Communication by Sampling in Time-Sensitive Distributed Systems *

Albert Benveniste† Benoît Caillaud Luca P. Carloni Paul Caspi
Alberto L. Sangiovanni-Vincentelli Stavros Tripakis

ABSTRACT

In *time-sensitive systems* writing to and reading from the communication medium is on a purely time-triggered but asynchronous basis. Writes and reads can occur at any time and the data are stored and sustained until overwritten. We study how to maintain data semantics when the duration of the actions change from specification to implementation.

In doing so, we rely on *tag systems* formerly introduced by the authors. The flexibility of tag systems allows handling the problem in a formal, yet tractable way.

Categories and Subject Descriptors: C.3.3 [Special-purpose and application-based systems]: Real-time and embedded systems.

General Terms: Theory.

Keywords: Heterogeneous reactive systems, tagged systems, distributed deployment, scheduling.

1. INTRODUCTION

Complex real-time control systems pose serious challenges to the design community as they require with increased frequency a distributed implementation. In a distributed implementation, choosing the communication architecture is often the most critical design step as communication characteristics ultimately determine efficiency and correctness of the design. Latency and blocking behavior of communication may introduce unforeseen effects on the behavior of the overall system. Choosing a communication architecture that can be formally analyzed and/or guaranteed to maintain the ideal behavior of the system is an active research area and of great industrial interest.

A system designer typically performs the following sequence of steps, as part of the overall design process:

1. System issues generally set the most important constraints on the overall system architecture. This is why the latter is defined first (even if it may need to be revised later), as a set of allowed architectural components, *the platform*, including communication structures and composition rules. This delimits the space of architectural exploration. The components include processors, communication media, and peripherals as well as middleware such as compilers, operating systems and device drivers that determine the way the hardware is used by the application software.

2. The system functions are given using mathematical models or executable code. These specifications may involve a combination of discrete mode changes and continuous signal processing and control functions. The *functional* specifications do not carry the notion of physical quantities such as time and power. They are usually analyzed formally or simulated extensively in closed loop to uncover potential algorithmic design errors.

3. A set of cost and constraint functions involving the quantities of interest for determining the *quality and correctness* of the design is given. Among those, timeliness requirements to ensure that the plant is controlled at the proper bandwidth and reaction time are of particular interest for control system design.

4. When the implementation platform includes one or more processors, functions are in general packaged into *tasks* which must be scheduled and/or distributed.

When the resources of the architecture are limited, the distribution of the tasks to the architectural elements requires a careful scheduling and assignment step. For example, if two or more concurrent functions are assigned to the same sequential processing element, we need to determine an order according to which the functions must be executed. By the same token, if a number of communication requests are made to a limited interconnect structure such as a bus, an arbitration protocol determines the order with which the communication requests are served. In a realistic scenario,

†A. Benveniste and B. Caillaud are with INRIA/IRISA, Campus de Beaulieu, 35042 Rennes cedex, France, {Albert.Benveniste,Benoit.Caillaud}@irisa.fr; L.P. Carloni is with the Department of Computer Science of Columbia University in the City of New York, NY 10027, USA, luca@cs.columbia.edu; P. Caspi is with CNRS/Verimag, Centre Equation, 2, rue de Vignate, F-38610 Gieres, France, Paul.Caspi@imag.fr; A. Sangiovanni-Vincentelli is with U.C. Berkeley, Berkeley, CA 94720, USA, alberto@eecs.berkeley.edu; S. Tripakis is with CNRS/Verimag and Cadence Berkeley Labs, 1995 University Ave, Suite 460, Berkeley, CA 94704, USA, Stavros.Tripakis@imag.fr.

*This research was supported in part by the European Commission under the projects ARTIST2, IST-004527, by the NSF ITR CHESS, and by the GSRC.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

architectural elements do take time to compute and to serve communication requests. Satisfying timeliness constraints requires clever assignment of functions to computing and communication elements. In addition, if the scheduling algorithm is not carefully designed, we may run into a deadlock situation that would impact in a catastrophic manner system behavior.

Schedulability analysis, a very hard problem in the general case, aims at answering questions related to the correct behavior of the implementation when compared to the functional specification. Because of its conservative nature and of its computational complexity, engineers are used to performing approximate analysis but in doing so, there is no guarantee that the final implementation would be always executing correctly, a very serious problem indeed for safety critical systems. Since the duration of tasks may vary depending on the execution platform characteristics, the functional semantics can be lost, unless rigid policies such as TTA or the one advocated by Giotto [6] are used.

An alternative approach to schedulability analysis as advocated by Kopetz with its Time-Triggered Architecture (TTA) [7] is to use *physical time* to coordinate communication allowing the implementation of the real-time periodic synchronous model in a distributed way. Using this approach, correctness of a distributed implementation can be analyzed rigorously with formal techniques. However, this approach carries cost and timing penalties that at times are non acceptable for the application considered. For this reason, there has been growing interests in less constrained architectures such as the Loosely Time-Triggered Architecture (LTTA) used in the aerospace industry and studied in [2, 4, 5, 8].

All modern real-time distributed architectures share the viewpoint that communications should not be blocking. One way to achieve this is by triggering actions and communications by *dates,* thus resulting in what we call *time-sensitive systems.* Recent work [1, 9, 10, 11] has considered, for these architectures, the problem of maintaining proper functional semantics while performing task scheduling.

Tracking how functional semantics may be skewed in this context requires a formal approach that captures causality dependencies and logical delays across the tasks of the functional specification as well as the resource availability and effective execution times that characterize its implementation on a given platform. Ultimately, this translates into the problem of guaranteeing that all the individual inter-task data exchanges occurring in the final implementation are consistent with those defined in the original specification. We address the hybrid nature of this problem, more formally defined Section 3, using the framework of tag systems, which were proposed to cope with this kind of heterogeneity [2, 3] and which are briefly reviewed in Section 2. Specifically, we first deal with a simplified version of the problem (Section 4) which is applicable only to ideal system implementation. The simplified version assumes that the starting time of the execution of each task always coincides with the instant of its activation.

Then, in Section 5 we present our results for the more realistic case where some time may actually pass between the activation instant of a task and the starting of its execution. For both scenarios, we formally derive an operational protocol that guarantees the preservation of data semantics as we move from the specification to a particular implementation. This is accomplished through the insertion of a proper number of *compensating logical delays* in the inter-task communication channels. A subtle but important point is that to perform this operation correctly and optimally, we need to account for the possible presence of *original logical delays* in the specification. Furthermore, sometimes it may be necessary to revisit the original specification in order to correct it by increasing the "delay budget" between some tasks to match the constraint imposed by a given implementation platform. A practical contribution of this paper is to provide formal means to guide the designers through this process.

2. BACKGROUND ON TAG SYSTEMS

Tag systems will provide us with the adequate framework for our study. We now collect a small subset of the results from [2]—we shall not need compositionality aspects of this framework.

DEFINITION 1 (TAG STRUCTURE). *A* tag structure *is a preorder* $(\mathcal{T}, \leq)$*, where* $\mathcal{T}$ *is called the set of* tags.

We assume an underlying set of variables $\mathcal{V}$ with domain $\mathcal{D}$. All systems will have $\mathcal{V}$ as set of variables but will indeed involve only finitely many of them, whereas the behavior attached to other variables will remain free. A $\mathcal{T}$*-signal* is a finite or infinite sequence of pairs $(\tau, x) \in \mathcal{T} \times \mathcal{D}$. Given a $\mathcal{T}$-signal $\mathbf{s}$ and a natural number $n \in \mathbf{N}$, $\mathbf{s}(n)$ denotes the n-th element in the sequence defined by $\mathbf{s}$. The set of $\mathcal{T}$-signals is denoted by $\mathcal{S}_{\mathcal{T}}$. A *behavior* is an element:

$$\sigma \quad \in \quad \mathcal{V} \mapsto \mathcal{S}_{\mathcal{T}}, \tag{1}$$

meaning that, for each $v \in \mathcal{V}$, the n-th occurrence of v in behavior σ has tag $\tau \in \mathcal{T}$ and value $x \in \mathcal{D}$. For σ a behavior, an *event* of σ is a tuple $e = (v, n, \tau, x) \in \mathcal{V} \times \mathbf{N} \times \mathcal{T} \times \mathcal{D}$ such that $\sigma(v)(n) = (\tau, x)$. We shall require that, for any two events $e = (v, n, \tau, x)$ and $e' = (v, n', \tau', x')$ belonging to the same behavior, if $n \leq n'$, then $\tau \leq \tau'$. Thus we can regard behaviors as tuples of signals, where signals are totally ordered chains of events associated with a same variable.

DEFINITION 2 (TAG SYSTEMS). *A* tag system *is a pair* $P = (\mathcal{T}, \Sigma)$*, where* $\mathcal{T}$ *is a tag structure, and* Σ *a set of behaviors.*

By abuse of notation, we shall write $\sigma \in P$ to mean $\sigma \in \Sigma$, *i.e.,* that σ is a behavior of tag system P.

3. PROBLEM FORMULATION

Interactive Tasks

The system considered in this paper consists of a set of possibly concurrent interactive processes called *tasks*, $V = \{v_1, v_2, \ldots, v_k\}$. Task occurrences are characterized by three dates:

the dates a of activation, s of start, and t of termination.

While the task activation occurs at will (i.e., is controlled by the external environment), the execution of a task can start only when certain conditions are satisfied, *e.g.,* the processor is ready to handle it. We do not assume periodicity nor any particular activation policy. Tasks interact by exchanging data after they have completed their execution. Tasks

may or may not take time. The different occurrences of a same task are totally ordered by their start date, but may otherwise overlap, meaning that a new occurrence may start prior to the previous one having completed (i.e., we do not assume that tasks are serialized).

Communication Model

Each task v has a *channel* $\mathbf{chan}_v$ where it outputs its results. When task v writes in $\mathbf{chan}_v$, the written value is sustained and available for reading until the next writing by v occurs. Reading of the current content of $\mathbf{chan}_v$ can be performed by any task at any time. Reads and writes occur asynchronously and are therefore non blocking.

REMARK 1. This communication abstraction that we call *communication by sampling* is a widely used communication scheme in distributed system design, especially for time-sensitive architectures. This scheme has not been given by the academic community the attention it deserves though it is a very natural one. For instance, readers familiar with Simulink diagrams would easily recognize that the communication scheme depicted in Figure 1 appears very frequently in their designs. In this diagram, the producer block writes "value" at the rate of its triggering clock and the consumers sample this "value" at the rates of their respective triggering clocks. These clocks may be periodic but also non periodic and then correspond to event triggered systems.

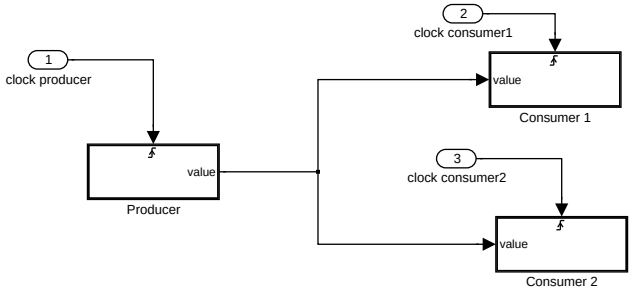

Figure 1: Communication by sampling in a Simulink diagram

REMARK 2. An important consequence of using this model of communication is that tasks are activated on a *local* basis, regardless of the environment of the task.

Of course, which data are exchanged between the different task occurrences critically depend on *when* reads and writes occur. If we refer to the design process described in the introduction, then the functional specification of the system of interacting tasks, together with the constraints, determine logical read and write dates associated to each task occurrence. When the implementation process has ended, the dates associated to the tasks are determined by the characteristics of the architectural elements *e.g.,* by task execution times and communication delays. Then the problem we are addressing in this paper is to ensure that data exchanges specified at the abstract design stage are consistent with the data exchanges occurring at the implementation stage, *i.e.,* we are looking for a robust implementation scheme that does not affect original specifications.

Design Scenario

While our approach can also handle the preservation of data semantics between two asynchronous executions when the task (or communication) duration changes, the following design scenario is of particular interest and will serve as a support for our presentation:

1. At the functional specification stage, we assume ideal, atomic tasks, of zero duration *i.e.,* we assume that the functional design paradigm is fully synchronous.

2. When implemented, tasks take some non zero duration and may become non atomic; this is often referred to as the "asynchronous" execution mode. This may change the actual values read by the tasks. If this happen, we say that *data semantics* is not preserved.

4. THE "SIMPLE TASK" PROBLEM

To ease the presentation of our approach, we begin with a simple case of system of interacting tasks where

the date of activation and start coincide, *i.e.,* $a = s$

Hence, each task is characterized by a pair of dates (s, t). In the next section, we deal with the more general case of pre-emptable tasks.

4.1 Task systems as tag systems

We consider a finite set $V \subseteq \mathcal{V}$ of *task names* or *tasks* for short; tasks are generically denoted by the symbols v or u. Call *events* the successive occurrences of a task. Thus events e have the form

$$\begin{array}{rcl} e & = & (v, n, \tau, x) \\ & = & \text{(task, occurrence, tag, output data)} \end{array} \quad (2)$$

Tag structures

Now we must define what the tag τ in (2) should be.

Task dates. We already know that each event possesses a pair (s, t) of start and termination dates, such that $s \leq t$. We shall therefore use the following tag structure of *task dates*:

$$\mathbf{R}^2_> =_{\text{def}} \{(s, t) \in \mathbf{R}^2 \mid s \leq t\}. \quad (3)$$

where $\mathbf{R}$ is the set of real numbers augmented with $-\infty$. $\mathbf{R}^2_>$ is equipped with the restriction of the product order defined on $\mathbf{R}^2$:

$$(s, t) \leq (s', t') \text{ iff } s \leq s' \text{ and } t \leq t'$$

Note that this does not require that $t \leq s'$.

Usage tag. Next, we need to indicate:

1. which other tasks outputs the considered event uses for its completion, and

2. what the associated logical delays are.

Regarding 1, we consider the binary tag $b : V \mapsto \{0, 1\}$, defined by

$$b(u) \quad = \quad 1 \text{ if } v \text{ uses } u, \text{ otherwise } b(u) \; = \; 0.$$

Regarding 2, note first that logical delays are often used in the functional specification (for example, to compare two successive values of a same signal to a threshold). Logical

delays are meant to be used at Stage 1 of the design scenario of Section 3. They are typically statically defined. Logical delays are captured by tag

$$m_v \quad = \quad (m_{uv})_{u \in V : d(u)=1}$$

which specifies the logical delay in the communication from u to v, for each task u whose output is used by v (notice that m_v is a vector).

Package the pair (b, m_v) into a tag

$$d =_{\text{def}} (b, m_v) \tag{4}$$

called *usage tag,* and call D the resulting tag structure. Equip D with the trivial preorder such that $d \leq d'$ for any two d and d'. The reason for using this trivial preorder is that nothing should prevent two successive events for a same task v from using outputs from another task u in reverted order.

Data dependency. The usage tag is adequate to specify the exchange of data between tasks at specification stage (stage 1 of the design scenario of Section 3). This type of tag is, however, not appropriate to describe which data is communicated when the application is deployed over the actual distributed architecture—we already mentioned that the functional semantics is generally not robust to physical delays in time-sensitive distributed systems.

To achieve this, we need to specify which data are communicated in a more explicit way, independent from actual execution mode. This is captured by another type of tag called data dependency. Let $\mathbf{N}_{-\infty} =_{\text{def}} \mathbf{N} \cup \{-\infty\}$. Define a *data dependency* to be a map: $\delta : \mathcal{V} \mapsto \mathbf{N}_{-\infty}$, and let Δ denote the set of all dependencies:

> $\delta(v) = m$ means that the event with the considered tag uses outputs of the m-th occurrence of task v; $\delta(v) = -\infty$ means that the considered event does not use the results from v.

Equip Δ with the trivial preorder such that $\delta \leq \delta'$ holds for each pair (δ, δ').

Δ describes which data a given task occurrence uses as its inputs—assuming tasks are functions, this entirely determines the actual output of the considered task occurrence. Thus, Δ keeps track of data semantics in a rigid way, regardless of actual execution mode.

Note that, unlike the usage tags, data dependencies are not appropriate for functional specifications since their description is not finite (it explicitly uses the infinite event index $n \in \mathbf{N}$). They will, however, be essential for the derivation of our protocol and associated mathematical analysis.

Summary. Our task systems are modeled as tag systems of the form

$$P \quad = \quad (\mathcal{T}, \Sigma) \tag{5}$$

Corresponding events have the form

$$e = (v, n, \tau, x), \quad \text{where} \quad \tau = ((s,t); (b, m_v); \delta\,) \tag{6}$$

where we recall that:

- (s,t) are the start and termination dates;
- $b(u) = 1$ if v uses u, otherwise $b(u) = 0$; m_{uv} is the logical delay from u to v when completing this event;
- δ codes explicit data dependencies.

Now we are ready to formalize communication by sampling.

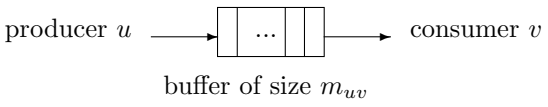

Figure 2: Illustration of Assumption 1.

Communication by sampling

This mode of communication is captured by the following assumption.

ASSUMPTION 1 (COMMUNICATION BY SAMPLING). *For any event $e = (v, n, \tau, x)$ of P of the form (6) , tag τ must satisfy the following condition:*

$$\begin{array}{lll} b(u) = 0 & \Rightarrow & \delta(u) = -\infty \ (e \text{ does not use } u) \\ b(u) = 1 & \Rightarrow & \delta(u) = n_t(u,s) - m_{uv} \end{array} \tag{7}$$

where we recall that integer m_{uv} is the logical delay in the communication from u to v when completing e, and, for $r \in \mathbf{R}$,

$$n_t(u,r) \quad =_{\text{def}} \quad \max\{k \mid t(u,k) < r\} \tag{8}$$

In (8),

- *$t(u,k)$ denotes the termination time of the kth occurrence of u in the considered behavior;*
- *$n_t(u,r)$ is the number of times task u has completed prior to current time r.*

Figure 2 illustrates Assumption 1. One can think of a buffer of size m_{uv} between the producer u and consumer v. The buffer is a FIFO and when it is full its first element (the oldest written) is removed. Thus, at each write operation, all elements are "shifted" to the right. The consumer reads the right-most (oldest) element but does not remove it.

Note that we do *not* require that, for two successive occurrences e^-, e of a same variable v, $t^- \leq s$ holds. In other words, it is possible, for a task, to start a new occurrence prior to having completed its previous one.

4.2 Semantics preserving

Clearly, variations in the duration of tasks may change the actual data tasks expose at a given date. Due to Assumption 1 regarding the MoCC, this may result in other tasks reading wrong data for their execution. The idea is to use *variable* delays to compensate for this mismatch and thus ensure the preservation of data semantics. To achieve this, we first need to formally define what we mean by "preserving data semantics".

Notation. Events have the form (6). To shorten notations, we shall write a dot for fields that are irrelevant for the point being discussed. For instance, $e = (v, ., ((s,t); .; .), .)$ focuses on v, and the pair (s,t). Also, whenever convenient, we shall denote by v_e, n_e, τ_e, etc, the components of event e.

DEFINITION 3 (DATA EQUIVALENCE). *Say that two behaviors σ and σ' over tag structure $\mathcal{T}$ are* data equivalent, *written*

$$\sigma \ \sim_{\text{data}} \ \sigma'$$

iff there exists a bijection χ, from the set of events e of σ onto the set of events e' of σ', such that

$$\begin{array}{c} e' = \chi(e) \\ \Downarrow \\ v_{e'} = v_e, n_{e'} = n_e, \text{ and } \forall u \in V : \delta_{e'}(u) = \delta_e(u) \end{array} \tag{9}$$

In particular, $\delta_e(u) = -\infty$ iff $\delta_{e'}(u) = -\infty$. In words, two behaviors are data equivalent iff they read their inputs from the same events—assuming that tasks are functions, data equivalent behaviors will output the same stream $x_1, x_2, \ldots$ of data, for each variable v, whence the term of data equivalence.

Using characterization (7,8) of the MoCC, the last statement of condition (9) rewrites as follows, for every $u \in V$ such that $\delta(u) \neq -\infty$:

$$n'_t(u, s_{e'}) - m'_{uv} \quad = \quad n_t(u, s_e) - m_{uv} \tag{10}$$

where the prime in n' refers to counting in the context of σ'. Now, rewriting (10) as

$$m'_{uv} \quad = \quad m_{uv} + n'_t(u, s_{e'}) - n_t(u, s_e)$$

reveals that m'_{uv} must vary with e' in order to compensate for the varying discrepancy $n'_t(u, s_{e'}) - n_t(u, s_e)$. Thus, whenever convenient, we shall write

$$m'_{uv}(s_e, s_{e'})$$

to explicit this fact.

With some overloading, say that two tag systems P and P' over $\mathcal{T}$ are *data equivalent,* written $P \sim_{\text{data}} P'$, iff there exists a bijection Θ between their respective sets Σ and Σ' of behaviors, such that $\Theta(\sigma) \sim_{\text{data}} \sigma$.

The problem we want to solve can now be formalized as follows. We are given a tag system P as in (5). Suppose we modify the behaviors σ of P by changing the duration of task occurrences and the logical delays as follows:

$$\begin{array}{rcl} e & = & (v, n, ((s,t);(b,m);.),.) \\ & & \text{is replaced by } e', \text{ where} \\ e' & = & (v, n, ((s,t');(b,m');.),.) \text{ where } t' \neq t, m' \neq m. \end{array} \tag{11}$$

Remark 3. Note that s and b are not changed. Reasons for this are:

1. s, the date of activation, is set locally (cf. Remark 2), and is the same for the specification and the implementation;
2. b is a rigid part of the functional specification.

Replacing every event e of P by e' as above yields a new system P'. With this modification, P' is generally not data equivalent to P.

Notation. In the sequel, we shall systematically use unprimed symbols to refer to entities from P and primed symbols to refer to the corresponding entities from P'. For σ a finite behaviour, n_u the length of the u-signal in σ, and $e = (u, n_u, ., .)$ an event, $\sigma \oplus e$ denotes the behavior σ extended with e. Finally, for e an event having termination date t and logical delay m, we denote by $e[t/t', m/m']$ the event e in which t and m have been replaced by t' and m', respectively.

Problem 1. *Regard the modification $t \mapsto t'$ as given. Can we adjust the delay m' in e' in order to recover data equivalence and thus compensate for the change in duration of events? Formally,*

$$\begin{array}{c} \text{for every } (\sigma, \sigma') \text{ such that } \sigma' \sim_{\text{data}} \sigma \\ \text{find } m' \text{ such that } \sigma' \oplus e[t/t', m/m'] \sim_{\text{data}} \sigma \oplus e \end{array} \tag{12}$$

The reason for considering the modification $t \mapsto t'$ as given is that this modification captures a change in the timing characteristics of the computing system and therefore it is not controlled. Using Assumption 1, Definition 3, and (10), Problem 1 can be reformulated as the following

Problem 2. *For event*

$$e = (v, ., ((s,t);(b,m_v);.),.)$$

as in (12), find, for each u such that v uses u, a compensating *logical delay $m'_{uv}(s,s)$ satisfying the balance equation:*

$$n'_t(u, s) - m'_{uv}(s, s) \quad = \quad n_t(u, s) - m_{uv} \tag{13}$$

Note that start dates are not changed under the substitution $e \to e'$ in (11), so that the same start date s occurs on both sides of (13). Observe that, in general, the compensating logical delay will vary with the index of the task occurrence, even if the original logical delay was fixed in the specification. By (10), (13) ensures data equivalence of e' and e and justifies replacing Problem 1 by Problem 2. We focus on solving Problem 2.

4.3 The protocol preserving data semantics

This protocol is based on formula (13). It aims at preserving the data semantics when the duration of events changes, from specification to implementation.

The protocol

We develop the protocol for the particular case where the specification P has zero-duration events, *i.e.,* for any event $e = (v, n, ((s,t);(b,m_v);.),.)$ of P,

$$s = t \text{ holds.} \tag{14}$$

When moving from specification to implementation, the duration of events becomes positive:

$$t \quad \text{is replaced by} \quad t' > t \tag{15}$$

Note that in this case, for every task u, we have $n_t(u,s) - n'_t(u,s) \geq 0$. Hence, by (13), in order to compensate for (15) we must point to a smaller buffer index when reading the value from u in order to produce a new event for v.

For $r \in \mathbf{R}$, define

$$n'_{st}(u, r) \quad =_{\text{def}} \quad n'_s(u, r) - n'_t(u, r) \tag{16}$$

Lemma 1. *Under (14) and (15), we have*

$$\forall r \in \mathbf{R} \quad : \quad n_t(u, r) - n'_t(u, r) = n'_{st}(u, r)$$

Proof. Indeed, $n_t(u, r)$ counts the number of completed occurrences up to date r, for the original system P; by (14), termination times equal start times, hence $n_t(u, r) = n_s(u, r)$. On the other hand, by (11), start times are identical for P and its modification P'; therefore, $n_s(u, r) = n'_s(u, r)$. This proves the lemma. ◇

As we have seen, solving Problem 2 amounts to constructing the compensating logical delay as in (13). Our protocol performs this incrementally.

Considering (13), while recursively extending behavior σ' as requested by Problem 1, we shall continuously maintain the following doubly indexed family of functions, for $r \in \mathbf{R}$:

$$m'_{uv}(r) \quad = \quad m_{uv} - n'_{st}(u, r) \tag{17}$$

where $v, u \in V$ are tasks such that v uses u.

PROTOCOL 1. *For each pair (v, u) of tasks, m'_{uv} is updated on-line according to the following rules ordered by decreasing priority:*

1. *Each time s_v a new occurrence for task v is started (in P' or, equivalently, in P), in completing this occurrence of v for P', insert $m'_{uv}(s_v)$ compensating logical delays between u and v, where $m'_{uv}(s_v)$ is given by (17).*

2. *each time s'_u a new event for $u \in V$* starts *for P', update:*

$$\lim_{r \searrow s'_u} n'_{st}(u, r) := n'_{st}(u, s'_u) + 1 \tag{18}$$

3. *each time t'_u a new event for u* completes *for P', update:*

$$\lim_{r \searrow t'_u} n'_{st}(u, r) := n'_{st}(u, t'_u) - 1 \tag{19}$$

If cases 2 and 3 happen simultaneously, do not change m'_{uv}.

Notation $\lim_{r \searrow s'_u} n'_{st}(u, r)$ is a right limit, *i.e.,* a limit when r tends to s'_u from above. Formula (18) indicates that we set the value of the considered counter to be $n'_{st}(u, s'_u) + 1$, immediately after $s'_u (= s_u)$ occurred. The same holds for (19).

Warning. So far we used m-delayed tasks for any integer m, positive of negative. This was for the purpose of reasoning. However it should be clear that

1. for causality reasons we cannot use negative m's (outputs cannot be used before being actually produced); and,

2. we cannot let m be unbounded, just because real-time applications require bounded buffering.

These two points are the subject of the next section.

4.4 Buffer size and causal feasibility

Consider (17). Index m_{uv} is part of the specification. Therefore, the set $\{m_{uv} \mid v \in V\}$ is bounded and contained in $\mathbf{N} \cup \{0\}$.

On the other hand, $n'_{st}(u, r)$ is nonnegative because generally a smaller number of occurrences for task u complete in a given time interval. Hence we already know that the index $m'_{uv}(r)$ is bounded from above when $r \in \mathbf{R}$ and the underlying behavior ranges over the set of all behaviors of P'.

Now, $n'_{st}(u, r)$ is not bounded in general. Therefore, it may be that $m'_{uv}(r)$ becomes negative, a nonsense since $m'_{uv}(r)$ should be a buffer index. To control this, we consider the following assumption, where we have made the underlying behavior σ' of P' explicit:

ASSUMPTION 2. *For each task u, a finite upper bound*

$$M(u) \ \geq \ \max_{\sigma' \in P', r \in \mathbf{R}} \left(n'_s(\sigma', u, r) - n'_t(\sigma', u, r) \right)$$

is known.

Note that Assumption 2 only involves the sparsity and duration of events, not the data semantics.

Under Assumption 2, we can avoid the risk of $m'_{uv}(r)$ becoming negative by having, in the synchronous "zero time" specification, a budget of $M(u) + 1$ registers on the output channel of each task u. Having this budget of registers will ensure that Protocol 1 will never lead to a negative buffer index. Of course, if, for functional reasons, the synchronous specification requires M' logical delays for communication between the two tasks u and v, then the number of registers required is $\max(M', M(u) + 1)$.

We formalize this by stating a condition [1] on the synchronous specification A that will ensure the soundness of Protocol 1:

THEOREM 1.

1. *It is always the case that $\forall r \in \mathbf{R}, m'_{uv}(r) \leq m_{uv}$, whence $m'_{uv}(r)$ is bounded from above when $r \in \mathbf{R}$, $u, v \in V$, and the underlying behavior ranges over the set of all behaviors of P'.*

2. *If P' satisfies Assumption 2, and the Synchronous Specification $P = (\mathcal{T}, \Sigma)$ is designed in such a way that the following condition holds:*

$$\forall v, u \in V : v \text{ uses } u \quad \Rightarrow \quad m_{uv} > M(u), \tag{20}$$

then $m'_{uv}(r) \geq 0$ always holds in Protocol 1, $\forall r \in \mathbf{R}, u, v \in V$.

Condition (20) expresses that $M(u) + 1$ registers must be budgeted on the output channel of each task u in *the synchronous specification P.* Let us now refine Assumption 2:

ASSUMPTION 3. *$M(u) = 1$ holds for each task u.*

Assumption 3 holds, for example, if tasks are *serialized* in P', *i.e.,* $t'^{-} \leq s'$ holds for each event $e' = (v, ., ((s', t'); .), .)$ of P', where t'^{-} denotes the termination date of the event for v preceding e' in P'. Under this assumption, inserting a double register at the output channel of each task is sufficient.

Optimized buffers

Assumption 2 is in force. Taking a closer look at formula (17) reveals that buffer size can be optimized by taking advantage of the range of the function $m'_{uv}(r)$ for $r = s_v$ a generic start date for task v. To this end, write $m'_{uv}(\sigma, r)$ to make the underlying behavior explicit. Then, consider the set:

$$[m'_u] =_{\text{def}} \left\{ m'_{uv}(\sigma', r) \ \middle| \ v \in V, \sigma' \in P', r \in S_v(\sigma') \right\} \tag{21}$$

where $S_v(\sigma')$ denotes the set of all start times of task v for behavior σ'. We have

$$[m'_u] \subseteq [0, \ldots, M(u)]$$

[1] We distinguish between *assumptions* that may or may not hold, depending on the environment, and design *conditions* on the considered system, which can be enforced by the designer.

and the inclusion can be strict, for some tag systems P. If a segment $[m,n] \subseteq [0,\ldots,M(u)]$ is a "hole" of $[m'_u]$, *i.e.,* lies outside $[m'_u]$, then when $m'_{uv}(r)$ reaches m from below, we statically know that it will exceed n at the next start time for any task v using u for its completion. The symmetric situation holds when $m'_{uv}(r)$ reaches n from above. Therefore, the slots of buffer $[0,\ldots,M(u)]$ not belonging to $[m'_u]$ are useless for the output buffer of task u, and can thus be discarded.

5. THE REALISTIC TASK PROBLEM

We now consider realistic tasks. Each occurrence of a task is characterized by *three* (not two) dates, namely, the

activation, start, and termination

of the task. Task activation occurs at will, whereas task starting can only occur when certain conditions are satisfied, *e.g.,* the processor is ready to handle the task. Therefore our tag domain of physical time is now defined by

$$\mathbf{R}^3_> \quad =_{\rm def} \quad \{(a,s,t) \in \mathbf{R}^3 \mid a \leq s \leq t\} \tag{22}$$

and is again equipped with the product order. On the other hand, the tag domain Δ of data dependencies (which entirely characterizes the data semantics) is unchanged. Consequently, Definition 3 of data equivalence is not modified.

5.1 Semantics preserving

Formula (11) defining Problem 1 must be replaced by the following one:

$$\begin{array}{rcl} e & = & (v,n,((a,s,t);(b,m_v);.),.) \\ & & \text{is replaced by} \\ e' & = & (v,n,((a,s',t');(b,m'_v);.),.), \text{ where} \\ & & s' \neq s, t' \neq t, m'_v \neq m_v, \end{array} \tag{23}$$

whereas a is unchanged. Finally, Problem 1 is replaced by the following

PROBLEM 3. *Regard the modifications $s \mapsto s'$ and $t \mapsto t'$ as given. Can we adjust the compensating logical delay m'_v in e' in order to recover data equivalence and thus compensate for the change in duration of events? Formally,*

$$\begin{array}{c} \textit{for every } (\sigma,\sigma') \textit{ such that } \sigma' \sim_{\rm data} \sigma \\ \textit{find } m' \textit{ such that } \sigma' \oplus e[s/s',t/t',m/m'] \sim_{\rm data} \sigma \oplus e \end{array} \tag{24}$$

Problem 2 is then replaced by the following

PROBLEM 4. *For event*

$$e = (v,.,((a,s,t);(b,m_v);.),.)$$

as in (24), find, for each u such that v uses u, a compensating *logical delay $m'_{uv}(s,s')$ for the modified event e' satisfying the balance equation:*

$$n'_t(u,s') - m'_{uv}(s,s') \quad = \quad n_t(u,s) - m_{uv} \tag{25}$$

The key difference with Problem 2 is that, now, $s' \neq s$, compare (25) with (13). We focus on solving Problem 4 in the sequel.

5.2 The protocol

The only difference with (13) is the presence of s' on the left hand side of (25). This is due to the fact that start times may now differ, from the original to the modified systems ($s' \geq s$). To handle this we rewrite (25) as follows:

$$\begin{array}{rcl} m'_{uv}(s,s') & = & m_{uv} + (n_t(u,s') - n_t(u,s)) \\ & & \quad - (n_t(u,s') - n'_t(u,s')) \\ & =_{\rm def} & m_{uv} + \partial\bar{n}_t(u,s,s') \\ & & \quad - \partial n'_t(u,s') \end{array} \tag{26}$$

Having the expression (26) for $m'_{uv}(s,s')$, it remains to interpolate the two additional terms on the right hand side of (26) as follows:

1. We maintain a counter $\partial n_t(u,r)$, for $r \in \mathbf{R}$; this counter is reset to zero at each start time s for task v in P, and then incremented at each subsequent termination time for task u in P, until corresponding start time s' for task v in P' is reached; we then set $\partial\bar{n}_t(u,s,s') = \partial n_t(u,s')$.

2. Counter $\partial n'_t(u,r)$ is incremented at each termination time for task u in P, and decremented at each termination time for task u in P'.

Accordingly, Protocol 1 is modified as follows:

PROTOCOL 2. *For each pair (v,u) of tasks, m'_{uv} is updated on-line according to the following rules ordered by decreasing priority:*

1. *each time $a_v = s_v (= a'_v)$ a new occurrence for task v gets activated in P, reset $\partial n_t(u,.)$ to zero;*

2. *each time s'_v a new occurrence for task v is started in P', in completing this occurrence of v, insert $m'_{uv}(s_v,s'_v)$ compensating logical delays between u and v, by using formula (26);*

3. *each time $t_u (= a'_u)$ a new event for $u \in V$* completes *in P, update:*

$$\lim_{r \searrow t_u} \partial n_t(u,r) \quad := \quad \partial n_t(u,r) + 1 \tag{27}$$

$$\lim_{r \searrow t_u} \partial n'_t(u,r) \quad := \quad \partial n'_t(u,r) + 1 \tag{28}$$

4. *each time t'_u a new event for u* completes *in P', update:*

$$\lim_{r \searrow t'_u} \partial n'_t(u,r) \quad := \quad \partial n'_t(u,r) - 1 \tag{29}$$

For the sake of clarity the description of the protocol refers to P, but it is important to notice that in the final implementation there is no need to simulate or execute P because $a_v = s_v (= a'_v)$ holds in rule 1 and $t_u (= a'_u)$ holds in rule 3.

5.3 Buffer size and causal feasibility

Consider (26). Index $m_{uv}(r)$ is part of the specification. Therefore, the set $\{m_{uv}(r) \mid v \in V, r \in \mathbf{R}\}$ is bounded and contained in $\mathbf{N} \cup \{0\}$. Hence, by formula (26), the bounds for the buffer size are obtained by inspecting the range of variation of

$$\delta n_t(u,s,s') \quad =_{\rm def} \quad n_t(u,s) - n'_t(u,s') \tag{30}$$

In (30), u ranges over the set of all tasks, (s,s') are the start dates of (e,e'), where (e,e') ranges over the set of pairs of corresponding events for task v in P and P', respectively.

The situation is more involved than for the simple task case. We know the following:

$$\begin{array}{rcl} \forall r \in \mathbf{R} & : & n_t(u,r) \geq n'_t(u,r) \\ \text{on the other hand} & : & s \leq s' \end{array}$$

Hence $\delta n_t(u,s,s')$ can be positive or negative (unlike for the simple task case, where the shift in buffer index was always non positive). For the following assumption, we write $n_t(\sigma;u,s,s')$ to make the underlying behavior explicit.

ASSUMPTION 4. *Finite lower and upper bounds*

$$\begin{array}{rcl} M^-(u,v) & \leq & \min\limits_{\sigma \in P,(s,s')} \delta n_t(\sigma;u,s,s') \\ & \leq & \max\limits_{\sigma \in P,(s,s')} \delta n_t(\sigma;u,s,s') \ \leq \ M^+(u,v) \end{array}$$

are known, where the pair (s,s') ranges as in (30).

We can now state the following counterpart of Theorem 1:

THEOREM 2.

1. *If P' satisfies the lower bound in Assumption 4, then $m'_{uv}(s,s')$ is bounded from above.*
2. *If P' satisfies the upper bound in Assumption 4, and the Synchronous Specification $P = (\mathcal{T}, \Sigma)$ is designed in such a way that the following condition holds:*

$$\forall v,u \in V : v \text{ uses } u \quad \Rightarrow \quad m_{uv} > M^+(u,v), \quad (31)$$

then $m'_{uv}(s,s') \geq 0$ always holds in Protocol 1.

Again, condition (31) indicates that $M^+(u,v)+1$ registers must be budgeted on the output channel of each task u in *the synchronous specification* P.

The case of serializable tasks.

ASSUMPTION 5 (SERIALIZABILITY). *Tasks are* serializable, i.e., *tasks can complete before getting re-activated.*

Under Assumption 5, with our setting, for each task u of modified system P', we have

$$t'^{-}_u < a'_u = t_u.$$

Therefore, for every $r \in \mathbf{R}$,

$$n_t(u,r) - n'_t(u,r) \quad \leq \quad 1 \qquad (32)$$

Regarding start dates, we know the following: for each task v, we have

$$a^-_v = s^-_v \leq s'^{-}_v \leq t'^{-}_v \leq a'_v = a_v = s_v, \qquad (33)$$

where the third inequality follows from the serializability assumption. The key remaining step is to bound the quantity:

$$n'_t(u,s'_v) - n'_t(u,s_v)$$

where u and v are fixed, and (s_v, s'_v) ranges over the start dates for a pair (e,e') of corresponding events for v in P and P', respectively.

Let L'_{uv} be a (finite or infinite) bound of $n'_t(u,s'_v) - n'_t(u,s_v)$ over all behaviors σ and all events for v. In essence, L'_{uv} captures the maximum number of occurrences of task u in any interval $[a'_v, s'_v]$ (notice that $s_v = a_v = a'_v$).

We have the following result:

THEOREM 3. *Under serializability assumption 5, if furthermore $L'_{uv} < +\infty$ holds for each pair (u,v) such that v uses u, then Assumption 4 holds (and therefore Theorem 2 applies).*

We proceed by discussing some ways which allow us to obtain a finite bound L'_{uv}.

Known minimum and maximum inter-arrival times. Suppose for each task v there is a known *minimum inter-arrival time* $m(v)$ and a known *maximum inter-arrival time* $M(v)$. This means that $m(v) \leq a_v - a^-_v \leq M(v)$.

By (33), an upper bound for the quantity

$$n'_t(u,a_v) - n'_t(u,a^-_v)$$

is also an upper bound for $n'_t(u,s'_v) - n'_t(u,s_v)$. This is because $s_v = a_v = a'_v$, therefore, the interval $[s^-_v, s'^{-}_v]$ is a sub-interval of $[a^-_v, a_v]$.

Now, an upper bound for $n'_t(u,a_v) - n'_t(u,a^-_v)$ is clearly the ratio $M(v)/m(u)$, that is, the maximum number of occurrences of u between two successive occurrences of v. Thus,

$$\frac{M(v)}{m(u)} \text{ is a valid (finite) value for } L'_{uv}.$$

Notice that the case of *periodic tasks* is a special case of this case, where the minimum and maximum inter-arrival times are both equal to the period of a task. This is the case dealt with in [9].

Same, plus known worst-case execution times. Suppose again that minimum and maximum inter-arrival times are known. If *worst-case execution times* of tasks are also known, the above bound can be made tighter. Let $W(v)$ be the worst-case execution time of task v. Thanks to the serializability assumption, and supposing also that the set of tasks is schedulable, this means that

$$s'^{-}_v \leq a'_v - W(v)$$

(i.e., the task must start early enough in order to be able to finish before the next instance arrives). Then, the ratio $M(v)/m(u)$ can be replaced by the ratio

$$\frac{M(v) - W(v)}{m(u)}$$

which is a better (i.e., smaller) value for L'_{uv}.

6. CONCLUSION

We addressed problems occurring in distributed time-sensitive architectures where communication is by sampling. These architectures are widely used in industrial distributed control systems since they naturally offer good features for fault tolerance (communication is not blocking) but have not caught enough attention as yet from both computer and control scientists.

In particular, we investigated how the functional semantics of an application can be preserved when deployed over a time-sensitive architecture. Since communication is by sampling, the functional semantics is sensitive to communication delays and action latencies, unless proper counter-measures are considered. Along the lines of [1, 9, 10, 11] we investigated a simple mechanism of controlled buffer to compensate for this.

Tracking how functional semantics is skewed due to the use of communication by sampling requires combining dates, logical time (via registers), and pointers to encode data exchange between tasks. The hybrid characteristic lends itself to the use of tag systems [2, 3]. This approach allowed us to highlight the essence of the problem without the need to consider details of scheduling mechanisms. Consequently, our framework applies to distributed and/or sequential tasks, or a combination thereof. Further, the use tag systems allows us to take into account other views of the system such as risk analysis by fault propagation, or resource consumption, or scheduling constraints. This aspect will be considered in forthcoming publications.

7. REFERENCES

[1] Baleani, M., Ferrari, A., Mangeruca, L., and Sangiovanni-Vincentelli, A. Efficient embedded software design with synchronous models. In *5th International Conference on Embedded Software, EMSOFT05* (2005), W. Wolff, Ed., ACM press.

[2] Benveniste, A., Caillaud, B., Carloni, L., Caspi, P., and Sangiovanni-Vincentelli, A. Heterogeneous reactive systems modeling: capturing causality and the correctness of loosely time-triggered protocols. In *5th International Conference on Embedded Software, EMSOFT04* (2004), G. Buttazzo, Ed., ACM.

[3] Benveniste, A., Carloni, L., Caspi, P., and Sangiovanni-Vincentelli, A. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *3rd International Wokshop on Embedded Software, EMSOFT03* (2003), R. Alur and I. Lee, Eds., vol. 2855 of *Lecture Notes in Computer Science*.

[4] Buttazzo, G. Scalable applications for energy aware processors. In *2th International Conference on Embedded Software (EMSOFT02)* (2002), A. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491 of *LNCS*, Springer Verlag, pp. 153–165.

[5] Caspi, P., and Salem, R. Threshold and bounded-delay voting in critical control systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems* (September 2000), M. Joseph, Ed., vol. 1926 of *Lecture Notes in Computer Science*, pp. 68–81.

[6] Henzinger, T. A., Horowitz, B., and Kirsch, C. M. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE 91* (2003), 84–99.

[7] Kopetz, H. *Real-time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic, 1997.

[8] Kossentini, C., and Caspi, P. Mixed delay and threshold voters in critical real-time systems. In *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems: joint international conferences on Formal Modeling and Analysis of Timed Systems (FORMATS 2004), and Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2004)* (Grenoble, Sept. 2004).

[9] The Mathworks. *Models with Multiple Sample Rates (Real-Time Workshop)*. www.mathworks.com/access/helpdesk/help/toolbox/rtw/ug.

[10] Scaife, N., and Caspi, P. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro Conference on Real-Time Systems (ECRTS'04)* (Catania, June 2004), IEEE Computer Society.

[11] Tripakis, S., Sofronis, C., Scaife, N., and Caspi, P. Semantic-preserving and memory efficient implementation of inter-task communication on static-priority or EDF schedulers. In *5th International Conference on Embedded Software, EMSOFT05* (2005), W. Wolff, Ed., ACM press.