, software functions play an essential role, replacing hydraulic and mechanic control systems. While model-based software development and verification are already used extensively in the automotive domain, their importance significantly increases in electric vehicles as safety-critical functions might no longer rely on mechanical (fall-back) solutions. The need for reducing costs, size, and weight in electric vehicles has also resulted in a considerable interest in topics such as the consolidation of ECUs as well as *efficient* implementation of control software. In this paper we discuss two broad issues related to model-based software development and verification in electric vehicles. The first is concerned with how to ensure that model-level semantics are preserved in an implementation, which has important implications on the verification and certification of control software. The second issue is related to techniques for reducing the computational and communication demands of distributed automotive control algorithms. For both these topics we provide a broad introduction to the problem followed by a discussion on state-of-the-art techniques.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies*

General Terms

Theory, Design, Performance, Verification

DAC'13, May 29 - June 07 2013, Austin, TX, USA

Keywords

Electric vehicles, control systems, model-based design, control/architecture co-design

1. INTRODUCTION

The volume of functionality implemented in software has been steadily increasing in the automotive domain over the past decades. In electric vehicles the importance of software functions will even further increase as mechanic and hydraulic systems will be replaced by software-based control. This will reduce the overall weight of the car and in turn increase the driving range which is of utmost importance in electric vehicles, compensating the limitations of state-of-the-art batteries. At the same time, software-based control may increase efficiency, for instance, for regenerative breaking, etc.

The shift towards software-based control increases the computational demand significantly. Current design methods in the automotive domain follow a federated architecture approach, where each function is implemented on a separate ECU. While this enables the Original Equipment Manufacturer (OEM) to outsource the development to various suppliers – with the role of the OEM being that of a function integrator – it increases the number of ECUs in the car as well as the cabling harness. In the case of electric vehicles this becomes a major problem, not only because of the complexity and the distributed nature of the resulting architecture, but also due to the additional weight of the ECUs and the cables. As a result, there is an increasing push to move towards integrated architectures, where multiple software functions are integrated onto a single ECU. An application in this architecture is distributed into different tasks, each of which runs on a different ECU, communicating via shared buses.

While integrated architectures combine many advantages, they also bring along many challenges due to the higher complexity in the software implementation and integration. As a result, the need for appropriate verification and certification techniques becomes more prominent. Whereas *model-based* software development and verification are already practiced in the automotive domain, their importance is significantly higher in electric vehicles, primarily due to the lack of mechanical fall-back solutions as, for example, in drive-by-wire implementations. In this paper we discuss two important issues in this context. The first is concerned with how to ensure that model-level semantics are preserved in an implementation, particularly for complex distributed

architectures with multiple ECUs communicating via a heterogeneous network using different bus protocols. This has important implications on the verification and certification when using model-based design techniques. The second issue is concerned with *efficient* implementations of control software, such that their computational and communication demands are reduced.

The paper is organized as follows. We first present an overview of how to model feedback control software and the implementation platform in Section 2 followed by an introduction of embedded control in Section 3. This is in turn followed by a discussion of how an implementation may be synthesized from high-level control models (in Section 4). In particular, we show how schedules in automotive-specific bus protocols like CAN or FlexRay might be synthesized in order to meet control performance requirements. The second part of the paper shows how control performance goals may be realized even when certain feedback signals are dropped (in Section 5). This reduces both computational and communication demands, and therefore results in cost-effective implementations. Towards this, we first show how to quantify the bounds on the number of feedback signal drops for given control performance requirements. We then show how such bounds may be verified for a given architecture, i.e., whether the number of signals dropped by an architecture is within the limits of what may be tolerated by the application.

2. SYSTEM MODEL

2.1 Feedback control model

Often the system model is considered to be linear time-invariant (LTI) in the following form

$$\begin{aligned} \dot{x}(t) &= A_c x(t) + B_c u(t), \\ y(t) &= C_c x(t), \end{aligned} \tag{1}$$

where $x(t)$ is the $n \times 1$ vector for *state variables*, $u(t)$ is the *control input* to the system and $y(t)$ is the *output*. A_c is a $n \times n$ system matrix, B_c and C_c are input and output matrices of appropriate dimension. Subsequently, the continuous-time system is sampled at a constant sampling interval h with zero-order-hold (ZOH). The resultant system is a discrete-time system of the form

$$\begin{aligned} x[k+1] &= Ax[k] + Bu[k], \\ y[k] &= Cx[k], \end{aligned} \tag{2}$$

where

$$A = e^{A_c h}, B = \int_0^h (e^{A_c t} dt) \cdot B_c, C = C_c. \tag{3}$$

Such feedback control systems have two physical components: *actuators* (to apply the input $u[k]$ to the system) and *sensors* (to read states $x[k]$ from the system). In a distributed architecture such as those in the automotive domain, the actuators and the sensors are often spatially distributed and connected to different ECUs which communicate via a bus system. A feedback *controller* is an algorithm to compute $u[k]$ as a function of the states $x[k]$ or output $y[k]$ (feedback signals) such that $x[k]$ or $y[k]$ behave according to the specified control goals – we assume all states $x[k]$ to be *measurable*.

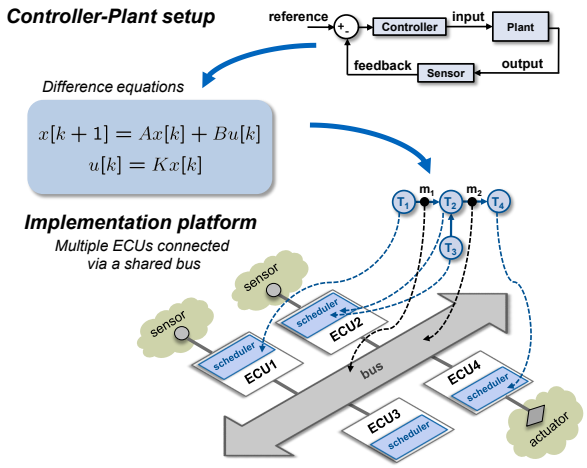

Figure 1: Distributed implementation platform.

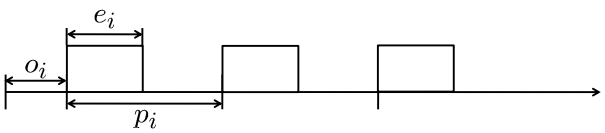

Figure 2: Periodic task model.

In this work we use full-state feedback controllers of the form shown in the following equation:

$$u[k] = Kx[k], \tag{4}$$

where K are $1 \times n$ *state-feedback gains*. The controller design essentially boils down to choosing K such that the system (i) is *stable*, i.e., the closed-loop system matrix $(A+BK)$ has all the *eigenvalues* within the *unit circle*, and (ii) meets performance requirements in steady-state and transient phases, e.g., minimization of the tracking error and the settling time of the control system.

2.2 Implementation platform

Let us consider an automotive in-vehicle network where several ECUs are connected via a bus as depicted in Fig. 1. The software implementation of distributed control applications running on such architectures is typically realized by model-based software development. This involves automatic code generation from high-level control models such as MATLAB/Simulink, along with the bus driver stack and the operating system (OS). For this, the high-level control models are partitioned into several software tasks that need to be mapped onto different ECUs. Subsequently, the generated code, e.g., C-code, is cross-compiled for the target hardware and flashed to the dedicated ECUs.

Automotive ECU. In general, an ECU consists of (i) a host microcontroller running the OS which schedules and executes application tasks T_i and system tasks (e.g., for communication purposes, etc.), (ii) a communication controller that implements the communication protocol (e.g., FlexRay, CAN) and (iii) bus drivers that realize the conversion of the logical bit stream into physical signals propagated on the communication bus.

The task model depends on the scheduling policy implemented by the OS. The scheduling policy can either be pre-

emptive (e.g., OSEK) or non-preemptive (e.g., eCos). Moreover, the ECUs can either be synchronous (e.g., FlexRay) or asynchronous (e.g., CAN) to the bus schedules.

The control tasks are often periodic and time-triggered. In such cases, the OS provides a task dispatcher which allows cyclic task execution. Let a dispatch event for a task T_i (see Fig. 2) be defined by the tuple $T_i = \{o_i, p_i, e_i\}$:

- The task offset o_i specifies the duration from start time to the first task invocation of T_i.
- The task period p_i specifies the time between two consecutive activations of T_i.
- The worst-case execution time e_i specifies T_i's maximum execution time.

Hence, the k-th instance of task T_i is triggered at

$$t_i^k = o_i + kp_i,\ k \in \mathbb{N}_0. \tag{5}$$

The scheduler of the OS processes all tasks according to a dispatch table in a cyclic manner where the length of the dispatch table is determined by the *hyperperiod* $H = \underset{\forall i}{lcm}(p_i)$. Consequently, all task invocation times can be computed offline and stored in the dispatch table. Further, the k-th instance of T_i finishes at latest at

$$\tilde{t}_i^k = o_i + kp_i + e_i,\ k \in \mathbb{N}_0. \tag{6}$$

Automotive Bus. Today's cars usually consist of a wide range of different bus systems connected via gateways. Local Interconnected Network (LIN), Controller Area Network (CAN), FlexRay, Media Oriented Systems Transport (MOST) and Ethernet are some of the bus protocols commonly used. Moreover, different bus systems are used in different functional domains depending on the requirements on bandwidth and temporal behavior. For example, the power-train and the chassis domain often use FlexRay due to their stringent timing constraints. On the other hand, MOST is suitable for the infotainment domain. Moreover, bus protocols can be purely time-driven, event-driven and hybrid in nature. In the following we illustrate the nature of a typical automotive bus system using FlexRay as an example.

FlexRay is a hybrid communication protocol which is organized as a sequence of 64 bus cycles that are periodically repeated [1]. Each cycle is of fixed duration T_{bus} and consists of a static and a dynamic segment. The communication is time-driven in nature on the static segment whereas communication over the dynamic segment is event-driven. Messages m_i are transmitted on the FlexRay bus either on the static or the dynamic segment according to their predefined bus schedules. We denote a bus schedule by the tuple $\Theta_i = \{S_i, B_i, R_i\}$ (Fig. 3):

- The communication slot S_i determines the time window in which a message may be transmitted. $S_1 = 3$ for message m_1 in Fig. 3.
- The base cycle $B_i \in \{0, 1, ..., 63\}$ specifies the first cycle during which S_i is available. For instance, in Fig. 3, m_2 is assigned $B_2 = 1$ since the first cycle in which $S_2 = 6$ is available is *cycle 1*.
- The repetition rate $R_i \in \{1, 2, 4, 8, 16, 32, 64\}$ determines the number of cycles between two consecutive transmissions. For example, m_1 is assigned repetition rate $R_1 = 2$ and hence S_1 is available in every second cycle. Similarly, m_4 is assigned $R_4 = 4$, therefore S_4 is available in every fourth cycle. Clearly, $B_i < R_i$ must hold.

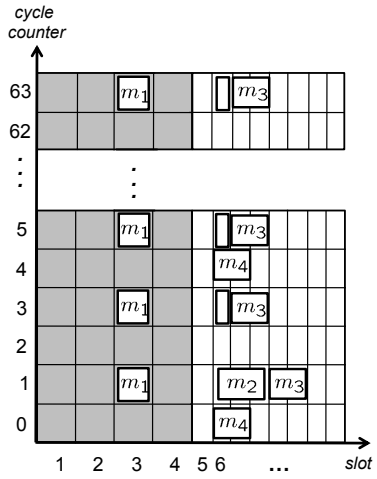

Figure 3: FlexRay Schedule: Gray boxes indicate static segment slots.

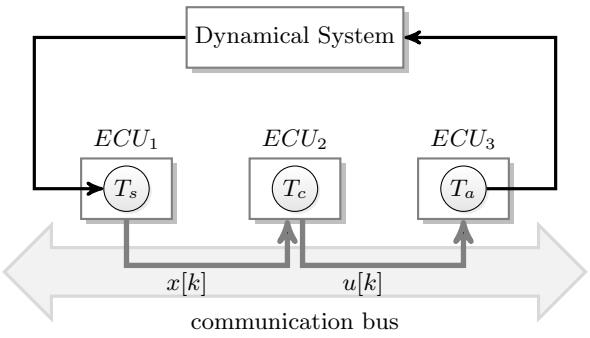

Figure 4: Distributed control application.

3. EMBEDDED CONTROL

We analyze in this section how to implement control software based on the models discussed above. In general, a control application $\mathcal{C}_l$ is *partitioned* into three tasks (see Fig. 4): (i) the sensor task T_s – reading $x[k]$ from sensors, (ii) the controller task T_c – computing $u[k]$ using (4) and, (iii) the actuator task T_a – applying $u[k]$ to the dynamic system. In an embedded control implementation T_c and T_a are typically time-triggered periodic tasks. Often, the period of T_c and T_a is equal to the sampling period h of the corresponding control application. The triggering paradigm of the sensor task T_s depends on the sensing mechanism. Generally, the sensor tasks have to constantly monitor some states of the control plant and are therefore *interrupt-driven*. T_s can be triggered many times within a sampling period with a negligible execution time. Further, the application tasks are *mapped* onto different ECUs and communicated over a bus. Fig. 4 shows an example of task mapping where T_s, T_c and T_a are mapped onto separate ECUs. Here, the system states

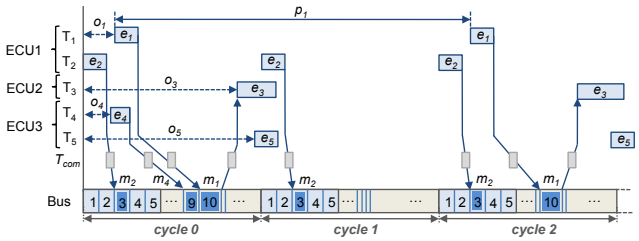

Figure 5: An example of control implementation.

$x[k]$, measured by T_s, are sent over the bus to T_c, which then computes $u[k]$ and sends it to T_a.

3.1 Implementation semantics

Generalizing the above description, each control application can be partitioned into a set of n control-related tasks T_i, $i \in \{1, ..., n\}$. Furthermore, depending on the task mapping, schedules Θ_i are assigned to messages m_j on the communication bus where $j \in \{1, ..., \kappa\}$. Thus, a control application $\mathcal{C}_l$ has a $(n + \kappa)$-tuple of platform parameters $\Phi_l = \{T_i, \Theta_i\}$. For example, let a control application $\mathcal{C}_l$ be partitioned into $n = 5$ tasks that trigger $\kappa = 3$ messages as shown in Fig. 5 – note that m_1, m_2 and m_4 are the only messages sent by T_1, T_2 and T_4 respectively. Then the corresponding platform parameters are denoted by the 8-tuple:

$$\Phi_l = \{T_1, T_2, T_3, T_4, T_5, \Theta_1, \Theta_2, \Theta_4\}.$$

Recall that $T_i = \{o_i, p_i, e_i\}$, and $\Theta_i = \{S_i, B_i, R_i\}$ are also tuples themselves.

3.2 Design questions

In this context, the pertinent design questions that have gained significant attention from both academia and industry are the following:

- **How to design Φ_l and $\mathcal{C}_l$ such that all the high-level control goals are met?** Traditionally, the design of Φ_l and $\mathcal{C}_l$ are performed in two isolated phases. That is, $\mathcal{C}_l$ is designed assuming a possible worst-case design of Φ_l and similarly, Φ_l is designed by translating the performance requirements of $\mathcal{C}_l$ into overly conservative timing constraints. Such isolated design phases introduce pessimism at various layers making the overall design very conservative. Hence, such design flows can potentially become a major bottleneck for electric cars in particular, due to design parameters like weight of cabling, number of ECUs, battery lifetime, etc. that might be worsened in this case. In contrast, the design of Φ_l and $\mathcal{C}_l$ can mutually be directed by each other. Hence, a part of Φ_l is first designed. Next, a part of $\mathcal{C}_l$ is designed exploiting/utilizing the information on the already designed part of Φ_l and so on. In recent literature [18, 16, 19, 17], it is shown that such an interactive design approach opens up scope for significant design optimization. We illustrate this joint design approach with an example in Section 4.

- **How to improve the overall design efficiency in terms of resources and performance?** In such platforms, a control performance is achieved at the cost of certain computational and communication resources. An efficient implementation consumes lesser resource towards achieving the same control performance. Hence, an interesting approach is to consider the trade-off between control performance and *quality of resource*. For example, consider a distributed implementation such as the one shown in Fig. 4. With a purely time-driven communication system, it is possible to assure a very short delay in the feedback loop and achieve very good performance. Thus, a high resource quality (because of the time-driven nature in this case) provides better performance. On the other hand, with an event-driven communication at a lower priority level, the control loop occasionally suffers a long delay and jitter which degrades the control performance.

  In a flurry of recent works [2, 20, 11, 12, 10], the control-related messages are assigned a lower priority communication schedule (lower quality resource). In such scenarios, the feedback delay occasionally becomes very large while *mostly* not exceeding a threshold d_{th}. When the delay exceeds the threshold d_{th}, the feedback signal is either *dropped* or overwritten. These cases are often considered to be "deadline misses" by the control message. Allowing such deadline misses in the design process makes it less stringent compared to the case where deadlines need to be met in all occasions. Thus, it is possible to reduce the amount of pessimism otherwise introduced by a worst-case based design. This approach raises further two research questions: How frequently should such dropping be allowed by the control application and how to formally validate that the implementation platform never violates that constraint? We illustrate these aspects in Section 5.

4. CONTROL/PLATFORM CO-DESIGN

As explained above, the design phase in automotive control systems is traditionally separated from the development phase. This makes it difficult to provide guarantees for high-level control requirements. To overcome this problem and, at the same time, achieve a resource-efficient system, both control and platform-related parameters (such as the schedules on ECUs and communication bus) have to be co-designed.

To illustrate this technique, we consider in this section a deterministic time-triggered system where the sensor-to-actuator delay does not suffer from any jitter. We first present a scheduling approach that determines feasible schedules satisfying all constraints or deadlines. Second, we show how the control parameters can be optimized iteratively, using the scheduling approach for each iteration.

4.1 Scheduling Approach

Based on control requirements, a deadline on the delay τ can be deduced for each application. Here, each path π of data-dependent tasks has to satisfy this deadline.

In time-triggered systems, a schedule is defined by the offset o_i of each task T_i for a globally synchronized time. Determining a feasible schedule that satisfies all deadlines might become a complex problem that makes a manual design impractical. For this reason, it can be converted to a mathematical programming problem such as Integer Linear Programming (ILP). In the following, one such solution is outlined. Here, the variables to be optimized are the offsets

which are bounded by the period of each task such that $\forall T_i$:

$$0 \leq o_i \leq p_i. \tag{7}$$

For each pair of tasks that is running on the same resource, only a single task can be executed at a time.

$\forall T_i, T_j$(on the same resource) :

$$(o_i + e_i + k \cdot p_i)\%H \leq o_j + \tilde{k} \cdot p_j \vee \tag{8}$$
$$(o_j + e_j + \tilde{k} \cdot p_j)\%H \leq o_i + k \cdot p_i, \tag{9}$$

where $k = \{0, .., \frac{H}{p_i} - 1\}$ and $\tilde{k} = \{0, .., \frac{H}{p_j} - 1\}$. Here, we assume a non-preemptive scheduler which might be extended to preemptive schedulers as proposed in [14].

Eq. (8) and (9) ensure that no instances/jobs from tasks that are running on the same resource preempt each other. In this context, all jobs k and $\tilde{k}$ in the hyperperiod H are considered.

To determine the offsets o_i of communication tasks sending messages via a bus, the specific protocol has to be considered. An efficient formulation for the FlexRay bus is presented in [14]. Given the offsets of all tasks and messages, the waiting time w_{T_i,T_j} between each pair of data-dependent tasks T_i and T_j can be determined.

$\forall T_i, T_j$(with data-dependencies) :

$$0 \leq w_{T_i,T_j} \leq p_i \tag{10}$$
$$w_{T_i,T_j} = (o_j - (o_i + e_i) + H)\%H \tag{11}$$

Here, the waiting time is always less than the period of the current application. Finally, the end-to-end delay of each application (and path within the application) has to be less than the specified constraint or deadline.

$\forall \tau_X$(deadline), π(paths of tasks) :

$$\sum_{T_i \in \pi} + \sum_{T_i, T_j \in \pi} w_{T_i,T_j} \leq \tau_X \tag{12}$$

Eq. (7) to (12) have to be linearized appropriately to enable an optimization with an ILP. This linearization requires the introduction of additional constraints and variables [14].

4.2 Optimization

The optimization of control parameters is performed iteratively as described in the following. For each iteration, the period p_i for each T_i is chosen from a predefined set of feasible periods. In the model presented above, the periods within one application are considered to be equal. However, this assumption might also be relaxed to allow different task periods within a single control application. Given all periods, the controller gains K can be determined under the assumption that the sensor-to-actuator delay constraint equals the period.

For this given configuration, a scheduling approach as discussed above has to be performed to determine a feasible schedule. If a feasible schedule exists, the performance of the current implementation can be determined using, for instance, the quadratic performance function for each control application:

$$J(h, \tau) = \sum_{k=0}^{N} \int_{kh}^{(k+1)h} [u(t)^2 + x(t)'x(t)]dt, \tag{13}$$

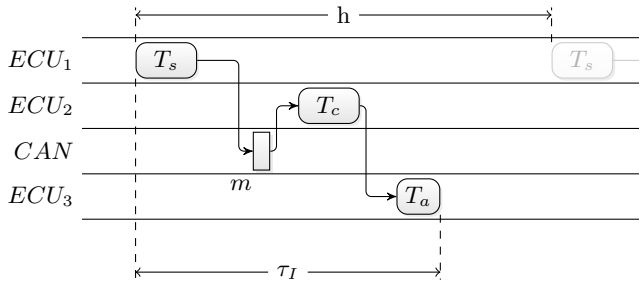

Figure 6: Timing requirements for the control messages: Case I, $\tau_I < h$.

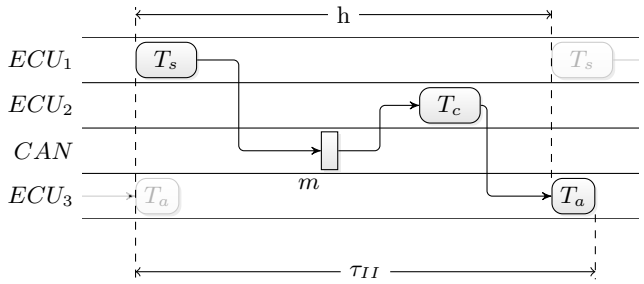

Figure 7: Timing requirements for the control messages: Case II, $\tau_{II} \approx h$.

where N is the total number of samples under consideration. Additional design objectives might be the utilization of ECUs or buses which in turn might find implementations with more economical hardware still satisfying the defined control quality constraints.

The optimization is performed iteratively such that the system is gradually improved. In case the design space (the set of available periods for all applications) is small, an exhaustive search can be performed as presented in [9]. If the design space becomes larger, an exhaustive search becomes impractical and randomized search algorithms like simulated annealing might be applied.

5. FEEDBACK DELAY AND SIGNAL DROP

In this section we are concerned with the second research question stated above. Using information from the control models, we relax some of the typical conservative assumptions that are usually made in the design and development process. In particular, we show that is possible to meet performance requirements even if some control-related messages are lost in the communication bus.

In general, the feedback signal has to go through a series of executions, i.e., $T_s \rightarrow T_c \rightarrow Bus \rightarrow T_a$ in a feedback loop. The delay τ is the time required to complete the entire execution sequence which highly depends on Φ_l. Thus, the delay depends on the triggering patterns of tasks T_s, T_c and T_a as well as the message schedules on the bus. There are two obvious possibilities (Fig. 6 and Fig. 7).

Case I: In this case, the triggering pattern of tasks T_s, T_c and T_a is such that the offset between T_s and T_a is less than one sampling interval. Tasks T_s, T_c and T_a are triggered periodically with period h. The entire series of executions

$T_s \rightarrow T_c \rightarrow Bus \rightarrow T_a$ is mostly expected to be finished by $\tau_I < h$ (see Fig. 6). Therefore, the *deadline* τ_I for the feedback signal is lesser than one sampling interval with such Φ_l. When $\tau_I << h$, the delay is assumed to be *zero* and the control law (4) is therefore *realizable* when the feedback loop meets deadline τ_I. This case is more difficult to analyze but it greatly exceeds the basic case's performance.

Case II: In this case, the triggering of T_s and T_a is synchronized with *zero* offset. Tasks T_s, T_c and T_a are triggered periodically with period h. The entire series of executions $T_s \rightarrow T_c \rightarrow Bus \rightarrow T_a$ is mostly expected to be finished by $\tau_{II} \approx h$ (see Fig. 7). Therefore, the *deadline* τ_{II} for the feedback signal is approximately equal to the sampling period h with such Φ_l. Since the feedback signal is delayed by one sampling interval, the control law (4) is not realizable and therefore, it is modified as follows:

$$u[k] = K \cdot x[k-1]. \tag{14}$$

In this case, the control law (14) is realizable when the feedback loop meets deadline τ_{II}.

From Cases I and II, we have seen that the execution of a feedback loop, i.e., $T_s \rightarrow T_c \rightarrow Bus \rightarrow T_a$, is associated with a deadline (τ_I and τ_{II}). Since the control tasks are usually time-triggered, the schedule of message m on the bus plays a key role in this context . In the case of priority-based arbitration on the communication bus, the transmission of message m might get delayed and reach ECU_2 after T_c has already been triggered – see Fig. 6 and Fig. 7. Such a timing scenario leads to a deadline miss, i.e., $\tau > \tau_I$ or $\tau > \tau_{II}$.

In a traditional approach, the communication schedule for message m is designed such that the deadline is always met. Towards this, the worst-case end-to-end delay of the feedback loop needs to be computed and this should essentially be within the deadline. As already mentioned, there has been a renewed interest in allowing certain deadline violations in the above context since the worst-case delay estimation is often overly conservative and only occurs very rarely.

5.1 Closed-loop system

Based on the above cases, the control law needs to be appropriately adapted. For example, for the Case I in Fig. 6, the following control law is adapted in [2, 20]:

$$u[k] = \begin{cases} Kx[k] & , \text{if } \tau \leq \tau_I \\ 0 & , \text{if } \tau > \tau_I \end{cases}$$

Here, it is typically assumed that the gain matrix K is designed for the lossless transmission case, e.g., via the LQG framework.

Similarly, for the Case II in Fig. 7, the following control law is adapted in [10, 11]:

$$u[k] = \begin{cases} Kx[k-1] & , \text{if } \tau \leq \tau_{II} \\ 0 & , \text{if } \tau > \tau_{II} \end{cases}$$

Of course, a more general case would be to consider deadlines of any length including multiple sampling periods [12].

In the above strategy, the control input $u[k]$ is applied only when the control message meets its deadline and $u[k]$ is set to *zero* otherwise. Based on this control strategy we have two systems: (i) when the deadline is met, $u[k]$ is applied and the resulting closed-loop system becomes A_c and, (ii) when the deadline is violated, $u[k] = 0$ and the resulting open-loop system is A. Note that the resulting system matrices A_c and A might have higher dimension due to the presence of feedback delay as in (14). Both for cases I and II, the closed-loop system becomes,

$$x[k+1] = A_\sigma x[k], \tag{15}$$

where $A_\sigma = A_c$ when the deadline is met and $A_\sigma = A$ in the case of deadline violation. Depending on the nature of the *delay pattern*, the closed-loop system keeps switching between A_c and A. Over a duration of l sampling periods, the closed-loop system is given by,

$$x[k+l] = A_{\sigma_{k+l}} \cdots A_{\sigma_{k+1}} A_{\sigma_k} x[k]. \tag{16}$$

The requirements from the control side mainly deal with stability and performance of the closed loop system (16). Each element of the overall system matrix sequence $A_{\sigma_{k+l}} \cdots A_{\sigma_k}$ results from either meeting or violating the deadlines.

5.2 Stability-based requirements

Clearly, the closed-loop system (16) is a *switched* system where the switching happens between $A_\sigma = A_c$ and $A_\sigma = A$ depending on the delay pattern. The question is: *What is the maximum frequency allowed for deadline violations with guaranteed stability of system (16)*?
A general answer to the above question follows from [21] with a slightly modified formulation of the control strategy,

$$u[k] = \begin{cases} Kx[k] & , \text{if } \tau \leq \tau_I \\ Kx[k-1] & , \text{if } \tau > \tau_I \end{cases}, \tag{17}$$

with $A_\sigma = A_1$ when $\tau \leq \tau_I$ and $A_\sigma = A_2$ when $\tau > \tau_I$.

THEOREM 1. (Stability condition) *Consider the closed-loop system (16) with control input (17) and assume that A_1 is Schur stable.*

- *If the open-loop system A is marginally stable, the system (16) is exponentially stable for $0 < r \leq 1$.*
- *If the open-loop system A is unstable, the system (16) is exponentially stable for*

$$\frac{1}{1 - \gamma_1/\gamma_2} < r \leq 1 \tag{18}$$

where r is the rate of met deadlines or the ratio of successful transmissions over the infinite horizon and

$$\gamma_1 = \ln \lambda^2_{max}(A_1), \gamma_2 = \ln \lambda^2_{max}(A).$$

The above theory holds true for any value of l in (16). For example, suppose we obtain $r = 0.1$ for a given choice of parameters in (16). With $l = 1000$, the above theorem implies that *any* 100 samples are allowed to violate their deadlines with guaranteed exponential stability. On the one hand, the above condition is very relaxed and generic enough to be applied to any system. On the other hand, such a condition is not suitable for analyzing performance-critical applications since it does not guarantee any performance (e.g., a desired settling time, etc.). Furthermore, the control law (17) is realizable if the worst-case delay of a feedback loop is bounded by one sampling interval. As already mentioned, it might be complex and overly pessimistic to design an architecture with only the worst-case delay estimation in mind.

A more structured requirement on stability was introduced in [12] where the systems are required to be stable with a desired *stability margin*. As discussed, not all feedback messages suffer the worst-case delay. With this observation, a deadline or threshold delay d_{th} (lesser than the worst-case delay) is chosen such that it is met by the *most* of the feedback messages. For example, one can choose $d_{th} = h$ in the case shown in Fig. 7. The control requirement is then represented as *delay frequency metric*.

DEFINITION 1 (DELAY FREQUENCY METRIC (d_{th}, n)). *If every feedback message with delay larger than d_{th} is followed by at least n feedback messages with delay no more than d_{th}, the* delay frequency metric *is said to be (d_{th}, n).*

For the delay frequency metric with $n = 3$, every sample violating the deadline is followed by at least three samples where the deadline is met. The overall system can then be represented as follows,

$$x[k + 2 + n_i + n_j] = AA_c^{n_i} \times AA_c^{n_j} x[k], \tag{19}$$

where $n_{i,j} \geq 3$. The stability is assured with a given stability margin by showing the existence of a Common Quadratic Lyapunov Function (CQLF) [15] between systems $AA_c^{n_i}$ and $AA_c^{n_j}$ for all combinations of n_i and n_j.

5.3 Performance-based requirements

In an assortment of recent works [2, 11, 20], the control requirements are specified using a notion of *exponential stability*,

$$\frac{\|x[k + l]\|}{\|x[k]\|} < \epsilon, \tag{20}$$

where $\|.\|$ denotes 2-norm. Hence, to ensure that the plant remains exponentially stable, any error must be reduced at least by a factor of ϵ in l sampling periods, i.e., $l \times h$ time. For example, $l = 5, \epsilon = 0.75$ means that any error signal must be reduced by at least 25% in five samples to maintain exponential stability of the system. It should be noted that the above notion of exponential stability is *stronger* compared to its definition found in control theory literature [8].

Coming back to the control requirement on exponential stability (20) and considering the closed-loop system (16), we obtain the following relation,

$$\begin{aligned} x_{k+l} &= A_{\sigma_{k+l}} \cdots A_{\sigma_k} x_k, \\ \Rightarrow \frac{\|x_{k+l}\|}{\|x_k\|} &\leq \left\|A_{\sigma_{k+l}} \cdots A_{\sigma_k}\right\| \\ &\Rightarrow \left\|A_{\sigma_{k+l}} \cdots A_{\sigma_k}\right\| < \epsilon. \end{aligned} \tag{21}$$

It follows that the exponential stability requirement can be re-written to (see [20, 2]):

$$ES(l, \epsilon) = \{\sigma_i \in \{o, c\}^\omega : \left\|A_{\sigma_{k+l}} \cdots A_{\sigma_{k+1}}\right\| < \epsilon \quad \forall k \in \mathbb{N}\}$$

$ES(l, \epsilon)$ is the language of strings σ over the alphabet $\{o, c\}$ corresponding to switching patterns of A and A_c that ensure that a possible error in the system is reduced by at least factor ϵ in l sampling periods. It was shown in [20, 2] that this language is ω-regular and how a Büchi automaton can be constructed to represent it.

Hence, ensuring the control performance requirement of exponential stability boils down to verifying that the system's switching between A and A_c remains within the acceptable patterns as specified by condition (21).

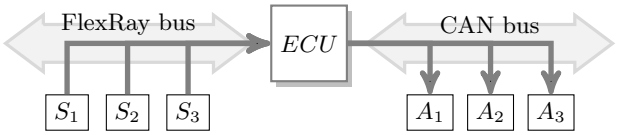

Figure 8: Example architecture.

The computation of this language of acceptable patterns or equivalently its Büchi automaton can be done by a brute-force search as described in [20, 2]. Model-checking this language pattern-by-pattern can become tedious however. To avoid this, a deadline constraint was used in [11]:

DEFINITION 2 ((f, H)-FIRM DEADLINE). *A stream of control messages is said to fulfill the (f, H)-firm deadline with respect to period h if at least f out of any H consecutive samples meet their deadline.*

Note that this definition takes a sliding window perspective. The idea is that among all possible patterns, one can rule out all the unacceptable ones by requiring a combination of (f, H)-firm deadlines. Translating the resulting set of (f, H)-firm deadlines to LTL is then straightforward. Once the allowable feedback signal drop patterns have been specified, the next step is to capture the timing characteristics or behaviors of the implementation platform and subsequently checking if they constitute a subset of the behaviors that the controller can tolerate. This guarantees that the *implementation* of the control system on this particular platform will meet the performance requirement.

5.4 Timing properties of architectures

Towards characterizing timing behaviors of the architectures, various techniques have been proposed in the area of real-time and embedded systems, as well as in the formal methods literature. Analyzing embedded platforms in the particular context of implementing distributed controllers has been studied in [11, 12]. In [12], the Real-Time Calculus [5] modeling formalism has been combined with the tool Uppaal [13] for verification of timed automata. Real-Time Calculus relies on specifying upper and lower bounds on the number of messages that might arrive at a communication resource over different time interval lengths. These arrival curves are typically denoted by $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ respectively. Similarly, the communication resource is modeled using upper and lower bounds on the number of messages it can transmit; let these service curves be denoted by $\beta^u(\Delta)$ and $\beta^l(\Delta)$ respectively. These bounds may then be used to compute the worst-case delays suffered by messages, in addition to properties like buffer requirements. The exact same techniques apply to both computation and communication resources alike.

When multiple message streams are to be scheduled on a communication resource, scheduling or resource arbitration policies like TDMA, fixed priority or EDF may also be modeled using Real-Time Calculus, where the *service bounds* $\beta^u(\Delta)$ and $\beta^l(\Delta)$ for the entire resource are transformed into similar bounds for each message stream (see [5] for details). Fig. 8 shows an architecture where sensor readings are transmitted over a FlexRay bus to a controller implemented on the ECU. Control messages are then transmitted over the CAN bus to actuators. The sensor-to-actuator delays experienced by the individual messages certainly influence the

stability and QoC of the controller. In order to compute this end-to-end delay, the service bounds of the individual architectural components (FlexRay and CAN buses and the ECU) need to be composed in order to obtain the service bound of the overall architecture. This is given by:

$$\beta^{\text{end-to-end}} = \beta^{\text{FR}} \otimes \beta^{\text{ECU}} \otimes \beta^{\text{CAN}}, \tag{22}$$

where $\otimes$ is the convolution operation as defined in Real-Time Calculus [5]. In order to estimate not only the maximum delay suffered by control messages, but also the *frequency* with which messages violate their deadline (derived from control-theoretic analysis, as described in the previous section) constraints, in [12] the service bounds or timing behaviors of the resource were transformed to a timed automata model that was then verified using UPPAAL.

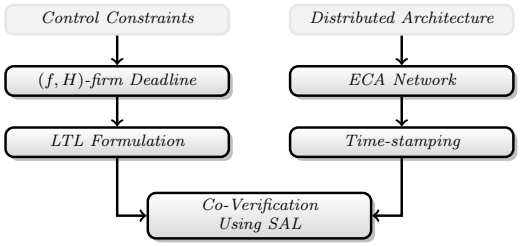

Figure 9: ECA-based co-verification workflow.

A similar technique was adopted in [11]. Its verification workflow is outlined in Fig. 9. First, the control performance specification is translated to LTL as described in the previous section. The communication hardware is then modeled as a network of Event Count Automata (ECA). ECA are a more flexible model for the streams that RTC is also modeling. Formally, ECA are tuples (see [6] for details)

$$\mathcal{A} = (S, s_{in}, X, V_{in}, Inv, \rho, \rightarrow) \tag{23}$$

where S is a set of states and s_{in} is the initial state, X is a set of count variables with V_{in} being their initial valuation. $Inv : S \rightarrow \Phi(X)$, the Invariant Constraint Function, describes the possible valuations for every state whereas $\rho : S \rightarrow \mathbb{N} \times \mathbb{N}$, the rate function, indicates how many events can occur in a state and how the count variables evolve. Finally, $\rightarrow \subset S \times \Phi(X) \times 2^X \times S$, the transition relation, describes how the system can evolve from state to state.

The language of an ECA consists of strings that record the event counts that occur along the path. This can be more readily seen from an example like the periodic with jitter arrival automaton in Fig. 10. There, the automaton starts in the initial state $s_{in} = A$ with initial valuation $V_{in} = \{x = 0\}$. It then moves to B with either $x = 0$ or $x = 1$. In the third step, it progresses to C, recording again $x = 0$ or $x = 1$ depending on what the valuation of x was in the second step before resetting and starting over. This produces strings of 100 or 010 that are repeated infinitely often. These strings indicate how many events arrive or how much service is available in a particular time step. This is an extension of the RTC concept of arrival and service curves. For any pair of upper and lower arrival curves α^u and α^l, there is an ECA for which it holds:

$$L_{ECA} = \left\{ R(t) : \alpha^l(t) \leq R(t) \leq \alpha^u(t) \right\} \tag{24}$$

The reverse is not true – ECA are in fact more expressive since they can accommodate state-based rate changes.

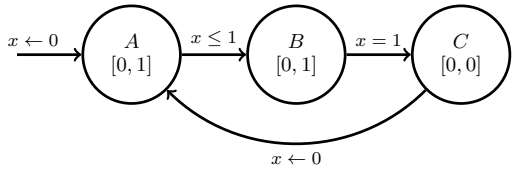

Figure 10: Periodic with jitter arrival ECA ($p = 3, j = 2$).

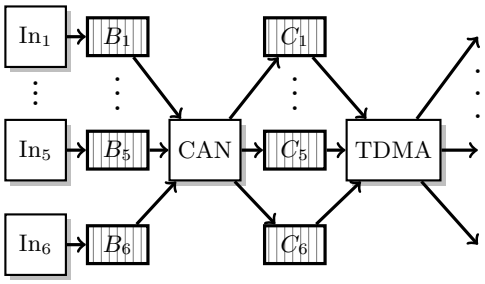

Figure 11: Running example modeled as ECA network.

On the next level, multiple ECA can be connected in a network of ECA by introducing buffers in between them. The network starts with arrival ECA that define the system inputs that are deposited into the first buffers. From there, service ECA process data in their input buffers to their output buffers – see Fig. 11 for an example.

To keep track of the delay for complicated patterns across the system, time-stamps for the messages are introduced (see Fig. 12). This allows running an evaluation automaton at the end that produces outputs that are compatible with the $\{o, c\}$ patterns from $ES(l, \epsilon)$.

5.5 Interface-based approach

Both the timed automaton approach from [12] and the timestamped ECA method from [11] are steps towards a common goal. The aim is to provide a richer interface for the specification of hardware/software requirements.

The next step in line with the intention of [2] is to formally specify the miss patterns of the architecture as a language $L_{architecture}$. It could then be combined with approaches that rely on *interface theory* [4, 7, 3]. More precisely, it would be possible to compare a language $L_{controller}$ that models the requirements of the control system, e.g., $ES(l, \epsilon)$. These two languages constitute the *interfaces* of the controller and the architecture. Checking the *compatibility* of these two interfaces now boil down to the problem of checking for language inclusion, i.e., whether

$$L_{controller} \subseteq L_{architecture}$$

Satisfaction of this inclusion implies that the controller may be implemented on the given architecture. This could then more readily be propagated through the system to verify multiple control applications. The work reported in [2, 20] leads into this direction, but it was written from a scheduling point of view. In other words, the requirements of multiple control systems were established and it was evaluated if all of them can be executed on a common processor.

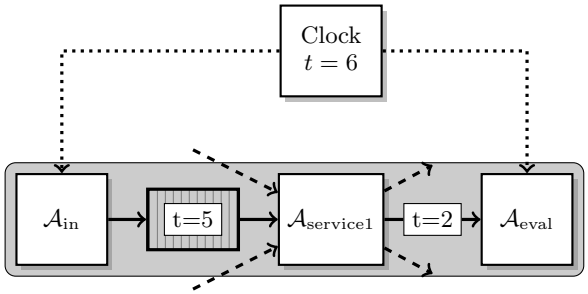

Figure 12: A stream of time-stamped messages in a network of ECA.

6. CONCLUDING REMARKS

Whereas, to a great extent, combustion cars still rely on mechanical fall-back systems, all safety-critical control applications in electric vehicles are exclusively implemented based on embedded electronics and software. As a result, it is necessary to carefully design such systems, in particularly, in the context of electric cars.

Nowadays, since model-based design methods dominate the design and development of automotive control systems, there is an increasing need for techniques that allow guaranteeing that the properties specified by high-level models translate into correct implementations. Towards this, it is necessary to utilize information about the underlying architecture at the model level and vice versa, i.e., to design the architecture using information provided by models.

In this paper we presented different techniques for modeling control performance requirements. Moreover, we discussed how to use those performance parameters to optimize an implementation and, in particular, the schedules on the ECUs and on typical automotive buses such as FlexRay. We showed that an iterative procedure increasingly involving information provided by models and that the implementation leads to more resource-efficient designs. Finally, we discussed different model-based techniques to formally verify properties of control systems such as timing, stability, and performance on a given architecture.

7. REFERENCES

[1] FlexRay protocol specification version 2.1. http://www.flexray.com.

[2] R. Alur and G. Weiss. Regular specifications of resource requirements for embedded control software. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.

[3] P. Bhaduri and S. Ramesh. Interface synthesis and protocol conversion. *Formal Asp. Comput.*, 20(2):205–224, 2008.

[4] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *EMSOFT*, pages 117–133, 2003.

[5] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.

[6] S. Chakraborty, L. T. X. Phan, and P. S. Thiagarajan. Event count automata: A state-based model for stream processing systems. In *26th IEEE Real-Time Systems Symposium (RTSS)*, pages 87–98, 2005.

[7] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT*, pages 148–165, 2001.

[8] R. C. Dorf and R. H. Bishop. *Modern Control Systems*. Addison Wesley, 1995.

[9] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. Time-triggered implementations of mixed-criticality automotive software. In *Design, Automation and Test in Europe (DATE)*, 2012.

[10] D. Goswami, R. Schneider, and S. Chakraborty. Co-design of Cyber-Physical Systems via controllers with flexible delay constraints. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2011.

[11] M. Kauer, S. Steinhorst, D. Goswami, R. Schneider, M. Lukasiewycz, and S. Chakraborty. Formal verification of distributed controllers using time-stamped Event Count Automata. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013.

[12] P. Kumar, D. Goswami, S. Chakraborty, A. Annaswamy, K. Lampka, and L. Thiele. A hybrid approach to cyber-physical systems verification. In *Design Automation Conference (DAC)*. ACM, 2012.

[13] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[14] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. Modular scheduling of distributed heterogeneous time-triggered automotive systems. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 665–670, 2012.

[15] O. Mason and R. Shorten. On common quadratic Lyapunov functions for stable discrete-time LTI systems. *IMA Journal of Applied Mathematics*, 69(3):271–283, 2002.

[16] P. Naghshtabrizi and J. Hespanha. Analysis of distributed control systems with shared communication and computation resource. In *ACC*, 2009.

[17] S. Samii, P. Eles, Z. Peng, and A. Cervin. Design optimization and synthesis of FlexRay parameters for embedded control applications. In *DELTA*, 2011.

[18] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty. Constraint-driven synthesis and tool-support for FlexRay-Based automotive control systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.

[19] H. Voit, R. Schneider, D. Goswami, A. Annaswamy, and S. Chakraborty. Optimizing hierarchical schedules for improved control performance. In *International Symposium on Industrial Embedded Systems (SIES)*, 2010.

[20] G. Weiss and R. Alur. Automata based interfaces for control and scheduling. *Hybrid Systems: Computation and Control (HSCC)*, 2007.

[21] W. Zhang, M. S. Branicky, and S. M. Phillips. Stability of networked control systems. *IEEE Control Systems*, 21:84–99, 2001.