Formal Verification of Distributed Controllers using Time-Stamped Event Count Automata

Matthias Kauer[1], Sebastian Steinhorst[1], Dip Goswami[2],
Reinhard Schneider[2], Martin Lukasiewycz[1], Samarjit Chakraborty[2]

[1] TUM CREATE, Singapore, Email: matthias.kauer@tum-create.edu.sg
[2] TU Munich, Germany, Email: samarjit@tum.de

***Abstract*—We study distributed controllers where sensor, controller, and actuator tasks are mapped onto different processors or Electronic Control Units (ECUs) in a distributed automotive architecture, communicating via a shared bus. Controllers in such setups are designed with a sampling period equal to the worst-case sensor-to-actuator message delay. However, this assumption of all messages having to meet their deadlines is too pessimistic. The inherent robustness of most controllers allows some of the messages to miss their deadlines, while still meeting specified control performance constraints. Given a controller, in this paper we first quantify the frequency of its acceptable deadline misses and represent this as a Linear Temporal Logic (LTL) formula. Further, we model the distributed architecture as a network of *Time-Stamped Event Count Automata (TS-ECAs)*. Such a network of TS-ECAs is then model-checked to verify whether it satisfies the LTL formula. The verification ensures that the controller may be mapped onto the architecture and the control performance constraints will be satisfied. We have implemented this methodology in Symbolic Analysis Laboratory (SAL), which is a well-known framework combining different tools for system verification. Our implementation and case studies using standard controller design shows the applicability of our proposed controller/architecture *co-verification*. It represents a significant improvement in current design flows where, although controller models are formally verified, their *implementation* on a distributed architecture is done in an ad hoc fashion with extensive testing and integration effort.**

I. Introduction

Distributed implementations of feedback control applications are common in many safety-critical domains like automotive and avionics. In this work, we consider a setup where a feedback control application is implemented on multiple ECUs that communicate over a shared bus. As shown in Fig. 1, the sensing devices and the actuators are connected to different ECUs and the control algorithm uses feedback signals that are transfered via the *shared* bus system. While being transmitted over such a bus, feedback signals often get delayed due to contention by other messages, thereby resulting in a *sensor-to-actuator* delay.

Traditionally, a feedback control application is designed to tolerate a maximum specified sensor-to-actuator delay and the underlying implementation architecture (i.e., ECU and bus schedules) is chosen to meet such an end-to-end delay requirement.

Hence, the specified maximum sensor-to-actuator delay for which the controller has been designed acts as a *deadline* for the feedback control messages. Towards this, a control application is mapped onto an architecture only when *all* instances of control messages are guaranteed to meet their deadlines. Such a design approach to convert control performance requirements to *hard* timing constraints often turns out to be pessimistic and fails to exploit the inherent robustness of the feedback control loops.

This work was financially supported in part by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

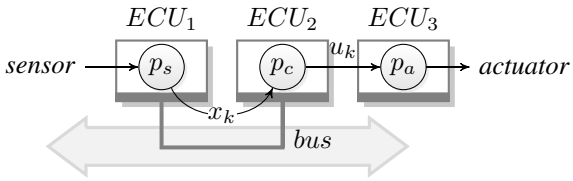

Fig. 1. A distributed control application (p_s, p_c and p_a are sensor, controller and actuator task respectively)

In contrast to the above *hard* real-time assumption, we consider a design that exploits the robust nature of the feedback loops and allows some of the control messages to miss their deadlines, while still meeting the desired control performance (e.g., stability and error-bound). In this work, we intend to formally *verify* such a design. This is split into two parts (see Fig. 2): (i) How frequently and in what order (or pattern) can the control application tolerate deadline violations? This first involves control-theoretic tools, followed by translating the result into a formally verifiable specification, e.g., an LTL formula. (ii) Modeling the implementation architecture – i.e., the ECUs, the bus, and their schedules – so that it can be formally verified, e.g., as an automaton. If the LTL formula is satisfied by the automaton, the architecture in question is a valid implementation of the controller.

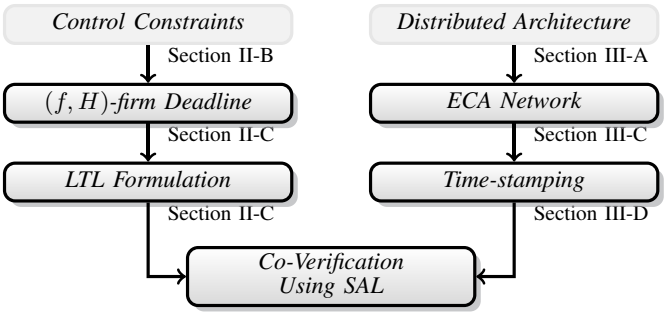

Fig. 2. Proposed co-verification framework

Our contributions and related work: Formal verification of high-level controller models and, to some extent, also the verification of the generated code has been studied before. However, *co-verification* of controllers and their implementation architectures – as we propose in this paper – has received

978-1-4673-3030-5/13/$31.00 ©2013 IEEE

significantly less attention so far. As implementation architectures become more complex and distributed, the semantic gap between high-level controller models and their actual implementations continues to increase, thereby necessitating the need for such co-verification.

Our work is related to Networked Control System (NCS) where controllers and stability issues [1] are studied in the presence of feedback irregularities such as delay, jitter and packet/message drops [2]. However, such studies have largely used tools from control theory and the *verification* of the *architecture* has been completely ignored. Architecture and implementation issues for distributed controllers – e.g., the problem of determining suitable ECU and bus schedules – have been recently studied in [3], [4], [5], [6] and a number of other related papers. On the other hand, formal verification of control software has been addressed in [7], [8]. In a similar direction, it was shown that robustness (to deadline misses) of feedback control loops in terms of meeting an *exponential stability* requirement (which is a stronger stability notion than *asymptotic* stability) can be expressed as an ω-regular language [9]. Our work is also similar to that in [10] where model checking was used to jointly verify controllers and their implementations. However, in contrast to [10], we have a more explicit model of the architecture, making it more general.

As described above, given a controller and certain control performance constraints, we first specify the allowable deadline misses as an LTL formula. Our architecture model is motivated by *Event Count Automata (ECAs)* [11] which have been shown to be a good model for capturing the timing properties of *streaming* applications (e.g., a *stream* of control messages in our case). However, we first extend ECAs with *time-stamps* and model an architecture with multiple ECUs and buses as a network of TS-ECAs, with each TS-ECA representing the timing properties of one component of the architecture (e.g., an ECU or a bus). This is followed by model checking to verify whether the LTL formula is satisfied by the network of TS-ECAs. Our approach allows an architecture to be checked for a wide variety of deadline miss patterns, which was not done in [10]. It also generalizes the work in [9] by making it applicable to a wide variety of architectures.

The rest of this paper is organized as follows. We first introduce the class of control systems under consideration, their requirements, and discuss how LTL formulations are done for the control requirements in Section II. In Section III, we introduce ECAs, their extension to a network of time-triggered ECAs for architecture modeling, and discuss the co-verification technique. Section IV demonstrates the applicability of the proposed co-verification method on experimental results while Section V draws the conclusions.

II. Control Requirements: Specifications

In this section, we present a representative control system example and its requirements that are retained throughout the paper. A feedback control system (as shown in Fig. 3) aims to regulate the behavior of a dynamic system (i.e., plant) by applying an appropriate input (i.e., control input) to it. In this work, we consider a discrete-time linear-time-invariant (LTI) system as a plant model,

$$x_{k+1} = Ax_k + Bu_k, \tag{1}$$

where (A, B) are system and input matrices and x_k, u_k denote the state variable and the control input respectively.

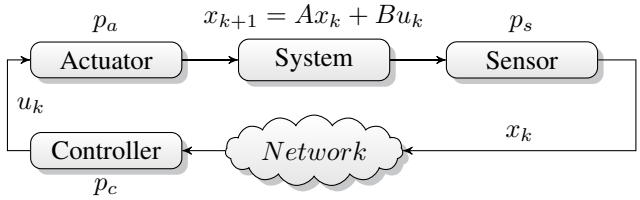

Fig. 3. Distributed control over a network

The discrete-time plant in (1) is derived with a sampling period $T = 3.5ms$. As example for the case study, we assume $A = \begin{bmatrix} 0.611 & 0.284 \\ 0.239 & 0.7253 \end{bmatrix} \in \mathbb{R}^{2\times2}$, $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \in \mathbb{R}^{2\times1}$. Hence, we consider a second-order discrete-time plant with sampling period $3.5ms$ to be controlled.

Exponential stability requirement: As a performance requirement, we consider the notion of exponential stability [9]. In this context, the exponential stability $ExpStab(l, \epsilon)$ is defined by

$$\frac{\|x_{k+l}\|}{\|x_k\|} < \epsilon, \tag{2}$$

where $\|.\|$ denotes 2-norm. Consequently, to ensure that the plant remains exponentially stable, any error must be reduced by at least a factor of ϵ in l sampling periods, i.e., $l \times T$ time. In our system, we require $ExpStab(5, 0.75)$, i.e., any error signal must be reduced by at least 25% in 5 samples (or $5 \times 3.5ms$ time) to maintain exponential stability of the system.

A. Implementation Architecture

A feedback control performs three operations: *measure*, *compute*, and *actuate*. In the setup under consideration, a control application is partitioned in three tasks: sensor task (p_s), controller task (p_c), and actuator task (p_a) running on three different ECUs. ECU_1 and ECU_2 communicate over a CAN bus, whereas ECU_3 is directly attached to ECU_2. In Fig. 1, p_s reads the state x_k in ECU_1 from the sensors and sends it to the bus. On ECU_2, p_c receives x_k, computes u_k based on x_k and stores it in a buffer. The periodically executed p_a then takes the computed input signal u_k – if it is present – and applies it at the beginning of the new period. Fig. 4 illustrates the timing diagram of such a control application. The time duration between reading sensor data x_k and the computation of u_k in ECU_2 is delay τ. Since the actuator can only apply it if the new signal is computed by the end of the period, we

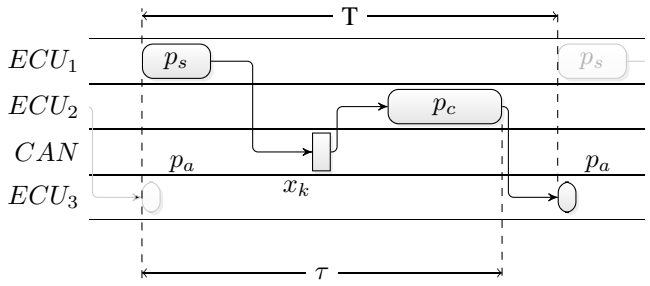

Fig. 4. Timing diagram of the distributed controller under consideration

consider the sampling period T as a *deadline* for the control signal to reach ECU_3. If $\tau > T$, we consider this to be a deadline miss.

B. Control strategy

Based on the above delay variation, we apply the following control strategy,

$$u_k \quad = \quad \begin{cases} Kx_{k-1} & \text{, if } \tau \leq T \\ 0 & \text{, if } \tau > T \end{cases}$$

Thus, the control input u_k as state-feedback with delayed state is applied only when the control message u_k meets its deadline and u_k is set to zero otherwise. Based on this control strategy we have two systems: (i) when $\tau \leq T$, $u_k = Kx_{k-1}$ and the resulting closed-loop system becomes A_c and, (ii) when $\tau > T$, $u_k = 0$ and the resulting open-loop system becomes A_o. Note that the resulting system matrices A_c and A_o have higher dimensions due to the presence of delay in the feedback loop. The closed-loop system becomes

$$x_{k+1} = A_\sigma x_k, \tag{3}$$

where $A_\sigma = A_c$ when $\tau \leq T$ and $A_\sigma = A_o$ when $\tau > T$. Depending on the nature of the sensor-to-actuator delay, the resulting closed-loop system keeps switching between A_c and A_o.

C. Linear Temporal Logic (LTL) formulation

Coming back to the control requirement on exponential stability (2) and considering the closed-loop system (3), we obtain the following relation,

$$\begin{aligned} x_{k+l} &= A_{\sigma_{k+l}} \cdots A_{\sigma_k} x_k, \\ \Rightarrow \frac{\|x_{k+l}\|}{\|x_k\|} &\leq \left\| A_{\sigma_{k+l}} \cdots A_{\sigma_k} \right\| \end{aligned} \tag{4}$$

In other words, the exponential stability requirement can be re-written as follows (see [9], [12]),

$$\begin{aligned} &ExpStab(l, \epsilon) = \\ &\quad \left\{ \sigma_i \in \{o, c\}^\omega \, : \| A_{\sigma_{k+l}} \cdots A_{\sigma_{k+1}} \| < \epsilon \quad \forall k \in \mathbb{N} \right\}, \end{aligned} \tag{5}$$

which essentially is the language of strings σ over the alphabet $\{o, c\}$ corresponding to switching patterns of A_o and A_c that ensure that a possible tracking error in the system is reduced by at least factor ϵ in l sampling periods.

Hence, the control requirement on exponential stability boils down to a set of acceptable patterns of occurrence of A_o and A_c that meet condition (4). The computation of such a set of acceptable patterns can be done by a brute-force search as in [9], but it becomes tedious to verify them pattern-by-pattern. To avoid this, we resort to the following deadline constraint:

Definition 1 ((f, H)-firm deadline)**.** *A stream of control messages is said to fulfill the (f, H)-firm deadline T if at least f out of any H consecutive messages meet their deadline.*

The idea is that among all possible patterns, one can rule out all the unacceptable ones by a combination of (f, H)-firm deadlines. For example, we evaluate all the patterns and separate those that do not fulfill $ExpStab(5, 0.75)$ in our example. We then see that we can exclude all the bad patterns by requiring our system to be both (1,2)-firm and (3,5)-firm with respect to its period. That is, in any two samples at most one message, and in any 5 samples at most two messages can have delay $\tau > T$. Therefore, the set of all acceptable patterns is represented by a combination of such (f, H)-soft deadlines where $H \leq l$.

Next, this kind of requirement can be written as an LTL formulation – slightly modifying the bounded existence formulation in the patterns project [13], a collection of common LTL specifications – in the form:

$$\begin{aligned} &G\,\{(c = c_0) \Rightarrow \\ &\quad \text{NOT(fail) W } ((c = c_0 + H) \text{ OR fail AND } X\,[\\ &\quad\quad \text{NOT(fail) W}(c = c_0 + H) \\ &\quad]) \\ &\} \end{aligned} \tag{6}$$

Here, $c = c_0$ and $c = c_0 + H$ refer to evaluation cycles on the current evaluation automata that we introduce later (Fig. 7). A cycle corresponds to the sampling period of the control process and the automaton will evaluate whether the system runs in open- or closed-loop. The specification says that during the window $[c_0, c_0 + H]$, "fail" can occur at most once. This LTL formulation has to be checked for every possible c_0 of these cycles that run round-robin with maximum value H. H is the period that shall be observed, i.e., if the stream has a $(1, 3)$-firm deadline, then $H = 3$ has to be chosen. The amount of acceptable deadline misses can be varied by adding additional lines similar to (6). If there are multiple formulations of this kind, they can be checked sequentially.

III. Architecture Model

A. Running Example – Communication Architecture

We now model the hardware platform from Fig. 1 and the communication pattern from Fig. 4. Towards this, we introduce the ECA framework and see how we can describe this setup as an ECA network. We then present a time-stamp extension that lets us verify the LTL specification we found in Section II-C.

B. Event Count Automata

As introduced in [11], ECAs are tuples

$$\mathcal{A} = (S, s_{in}, X, V_{in}, Inv, \rho, \rightarrow) \tag{7}$$

where

- S is a set of states and s_{in} is the initial state
- X is a set of count variables
- V_{in} is the initial valuation of the count variables.
- $Inv : S \rightarrow \Phi(X)$ is the Invariant Constraint Function with

  $$\Phi(X) = x \leq c | x < c | x \geq c | x > c | \varphi_1 \wedge \varphi_2$$

  It assigns invariance constraints to the states.
- $\rho : S \rightarrow \mathbb{N} \times \mathbb{N}$ is the rate function. Every state is assigned an interval for the arrival or service rate in that state:

  $$\rho(s) = [l, u]$$

- $\rightarrow \subset S \times \Phi(X) \times 2^X \times S$ is the transition relation.

The *language* of ECAs contains strings of integers that denote certain arrival or processing patterns of data – mostly control messages in our case. For example, in the case of an *arrival ECA* $''201''$ denotes an arrival pattern with two messages arriving in the first interval, no messages in the second interval and one message in the third interval. In the

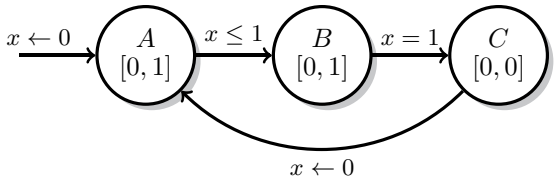

Fig. 5. Periodic with jitter arrival ECA ($p = 3, j = 2$)

case of a *service ECA*, it denotes the amount of data that is processed during the respective intervals.

An ECA starts in the configuration (s_{in}, V_{in}) – an initial state and an initial valuation of all the count variables. In the case of the periodic with jitter automaton in Fig. 5, this corresponds to state A and the only count variable $x = 0$. From there the ECA can make moves $(s, V) \overset{k}{\Rightarrow} (s', V')$. To make such a move, there needs to be a transition from s to s' and $V' = (V + k)$ has to be in accordance with the rate intervals of the state (here $\rho(A) = [0, 1]$), the guards of the transition ($x \leq 1$) and possible invariants (none in B). Additionally, some count variables may have to be reset (such as x when moving from C to A).

Transitions are considered urgent, i.e., they have to be taken if possible. If no transition is possible, the automaton can also remain in its current state.

A string $\sigma = n_1 n_2 \dots n_k \in [0, \rho_{\max}]^\omega$ is accepted if and only if the automaton can produce a sequence $(s_0, V_0) \overset{n_1}{\Rightarrow} (s_1, V_1) \overset{n_2}{\Rightarrow} (s_2, V_2)$. Our example ECA in Fig. 5 accepts strings that have one event occurring in either state A or B. This results in a jitter of $j = 2$ and a period of $p = 3$. No events can occur in C and the guard $x = 1$ on transition (B, C) guarantees that it occurred beforehand. For a more rigorous description of ECA semantics, please refer to [11].

C. ECA Networks

ECAs can be connected to form an ECA network which is a structure

$$\mathcal{N} = (\{\mathcal{A}_p\}_{p\in\mathcal{P}}, \{U_p\}_{p\in\mathcal{P}}, \mathcal{B}, b, C, IN, OUT) \tag{8}$$

In this definition:

- $\mathcal{P}$ is a finite set of nodes that the various ECA $\mathcal{A}_p$ are associated with.
- U_p is the update function that defines how an ECA consumes and deposits items from its buffers.
- $\mathcal{B}$ is a finite set of buffers of capacity b.
- C is the maximum number of items that any ECA in the network can handle in one step. This needs to be finite for the state space behind the ECA that we need to explore to remain finite.
- $IN : \mathcal{P} \to 2^{\mathcal{B}}$ and $OUT : \mathcal{P} \to 2^{\mathcal{B}}$ link the buffers to the ECA. It is assumed that every $\mathcal{A}_p$ has at least one buffer and that its input and output buffers are disjoint. Furthermore, buffers cannot be shared. To sum up:
  1) $IN(p) \cup OUT(p) \neq \emptyset$
  2) $IN(p) \cap OUT(p) = \emptyset$
  3) $IN(p) \cap IN(q) = OUT(p) \cap OUT(q) = \emptyset \quad \forall p \neq q$
  4) For any buffer $\mathcal{B}$, $\exists p$ such that $B \in IN(p)$ or $B \in OUT(p)$

Our running example then results in the ECA network in Fig. 6. The stream we want to verify is the lowest-priority stream 6. The ECAs' update functions are chosen such that the streams remain separate and are processed according to their priority or their schedule on the Controller Area Network (CAN) and the Time Division Multiple Access (TDMA) schedule on the ECU, respectively.

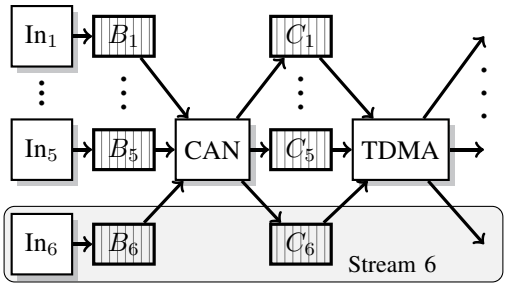

Fig. 6. Running example modeled as ECA network

D. Time-Stamped Event Count Automaton (TS-ECA)

In the original ECA framework, it is impossible to track patterns of missed deadlines, because once the messages are in the network, we do not know when they arrived. The buffers do not hold individual messages either – only the number of messages that are present in a specific buffer is stored. This is sufficient to answer the questions the ECAs were originally designed for like: "Is there a buffer overflow?" or "What is the maximum end-to-end delay of a stream?". In our framework, the messages that are transmitted are usually sensor readings of a state that are not useful once a new reading becomes available. We therefore model all the buffers that pertain to control message streams as size 1 and let new messages overwrite old ones.

On top of this, because we want to track patterns of individual message delay, we introduce a global clock that runs in a round-robin fashion to keep the state space finite. We then time-stamp messages as they enter the system, track their way from buffer to buffer through the system and compare their stamp to the global clock once they are through the network.

For tracking purposes, we restrict the part of the ECA network that we want to validate to non-merging streams, as it is typical in independent control applications. If we want to consider merging streams as well, we can tag the merged message with the time of the older message, but this needs to be investigated further.

This leads us to the Time-Stamped Event Count Automaton (TS-ECA) network:

$$\begin{aligned}\mathcal{N} = (\{A_p\}_{p\in\mathcal{P}}, \mathcal{B}, b, C, IN, OUT,\\ \mathcal{M}, M, \mathcal{S}, (\mathcal{E}_i)_{i\in\mathcal{S}}, \mathrm{CLK}_{\max})\end{aligned} \tag{9}$$

In this definition:

- $\mathcal{S}$ is the set of the streams that need to be tracked.
- M is an upper bound for the amount of messages any of the tracked streams has inside the system at any time.
- $\mathcal{M} = \{m_{ij} : i \in \mathcal{S}, j \in [0..M]$ is the message array where the time-stamps are stored.
- IN, OUT are as in (8).
- The node set $\mathcal{P}$ is constrained such that the tracked streams in $\mathcal{S}$ are not allowed to merge.
- $\mathcal{A}_p$ and U_p do not change from (8).

- The buffers $\mathcal{B}$ still hold the messages while they are waiting for processing. Whereas before, it was sufficient to imagine the buffers as a counter that went up and down depending on how many elements were processed, we now need to actually move the messages – or their time-stamps – through the system.
- $\mathrm{CLK}_{\max}$ is the maximum time the global clock adopts before resetting in a round-robin fashion. This limit must be higher than the longest time it takes a message to leave the system again.
- $\mathcal{E}_i$ are the evaluation automata that supervise the message streams in $\mathcal{S}$. These are simple automata with three states – *fail*, *neutral*, and *success*. Fig. 7 shows how they are implemented. They remain neutral until the periodic actuator task they correspond to becomes active. It is then checked whether a message has arrived during the last period and whether its age is less than d_{eval}. d_{eval} can be chosen arbitrarily, but here we choose $d_{eval} = T$, the sampling period.

Note that the parameters M, $\mathrm{CLK}_{\max}$ and $\mathcal{S}$ have an immediate effect on the computation time of the model checker, so they need to be chosen carefully.

The overall situation of a supervised stream is shown in Fig. 8. It is started by an arrival ECA and ends with an evaluation automaton. Messages that go into the system are stamped with the value of the clock in that step. Here, we see an old message with $t = 2$ and a more recent message with $t = 5$. In between, they are processed by a shared service ECA (as in Fig. 8, the dashed connections are from and to competing streams) or possibly a sequence of service ECAs.

E. Implementation

We implemented the TS-ECA in SAL [14]. The message array and its task of time-stamping the messages was implemented within the evaluation automata. Towards this, they communicate with the arrival ECA as well as with the last service ECA in a stream via in- and output variables.

Instead of inserting the messages with their timestamps attached directly into the system and passing them around as one might have preferred in an object-oriented environment, we have the buffers store and pass around indices that refer to the timestamps managed by the evaluation automata.

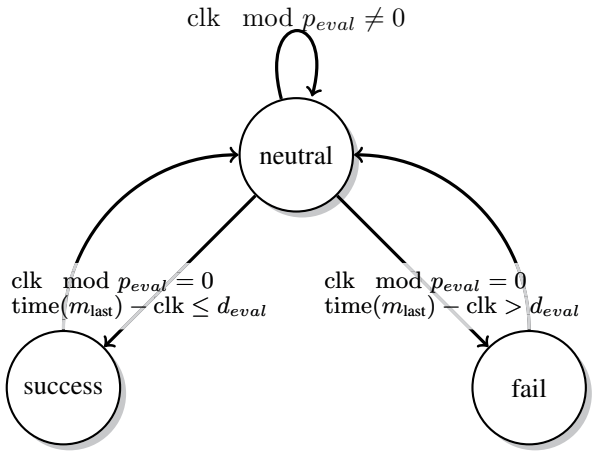

Fig. 7. Evaluation Automaton

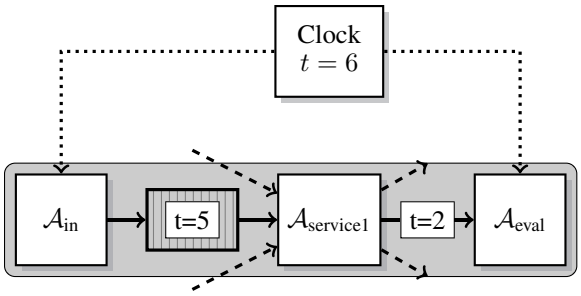

Fig. 8. A stream of time-stamped messages in a network of ECA

Concretely, the messages that enter the system are numbered in a round-robin fashion. The buffers store an index that marks the position of the messages it currently holds with respect to all the messages in a stream array.

The buffers are initialized with $\mathrm{idx}_0 = 0 = \mathrm{buf}_0$ and updated via

$$\mathrm{buf}_{t+1} = \max\left(0, \min\left(b, \mathrm{buf}_t + \mathrm{in}_{t+1} - \mathrm{out}_{t+1}\right)\right)$$

$$\mathrm{idx}_{t+1} = \begin{cases} \mathrm{idx}_{\text{in}} & \text{, if } \mathrm{idx}_{\text{in}} \text{ valid and } \mathrm{in}_{t+1} > 0 \\ \mathrm{idx}_t + \mathrm{in}_{t+1} & \text{, otherwise} \end{cases}$$

IV. Case Study

In this section, we revisit the architecture that we described in the previous sections and illustrated in Fig. 1 and Fig. 6 respectively. As already described, the control message is transmitted over the CAN which is a shared resource. In this case, we consider five other higher-priority message streams which are interfering with the control message. All the messages are assumed to be periodic with jitter as detailed in Table I. The control message has a period of $3.5ms$ which is equal to the sampling period T of the discrete-time plant in (1). Therefore, the deadline for control message is $3.5ms$ and the control strategy is as per (3). The message length is assumed to be 8 bytes and it results in $0.21ms$ transmission time overhead on the 512kbit CAN bus.

For this given architecture, we performed standard worst-case timing analysis [15] for the control message and obtained a delay of $\tau = 4.84ms$. Clearly, this is larger than the controller's sampling period of $T = 3.5ms$ and the performance requirement could not be guaranteed with a design with such a worst-case delay.

We then apply the proposed co-verification framework to address the above problem. Note that ECA are based on an implicitly chosen smallest time step. The obvious choice would be the smallest time that occurs in the system. This is very small however and leads to an explosion of the state space. In this case study, we therefore set the ECA's

Task i	Period p_i	Jitter j_i	Exec time e_i
1	1.0ms	0.5ms	0.1ms
2	1.0ms	1.0ms	0.1ms
3	2.0ms	1.5ms	0.3ms
4	2.0ms	0.5ms	0.15ms
5	3.0ms	1.5ms	0.15ms
6	3.5ms	0.5ms	0.2ms

TABLE I
Case Study Parameters

time unit to $0.5ms$ such that the CAN bus processes two messages in every step. We then implemented this problem using TS-ECA in SAL as discussed throughout this paper. We let the sensor and actuator task execute periodically and synchronized. As mentioned in Section II-C, the performance requirement $ExpStab(5, 0.75)$ from (5) can be expressed as a combination of two (f, H)-firm deadlines – (3,5) and (1,2)-firm with respect to the sampling time T.

(1) Example of a valid architecture: We consider the control specification as (3,5) and (1,2)-firm deadlines, and the architecture model as described above with the interfering message properties being as shown in Table I. Using our framework, we verified that our TS-ECA fulfills these soft deadlines requirements (or exponential stability requirement). This indicates that the control application can be implemented on the given architecture meeting the $ExpStab(5, 0.75)$ requirement. To evaluate the resulting control performance, we obtained the worst-case trace of delay from SAL for the architecture with message properties as in Table I (load on the bus: 90%) and simulated the corresponding system behavior in terms of $\|x\|$ in MATLAB. In Fig. 9, we compared the control performance from such a delay-trace and the control performance with ideal transmission without any deadline miss. It can be seen that the architecture that satisfies the specifications (that are derived from the exponential stability constraints) reduces an error signal by 72% in 5 samples while the same system without any deadline miss (i.e., with ideal transmission) could reduce 82% in the same duration.

(2) Example of an invalid architecture: We try to implement the same control algorithm on an architecture with interfering message properties slightly different from Table I. With $j_4 = 1.5ms$ or $p_5 = 2.5ms$, the given architecture fails to meet (3,5) and (1,2)-firm deadlines. Essentially, this means that an architecture with modified message properties fails to guarantee the $ExpStab(5, 0.75)$ requirement. We obtained a similar delay-trace as the previous example with $p_5 = 2.5ms$ (load 92%). In Fig. 9, we can see that the control performance deviates significantly (only 52% compared to 72% in 5 samples in the case of valid architecture) from the one with ideal transmission without deadline miss. Although the invalid architecture in this example still meets the controller's exponential stability criteria that any error signal should be reduced by at least 25% in 5 samples, it is possible to find some delay-trace and initial condition by extensive simulation that will cause a violation of the control constraints.

It can be noticed from the above two examples that a small change in the architecture (e.g., period and jitter of interfering messages) can lead to a significant impact on the resulting control performance. Clearly, this shows the necessity for such a formal verification technique in order to avoid extensive testing and integration effort in an incremental design process.

V. Concluding Remarks

In this paper, we proposed a co-verification framework that can essentially verify whether a given distributed architecture is suitable to implement a control application with a specified performance requirement. From the control side, the basic idea is to quantify the inherent robustness of the feedback loop and translate it into an LTL formula. From the architecture side, the entire platform is modeled as a time-stamped network of ECAs. Finally, it is verified if the architecture model satisfies the control specifications using SAL. In this case, the architecture is valid for the implementation of the controller. Our results show how the effect of the robust nature of a feedback loop can be exploited for verifying its implementation on a given architecture. As a part of future work, we would like to investigate various other control requirements and the generation method of the corresponding specifications. We also want to look closer at the scalability of the SAL implementation.

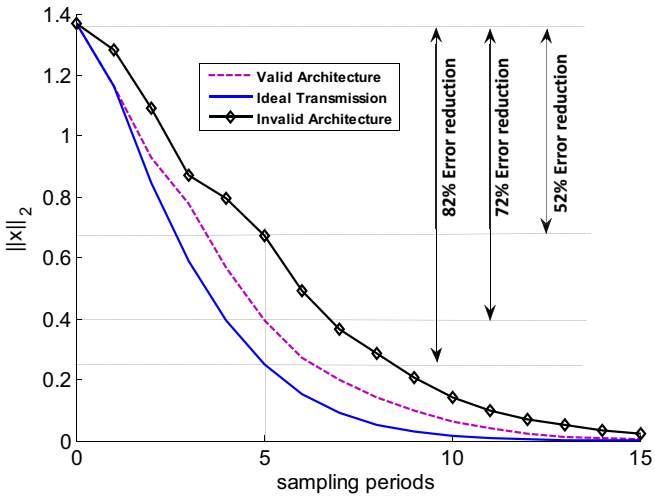

Fig. 9. Control performance evolution for different architectures

References

[1] G. C. Walsh, H. Ye, and L. G. Bushnell, "Stability Analysis of Networked Control Systems," *IEEE Transactions on Control Systems Technology*, vol. 10, no. 3, pp. 438–446, May 2002.

[2] Y. Xu and J. Hespanha, "Optimal Communication Logics in Networked Control Systems," in *CDC*, 2004.

[3] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty, "Constraint-Driven Synthesis and Tool-Support for FlexRay-Based Automotive Control Systems," in *CODES+ISSS*, 2011.

[4] F. Zhang, K. Szwaykowska, W. Wolf, and V. J. Mooney, "Task Scheduling for Control Oriented Requirements for Cyber-Physical Systems," in *RTSS*, 2008.

[5] R. Castane, P. Mart, M. Velasco, and A. Cervin, "Resource Management for Control Tasks Based on the Transient Dynamics of Closed-Loop Systems," in *ECRTS*, 2006.

[6] S. Samii, P. Eles, Z. Peng, P. Tabuada, and A. Cervin, "Dynamic Scheduling and Control-Quality Optimization of Self-Triggered Control Applications," in *RTSS*, 2010.

[7] A. A. Martinez, R. Majumdar, I. Saha, and P. Tabuada, "Automatic Verification of Control System Implementations," in *EMSOFT*, 2010.

[8] E. Feron and F. Alegre, "Control Software Analysis, Part I Open-Loop Properties," *CoRR*, vol. abs/0809.4812, 2008.

[9] G. Weiss and R. Alur, "Automata Based Interfaces for Control and Scheduling," *HSCC*, 2007.

[10] P. Kumar, D. Goswami, S. Chakraborty, A. Annaswamy, K. Lampka, and L. Thiele, "A Hybrid Approach to Cyber-Physical Systems Verification," in *DAC*, 2012.

[11] S. Chakraborty, L. Phan, and P. Thiagarajan, "Event Count Automata: a State-Based Model for Stream Processing Systems," in *RTSS*, 2005.

[12] R. Alur and G. Weiss, "Regular Specifications of Resource Requirements for Embedded Control Software," in *RTAS*, 2008.

[13] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in Property Specifications for Finite-State Verification," in *ICSE*, 1999.

[14] L. de Moura, S. Owre, H. Rue, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3114, pp. 251–254.

[15] E. Wandeler and L. Thiele, "Real-Time Calculus (RTC) Toolbox," `http://www.mpa.ethz.ch/Rtctoolbox`.