Characterizing Feedback Signal Drop Patterns in Formal Verification of Networked Control Systems

Dip Goswami[1], Samarjit Chakraborty[1], Purandar Bhaduri[2] and Sanjoy K. Mitter[3]
[1]Institute for Real-Time Computer Systems, TU Munich, Germany
[2]Department of Computer Science & Engineering, IIT Guwahati, India
[3]Laboratory for Information and Decision Systems, MIT, USA

***Abstract*— In order to obtain resource efficient implementations of control loops on embedded platforms, recently there has been a renewed interest in studying stability and various other quality-of-control (QoC) metrics in the presence of control message drops. Towards this, different methods have been proposed to quantify the impact of message drops on stability and control performance. In this paper we will survey these techniques and clarify the relationship between them. Given a drop pattern that satisfies stability and specified QoC constraints, it is important to check whether an implementation platform satisfies this pattern. In other words, whether the control loop in question may be implemented on this platform. Given an architecture, we will also show how certain notions of expressing drop patterns are easier to verify compared to others.**

I. Introduction

Many embedded control systems are implemented in a distributed fashion with different sensing, control and actuation tasks being mapped onto different processing units that communicate over a network (see Fig. 1). This setup has been extensively studied in the domain of *networked control systems*, where the (especially wireless) network characteristics such as delay, jitter and packet loss have been incorporated in the controller design [1], [2], [3], [4], [5], [6]. In such settings, the relationship between control performance and network bandwidth has also been extensively studied [7], [8]. In more recent times, several studies have focused on controller design with wireline networks – such as those in automotive electronic architectures like CAN and FlexRay – and on how the controller and the network may be *co-designed* [9], [10], [11], [12]. The *network design* in such cases include mapping of tasks to processors (or *electronic control units*/ECUs in the automotive context) and determining the scheduling parameters of the network. Such studies are strongly connected to a basic question in *model-based design*, i.e., *how can we ascertain that model properties are preserved in the implementation*? In the context of embedded control systems, control design is usually done at a high level of abstraction using models like Simulink and Stateflow. At this level, using both analysis and simulation, it is ensured that the designed controllers satisfy the required stability and quality-of-control (QoC) constraints. However, to ensure that these constraints continue to be satisfied in the final implementation, the implementation architecture needs to be suitably designed, which includes, e.g., scheduling the computation tasks and control messages appropriately.

As implementation platforms or embedded systems are becoming more complex, in many cost sensitive domains, explicitly taking into account feedback signal drops and delays become important even in the case of wireline networks. This is because designing systems that ensure *all* control messages are reliably delivered within their deadlines severely increases the cost of the system. Further, computing safe estimates on worst-case message delays for complex communication protocols (like FlexRay) might introduce a significant amount of pessimism and hence increase costs. Needless to mention, in the case of wireless networks message delays and drops are even more unavoidable. When designing control systems by taking into account feedback signal drops and delays, there are three important questions that need to be answered:

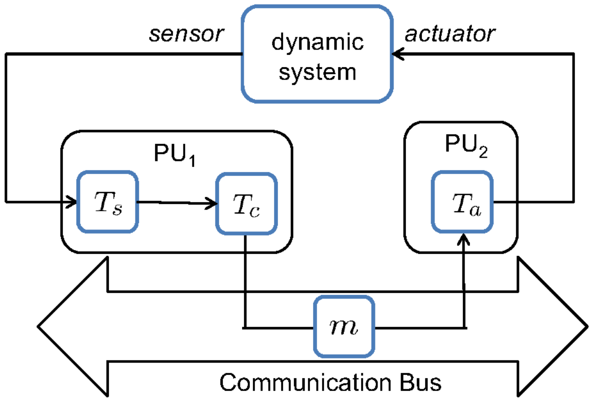


Fig. 1. Distributed control application

1) How should the impact of feedback signal drops and delays on stability and QoC be quantified and computed?
2) How should the timing characteristics of the implementation platform be computed?
3) How can we check whether the timing characteristics of the platform satisfy the signal drop thresholds such that the specified stability and QoC constraints are not violated?

In this paper we discuss these three questions, survey previous work that has been done to answer them and study the relationships between them, e.g., how does the characterization of tolerable signal drops influence the complexity of checking whether a given controller may be implemented on a given platform while satisfying specified stability and QoC constraints?

The rest of this paper is organized as follows. In Section II, we outline the basics of implementation of distributed feedback control system on embedded platforms. In the following section we describe different characterizations of signal drops that has been studied in the literature and the relationship between them. Such characterization is the main focus of this paper. For the purpose of this discussion we do not distinguish between signal drop and delay (i.e., a "delayed" signal – which is elaborate later – is assumed to be dropped). In the following three sections (Section. IV, V-A and V-B), we summarize the state-of-the-art in the directions of three main research questions above. Finally, we conclude by sketching the relationship between the characterization of signal drops and the complexity of the checking problem.

II. Discretized feedback control system

A feedback control system aims to achieve the desired behavior of a dynamical system by applying an appropriate input to the system. In a dynamical system, the relation between inputs and outputs of the system is modeled by a set of differential equations, called the *state space model*. Some common examples of dynamical systems are the variation of DC motor speed with respect to the applied terminal voltage and the adaptive cruise control system of automobiles. In a digital implementation with a constant sampling period h, a linear dynamical system is modeled by the following set of difference equations,

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

where $x[k]$ is the $n \times 1$ vector of *state variables* and $u[k]$ is the *control input* to the system at the kth sampling instant. $D[k]$ is the impulse disturbance which will be explained later. (A, B, C) are the system matrices used to model the dynamic behavior (they are usually computed by system identification techniques). A feedback loop involves three sequential operations:

- *measure* the states $x[k]$,
- *compute* the input signal $u[k]$ and,
- *actuate* the system by $u[k]$.

The task of a control designer is to compute $u[k]$ (control law) such that $y[k] \rightarrow r$ (r is the reference signal) is achieved. Towards this, the control law utilizes the available feedback signals and a general form of a control law is known as *state-feedback controller*,

$$u[k] = K \cdot x[k]. \tag{2}$$

A. Distributed implementation

In the distributed implementation of a control application, the whole function has to be partitioned into several application tasks and mapped onto different processing units (PUs). In general, a control application is *partitioned* into three categories of application tasks (see Fig. 1): (i) the sensor task T_s – reading $x[k]$ from sensors, (ii) the controller task T_c – computing $u[k]$ using (2) and, (iii) the actuator task T_a – applying $u[k]$ to the dynamical system. In a digital control implementation T_c and T_a are typically time-triggered periodic tasks. Often, the period of T_c and T_a is exactly 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 sensors tasks have to constantly monitor some states of the control plant and is therefore interrupt-driven. T_s can be triggered many times within a period with a negligible execution time of each instance. Further, the application tasks are *mapped* into different PUs and communicate over a bus. Fig. 1 shows an example of task mapping where T_s and T_c are mapped onto one PU, T_a is mapped onto the other and the control input $u[k]$ is sent as message m.

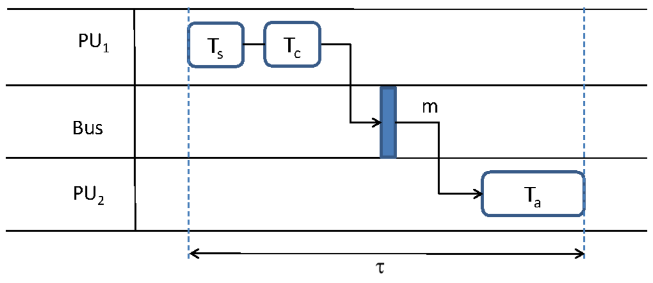


Fig. 2. Timing diagram of an implementation shown in Fig.1

B. Sensor-to-actuator delay

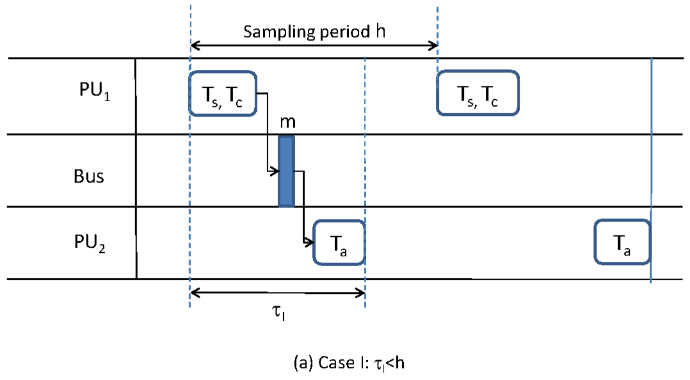
The time interval between *measure* and *actuate* is known as the *sensor-to-actuator* delay τ, which is generally characterized by the delay between the triggering of the sensor task T_s and the finishing time of the actuator task T_a. Utilizing the task partition and mapping described above, one important factor of the sensor-to-actuator delay is the interplay between the schedules of the application tasks and the message schedule on the communication bus. An example of possible task and message schedule is shown in Fig.2.

III. Feedback delay and signal drop

The sensor-to-actuator delay τ is often referred to as *feedback delay* or *delay* in this context. From Fig.2, it can be seen that the feedback signal has to go through a series of executions, i.e., $T_s \rightarrow T_c \rightarrow Bus \rightarrow T_a$. The delay τ can be computed by the time required to complete the entire execution sequence. Depending on the triggering patterns of tasks T_s, T_c and T_a as well as the message schedules on the bus, the exact delay value τ varies. There are two obvious possibilities (Fig. 3).

Case I: In this case, the triggering patterns of tasks T_s, T_c and T_a are 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 expected to be finished by $\tau_I < h$ (see Fig. 3(a)). Therefore, the *deadline* τ_I for the feedback signal is less than one sampling interval in such a choice of task/message schedules. When $\tau_I << h$, the delay is assumed to be *zero* and the control law (2) is therefore *realizable* when the feedback loop meets deadline τ_I.

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 expected to be finished by $\tau_{II} \approx h$ (see Fig. 3(b)). Therefore, the *deadline* τ_{II} for the feedback signal is approximately equal to the sampling period h for such a choice of task/message schedules. Since the feedback signal is delayed by one sampling interval, the



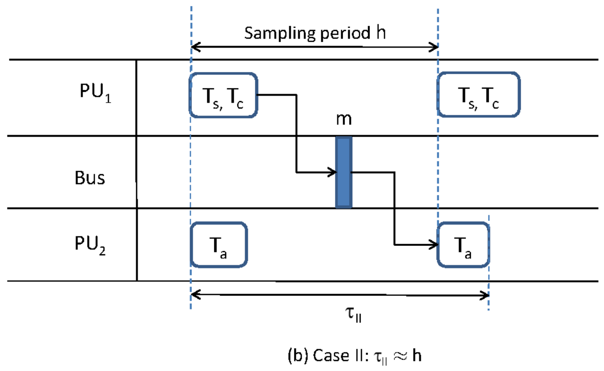


Fig. 3. Timing requirements for the control messages: Case I and Case II

control law (2) is not realizable and therefore, it is modified as follows,

$$u[k] = K \cdot x[k-1]. \tag{3}$$

In this case, the control law (3) 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., $T_s \to T_c \to Bus \to T_a$, is associated with a deadline (τ_I and τ_{II}). Since the tasks T_s, T_c and T_a 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 in Fig. 1 might get delayed and reach PU_2 after T_a is already triggered. Such timing scenario causes a violation of deadline, 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 loops need to be computed and it should essentially be within the deadlines. 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 very conservative and happens rarely.

A. Closed-loop system

Based on the above cases, the control law needs to be appropriately adapted. For example, for the Case I in Fig. 3(a), the following control law is adapted in [13], [14],

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

Similarly, for the Case II in Fig. 3(b), the following control law is adapted in [9], [15],

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

Of course, a more general case would be to consider deadline of any length including multiple sampling periods [16].
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 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 (3). Both for cases I and II, the closed-loop system becomes,

$$x[k+1] = A_\sigma x[k], \tag{4}$$

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 *delay pattern*, the closed-loop system keep switching between A_c and A. Over a duration of l sampling periods, the closed-loop system is given by,

$$x[k+l] = A_{\sigma_{k+l}} \cdots A_{\sigma_{k+1}} A_{\sigma_k} x[k]. \tag{5}$$

The requirements from control side mainly deals with stability and performance of the closed loop system (5). Each element of system matrix $A_{\sigma_{k+l}} \cdots A_{\sigma_k}$ results from either meeting or violating the deadlines.

IV. Control requirements: specifications

In general, the control specifications are formulated based on requirements on stability as well as performance of the closed-loop system (5). In the following, we summarize the state-of-the-art on the available analysis tools for this purpose.

A. Stability-based requirements

Clearly, the closed-loop system (5) 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 maximum frequency of allowed deadline violation with guaranteed stability of system (5)?*
A general answer to the above question follows from [2] with slightly modified formulation of control strategy,

$$u[k] = \begin{cases} Kx[k] & \text{, if } \tau \le \tau_I \\ Kx[k-1] & \text{, if } \tau > \tau_I \end{cases}, \tag{6}$$

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

Theorem 1. (Stability condition) *Consider the closed-loop system (5) with control input (6) and assume that A_1 is Schur stable.*

- *If the open-loop system A is marginally stable, the system (5) is exponentially stable for $0 < r \le 1$.*

- *If the open-loop system A is unstable, the system (5) is exponentially stable for*

$$\frac{1}{1-\gamma_1/\gamma_2} < r \leq 1 \tag{7}$$

where r is the rate of meeting deadline over infinite horizon and

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

where λ_{max} denotes the eigenvalue of the corresponding matrix with the maximum absolute value. The above theory holds for any value of l in (5). For example, suppose we obtain $r = 0.1$ for a given choice of parameters in (5). With $l = 1000$, the above theorem implies that *any* 100 samples are allowed to violate their deadlines with guaranteed exponential stability. On 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., guarantee on settling time). Further, the control law (6) 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 over pessimistic to design an architecture with worst-case delay estimation.

A relatively more structured requirement on stability was introduced in [16] where the systems are required to be stable with a desired *stability margin*. As discussed, not all feedback messages suffers 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. 3(b). The control requirement is then represented as *delay frequency metric* which is defined as follows.

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 that violates deadline (which has a system matrix A) is followed by at least three samples (which have system matrix A_c) where deadline is respected. Hence, the overall system can be represented as follows,

$$x[k+2+n_i+n_j] = AA_c^{n_i} \times AA_c^{n_j} x[k], \tag{8}$$

where $n_i, n_j \geq 3$. The stability is assured with a given stability margin by showing the existence of common quadratic Lyapunov function (CQLF) [17] between systems $AA_c^{n_i}$ and $AA_c^{n_i}$ for all combinations of n_i and n_j.

B. Performance-based requirements

In a flurry of recent works [13], [14], [15], the control requirements are specified using a notion of *exponential stability*,

$$\frac{\|x[k+l]\|}{\|x[k]\|} < \epsilon, \tag{9}$$

where $\|.\|$ denotes 2-norm. That is, to ensure that the plant remains exponentially stable, any error must be reduced by 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 [18].

Coming back to the control requirement on exponential stability (9) and considering the closed-loop system (5), we obtain the following relation,

$$\begin{aligned} x_{k+l} &= A_{\sigma_{k+l}} \cdots A_{\sigma_k} x_k, \\ \Rightarrow \frac{\|x_{k+l}\|}{\|x_k\|} &\leq \left\|A_{\sigma_{k+l}} \cdots A_{\sigma_k}\right\| \\ &\Rightarrow \left\|A_{\sigma_{k+l}} \cdots A_{\sigma_k}\right\| < \epsilon. \end{aligned} \tag{10}$$

In other words, the exponential stability requirement can be re-written as follows (see [14], [13], [15]),

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

which essentially 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. The works in [14], [13] construct a Büchi automaton to represent this language.

Hence, the control requirement on exponential stability boils down to a set of acceptable patterns of occurrence of A and A_c that meets the condition (10). The computation of such a set of acceptable patterns can be done by a brute-force search as in [14], [13], but it becomes tedious to verify them pattern-by-pattern. To avoid this, a deadline constrain is introduced in [15]:

Definition 2 ((f, H)-firm deadline)**.** *A stream of control messages is said to fulfill the (f, H)-firm with respect to period h if at least f out of any H consecutive samples 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, one can evaluate all the patterns and separate those that do not fulfill $ES(5, 0.75)$ in the above example. Further, one can exclude all the patterns by requiring the 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 five samples at most two messages can have delay $\tau > h$ (Case II in Fig. 3(b)). Therefore, the set of all acceptable patterns is represented by number of such (f, H)-soft deadlines where $H \leq l$.

V. Architecture modeling and verification

Once the allowable feedback signal drop pattern has been quantified, the next step is to capture the timing characteristics or behaviors of the implementation platform, followed by checking if these behaviors constitute a subset of the behaviors that may be tolerated by the controller. This then determines whether the controller may be *implemented* on this platform.

A. Timing properties of architectures

Towards characterizing timing behaviors of the architectures, various techniques have been proposed in the real-time and embedded systems, as well as in the formal methods literature. Analyzing embedded platforms in the particular context of implementing distributed controllers have been

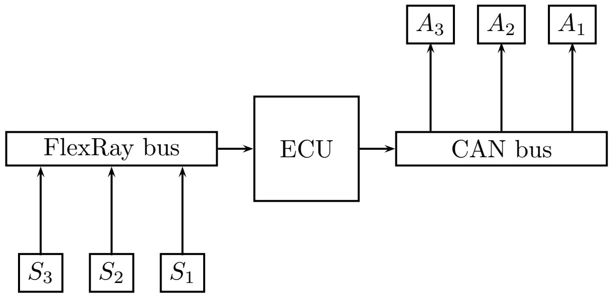


Fig. 4. Example architecture.

studied in [15], [16]. In [16], the Real-Time Calculus [19] modeling formalism has been combined with the Uppaal [20] modeling and verification environment. 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. Let these be 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 be denoted by $\beta^u(\Delta)$ and $\beta^l(\Delta)$ respectively. These bounds may then used to compute the worst-case delays suffered by messages, in addition to properties like buffer requirements. Exactly similar techniques also apply to computation (rather than communication) resources and number of task executions (rather than messages).

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 [19] for details). Fig. 4 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 delay 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:

$$\beta^{\text{end-to-end}} = \beta^{\text{FR}} \otimes \beta^{\text{ECU}} \otimes \beta^{\text{CAN}}, \tag{11}$$

where $\otimes$ is the convolution operation as defined in Real-Time Calculus [19]. 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 [16] the service bounds or timing behaviors of the resource were transformed to a timed automata model in Uppaal.

A similar technique was adopted in [15], where a communication resource was modeled using a *Time-Stamped Event Count Automata* (TS-ECA). An *Event Count Automata* (ECA) [21] is given by the tuple

$$\mathcal{A} = (S, s_{in}, X, V_{in}, Inv, \rho, \rightarrow) \tag{12}$$

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 where
$$\Phi(X) = x \leq c | x < c | x \geq c | x > c | \varphi_1 \wedge \varphi_2$$
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 are strings of integers that in our setting denote the arrival patterns of control messages or sensor readings. For example, in the case of an *arrival ECA*, 201... denotes an arrival pattern with two messages arriving in the first time interval, no messages in the second time interval and one message in the third interval. In the case of a *service ECA* representing a communication resource, it may denote the number of control messages transmitted during three consecutive time intervals. It has been shown in [21] that bounds on message arrivals and resource availability (i.e., $\alpha(\Delta)$ and $\beta(\Delta)$ respectively) as used in Real-Time Calculus can also be represented as corresponding ECAs.

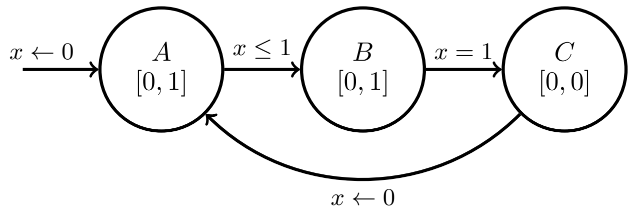


Fig. 5. Periodic with jitter arrival ECA ($p = 3, j = 2$)

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 an ECA representing the arrival pattern of a message stream which is periodic and has some jitter (see Fig. 5), this corresponds to state A and the only count variable $x = 0$. From there the ECA can make moves of the form $(s, V) \overset{k}{\Rightarrow} (s', V')$. 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 on the transition ($x \leq 1$), and possible invariant conditions (none in the states of the ECA in Fig. 5). 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.

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 the guard $x = 1$ on transition (B, C) guarantees that it occurred beforehand. For a more rigorous description of ECA semantics, please refer to [21].

While ECAs as described were augmented with time stamps in [15] – with the resulting model being referred to as Time-Stamped Event Count Automata – in order to keep track of the delays suffered by individual messages.

B. Control/Architecture Co-Verification

As seen in the previous subsections, the timing behaviors of an implementation platform or architecture may be modeled as an automaton. In particular, the language of such automata – as seen with the example from ECAs – represent different timing behaviors of the architecture. The next question that needs to be answered is how can we check whether these behaviour constitute a subset of the behaviors that may be tolerated by the controller in order to meet given stability and QoC constraints?

There are two broad categories of approaches towards this. The first follows conventional the *model checking* approach [22]. Here, the automaton corresponding to the implementation platform is model-checked to verify whether it violates any of the acceptable behaviors of the controller (in terms of feedback signal drop patterns). Such acceptable behaviors of the controller were transformed to *Linear Temporal Logic* (LTL) formula in [15], followed by checking whether the TS-ECA corresponding to the architecture satisfies this formula (using model checking in SAL). [16] avoided the explicit transformation into LTL but instead used an *observer automaton* to check for timing property violations in the architecture model.

The second category of approaches rely on *interface theories* [23], [24]. Here, all possible signal drop patterns that still satisfy stability and control performance constraints may be represented as a language, say $L_{controller}$. Similarly, the timing behaviors of the implementation platform – capturing sequences of messages that are delivered within the specified deadline, as described above – may be represented by the language $L_{architecture}$. 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 $L_{architecture} \subseteq L_{controller}$. Satisfaction of this inclusion implies that the controller may be implemented on the given architecture. The work reported in [13], [14] followed this line of approach, but did not explicitly model the timing properties of the architecture. It rather characterized the stability properties of the controller in language-theoretic terms.

The complexity of the above-mentioned co-verification problem heavily depends on the exact characterization of the feedback signal drop patterns that were discussed in Section IV. The characterization defined by Theorem 1 requires checking the satisfaction of the inequality (7) over an infinite horizon and hence cannot be realized by an automaton with finitely many states. On the other hand, the *tighter* characterization as defined by inequality (9) – that takes into account control performance in addition to stability – requires a check over a bounded horizon and is therefore easier from the perspective of verification.

VI. Concluding remarks

In this paper, we have studied different characterizations of feedback signal drop patterns in embedded control systems. We then looked at how the timing behaviors of implementation architectures can be modeled and finally we briefly discussed different possibilities of checking whether the architecture timing behaviors match those allowed by the controller subject to satisfaction of stability and performance constrains.

References

[1] H. Y. G. C. Walsh and L. G. Bushnell, "Stability analysis of networked control systems," *IEEE Trans. on Control System Technology*, vol. 10, no. 3, pp. 438–446, 2002.

[2] W. Zhang, M. S. Branicky, and S. M. Phillips, "Stability of networked control systems," *Automatica*, vol. 21, p. 8499, 2001.

[3] M. Pajic, S. Sundaram, G. J. Pappas, and R. Mangharam, "The wireless control network: A new approach for control over networks," *IEEE Transactions on Automatic Control*, vol. 56, no. 10, pp. 2305–2318, 2011.

[4] R. Alur, A. D'Innocenzo, K. H. Johansson, G. J. Pappas, and G. Weiss, "Compositional modeling and analysis of multi-hop control networks," *IEEE Transactions on Automatic Control*, vol. 56, no. 10, pp. 2345–2357, 2011.

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

[6] X. Wang and M. Lemmon, "Event-triggering in distributed networked control systems," *IEEE Transactions on Automatic Control*, vol. 56, no. 3, pp. 586–601, 2011.

[7] S. Tatikonda and S. K. Mitter, "Control under communication constraints," *IEEE Trans. Automat. Contr.*, vol. 49, no. 7, pp. 1056–1068, 2004.

[8] ——, "Control over noisy channels," *IEEE Trans. Automat. Contr.*, vol. 49, no. 7, pp. 1196–1201, 2004.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] M. Kauer, S. Steinhorst, D. Goswami, S. Reinhard, 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).*

[16] 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.

[17] O. Mason and R. Shorten, "On common quadratic Lyapunov functions for stable discrete-time LTI systems," *IMA Journal of Applied Mathematics*, vol. 69, no. 3, pp. 271–283, 2002.

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

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

[20] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *STTT*, vol. 1, no. 1-2, pp. 134–152, 1997.

[21] 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)*, 2005, pp. 87–98.

[22] C. Baier and J.-P. Katoen, *Principles of model checking.* MIT Press, 2008.

[23] L. de Alfaro and T. A. Henzinger, "Interface-based design," in *Engineering Theories of Software-intensive Systems, Marktoberdorf Summer School, NATO Science Series*, 2004.

[24] ——, "Interface theories for component-based design," in *EMSOFT*, 2001, pp. 148–165.