End-To-End Latency Characterization of Implicit and LET Communication Models

Jorge Martinez[2], Ignacio Sañudo[1], Paolo Burgio[1], Marko Bertogna[1]
[1]Universita di Modena, Italy
[2]Robert Bosch GmbH, Germany
{ignacio.sanudoolmedo, paolo.burgio, marko.bertogna}@unimore.it
JorgeLuis.MartinezGarcia@de.bosch.com

***Abstract*—The objective of this paper is to provide a solution to this year's FMTV industrial challenge presented by Robert Bosch GmbH. Last year, the challenge consisted in a timing and schedulability analysis of an engine management system to be executed on a shared–memory multi-core platform, in which the tasks were scheduled following a mixed preemptive configuration. This year, the challenge is integrated with the analysis of two different inter-task communication semantics, namely, Implicit Communication and Logical Execution Time. Whereas the former focuses on data consistency as proposed by AUTOSAR, the latter guarantees temporal determinism. This paper presents a formal definition of these semantics, their possible implementations (Challenge I) as well as their overhead (Challenge II). Taking into account these semantics, we perform an end-to-end-latency analysis of the event chains present in the given AMALTHEA model (Challenge III).**

I. Introduction

In the recent years, the amount of electronics in automotive vehicles has risen dramatically, and it will further increase in the future. The technological reason behind such a trend in the automotive industry is due to increased number of safety and control functionalities that are being integrated in modern cars, as well as to the replacement of older hydraulic and mechanical direct actuation systems with modern by-wire counterparts, leading to an increased safety and comfort at a reduced unit cost. Well-known examples are electronic engine control, ABS, electronic stability program (ESP), active suspension, etc. With the evolution of processors, several industries are facing a transition from single-core to multi-core systems. This kind of platforms allow application providers to continue exploiting Moore's law greedy demand for computing power without incurring thermal and power problems.

In the automotive domain, multi-core platforms bring major improvements for some applications requiring high performance such as high-end engine controllers, electric and hybrid powertrains, advanced driver assistance systems, etc. Moreover, the increased computational power of multi-core platforms may allow integrating into a single controller multiple functionalities that were spread around different ECUs, reducing the number of computing units as well as communication overhead. Some of the cores may be dedicated to handling low-level services (AUTOSAR's basic software) or high-level services (AUTOSAR's application software), provided the necessary timing and safety constraints are satisfied, adapting existing design methods to the new multi-core paradigm.

The basic timing requirement for most tasks in automotive systems is to finish execution before their deadline. Typically, different hard real-time tasks share the same hardware and software resources, relying on each other to implement a desired functionality. The resulting inter-task communication might lead to data consistency issues.

This paper presents a brief overview of a solution to the FMTV verification challenge [1]. The challenge consists in the analysis of a powertrain application that runs in a multi-core embedded platform. The application information is abstracted and represented through the AMALTHEA model. AMALTHEA is an open source platform for the modeling of embedded multi/many core systems. The AMALTHEA model contains the hardware description, constraints and software requirements. The model follows the same hardware and software specification of the 2016 model challenge [2], but with different inter-task communication semantics, namely, implicit communication and Logical Execution Time (LET). Specifically, the addressed challenges are:

1) propose and demonstrate how implicit and LET communication may be realized, e.g., by adding additional runnables and/or tasks performing copy operations.
2) compute the overheads in terms of extra cycles used for memory access and also in terms of extra memory required due to the proposed implementation.
3) compute end-to-end latencies (age/reaction latency) of the event chains (best, average and worst case). The solution should be able to handle multi-rate effect chains consisting of tasks with harmonic and non-harmonic periods.
4) propose a different label mapping that could possibly reduce the memory access overheads.
5) factor in the effects of contention on the interconnect in the memory access overhead and show the impacts on end-to-end latencies.

II. System model and notation

This section describes the terminology and notation used throughout the paper, following the solution provided last year

in [3] and considering the information abstracted out of the AMALTHEA model.

The smallest functional entity in the automotive management software model is called *runnable*. Runnables having the same functional period according to the control dynamics are grouped into the same task. In the simplest case, one functionality is realized by means of a single runnable. However, more complex functionalities are typically accomplished using several communicating runnables, possibly distributed over multiple tasks.

The platform is assumed to comprise four identical cores, with tasks and runnables statically partitioned to the cores, and no migration support. Each task τ_i is specified by a tuple $(C_i, D_i, T_i, P_i, PT_i)$, where C_i stands for the worst-case execution time (WCET), D_i is the relative deadline, T_i is the period, P_i is the priority, and PT_i defines the type of preemption. Every period T_i, each task releases a job composed of γ_i subsequent runnables, where τ_i^r represents the r^{th} runnable of τ_i, with $1 \leq r \leq \gamma_i$. The worst-case execution time of τ_i^r is denoted as C_i^r. Therefore,

$$C_i = \sum_{r \in [1,\gamma_i]} C_i^r. \tag{1}$$

We also denote as $\overline{C}_i^r$ the cumulative execution time of runnables $\tau_i^, \ldots, \tau_i^r$, i.e.,

$$\overline{C}_i^r = \sum_{r \in [1,r]} C_i^r. \tag{2}$$

Tasks are scheduled by the operating system based on the assigned (fixed) priorities. The scheduling policy may be either preemptive or cooperative, as specified by PT_i. Preemptive tasks may always preempt lower priority tasks, while cooperative tasks may preempt a lower priority one only at runnable boundaries. Preemptive tasks are assumed to have always a higher priority than any cooperative task.

The model assumes one instruction-per-cycle (i.e., $IPC = 1$), so that the execution time of a runnable τ_i^r, without taking memory computation into account, can be computed as $C_i^r = E_i^r / f$, where E_i^r is a bound on the number of instructions for the considered runnable, and f is the core frequency. The computational phase is also characterized by a parameter $F_\ell^{R/W}$ that represents the number of times a runnable accesses a label (a.k.a. frequency access in the model).

Regarding the type of access, a task can be either a sender or a receiver of a label. A sender is a task that writes a label. In the considered AMALTHEA model, there is only one sender per label, while there may be multiple receiving tasks reading that label.

The time it takes to access a label depends on the memory the label is mapped on to. If the label is allocated to the local memory, the considered task may access it within 1 clock cycle. Otherwise, if the label is in the LRAM of a different core or it is in the GRAM, the task pays an access penalty of 8 time units. Since multiple cores may concurrently access the same shared memory, an additional contention penalty is paid when accessing external memories due to the arbitration mechanism. The model assumes a FIFO arbitration, so that a core may wait up to m cycles to obtain access to the addressed memory, where $m = 4$ is the number of cores.

Since reading and writing times are assumed to be equivalent in the model, we denote as $\xi_\ell(x)$ the time it takes to access a shared label ℓ from memory x, where x may be $GRAM$, local LRAM ($LRAM_L$) or external LRAM ($LRAM_E$). In the considered model, we thus have $\xi_\ell(GRAM) = \xi_\ell(LRAM_E) = 8 + m$, and $\xi_\ell(LRAM_L) = 1$.

The overall worst-case execution time C_i^r of runnable τ_i^r taking into consideration the memory computation of all labels ℓ is given by,

$$C_i^r = \frac{E_i^r}{f} + \sum_\ell \left\{ F_{i,\ell}^r * \xi_\ell(x) \right\} \tag{3}$$

where $F_{i,\ell}^r$ represents the number of times the label ℓ is accessed by runnable τ_i^r.

A. Communication semantics

In line with the multi-core complexity trend, automotive applications are evolving towards more complicated task and runnable settings. As tasks communicate across the memory hierarchy, data consistency problems may arise. Different communication semantics have been proposed in order to provide a deterministic and consistent task communication. The challenge proposes an analysis of (i) implicit and (ii) LET communication semantics.

According to AUTOSAR's implicit communication [4], tasks accessing shared labels should work on task-local copies instead of the original labels. To avoid data inconsistency, each task instance performs a copy of the required labels at the beginning of its execution. After working on local copies in an exclusive way, it then publishes its results at the end of its execution. Depending on the task instance and the delay due to interfering tasks, the time it takes to update a shared label may significantly vary.

Logical Execution Time (LET) [5] [6] is a hard real-time programming abstraction that was introduced with the time-triggered programming language Giotto in order to improve the determinism of the communication times. As the relevant behavior of real-time tasks is determined by when inputs are read and outputs are written, the LET semantics requires that inputs and outputs are updated logically at the beginning and at the end of the so called *communication interval*, i.e., in correspondence to the release times of the communicating tasks. This allows to deterministically fix the time it takes from reading an input to writing an output regardless of the actual response time of the involved communicating tasks.

III. Inter task communication

In this section, we describe the implementations proposed for the Implicit and LET communication paradigms.

A. Implicit communication

Let I_i and O_i be the set of all shared labels read and written by tasks τ_i, respectively. I_i and O_i therefore represent the inputs and outputs of the considered task. Our proposal for implicit communication suggests that any task τ_i accessing a shared label works on a copy instead of the original label. Copies are created, statically allocated to the task-local scratchpad and inserted in runnables at compile time. Furthermore, two task-specific runnables τ_i^0 and $\tau_i^{(\gamma_i+1)}$ (also called τ_i^{last} for simplicity) are to be inserted at the beginning and at the end of the task. Runnable τ_i^0 is responsible of reading shared labels to the local copies, while τ_i^{last} will write the local copies to the corresponding shared variables. If $\dot{I}_i$ and $\dot{O}_i$ represent the set of τ_i-local copies of the labels contained in I_i and O_i, respectively, runnable τ_i^0 updates $\dot{I}_i$, whereas runnable τ_i^{last} publishes its updates by writing $\dot{O}_i$ to the corresponding shared variables in O_i. See Figure 1.

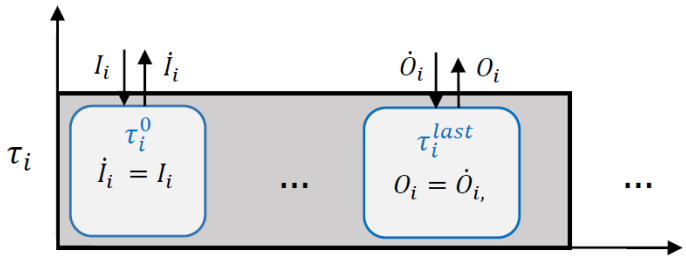

Figure 1. Implicit communication.

For example, suppose a task τ_i reads shared label L_1 and writes to shared label L_2. Let $L_{i,1}$ and $L_{i,2}$ represent the τ_i-local copies of L_1 and L_2 respectively. The implicit model dictates that $L_{i,1}$ be updated by runnable τ_i^0 at the beginning of task τ_i. After that, τ_i reads $L_{i,1}$ and writes to $L_{i,2}$, never accessing the original labels L_1 and L_2. In the end, runnable τ_i^{last} writes the latest value of $L_{i,2}$ to L_2. It does not need to publish L_1, since it did not modify it. See Figure 2.

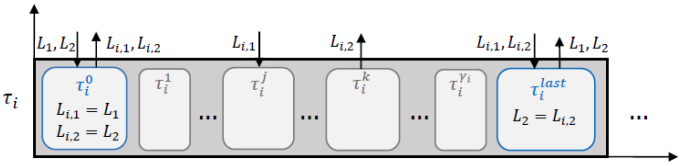

Figure 2. Implicit communication

An upper bound on the overhead introduced by the copy-in (τ_i^0) and copy-out (τ_i^{last}) runnables can be easily computed as

$$C_i^0 = \sum_{\ell \in I_i} \xi_\ell(x), \tag{4}$$

and

$$C_i^{last} = \sum_{\ell \in O_i} \xi_\ell(x), \tag{5}$$

where the sum is extended over all shared labels read (resp. written) by the considered task τ_i. Note that the model assumes that labels to be written are mapped on to the local scratchpad of the (unique) writer task, while labels that are only read but never written are mapped on to the global RAM. Therefore, a copy-in may take 1 or 9 time units, while a copy out will always take 1 time unit.

The total execution time of τ_i is computed as

$$C_i = C_i^0 + C_i^{last} + \sum_{r \in [1,\gamma_i]} C_i^r, \tag{6}$$

where the execution time of a single runnable includes only accesses to local variables, i.e., 1 time unit for each one of the $F_{i,\ell}^r$ accesses by the considered runnable:

$$C_i^r = \frac{E_i^r}{f} + F_{i,\ell}^r. \tag{7}$$

The additional memory occupancy in the implicit model is given by the local copies created for shared labels, i.e., all labels in $I_i \cup O_i$ for all tasks τ_i.

B. LET communication

Differently from the implicit case, the LET paradigm enforces task communications at deterministic times, corresponding to task activation times. In our implementation, each reader creates one or more local copies of the shared label. Since the considered model allows just one writer task for each label, the writer task is allowed to directly modify the original label, updating the readers copies at well-determined times.

We hereafter consider the communication between the writer and one of the readers. Assume the writer has period $T_W = 2$ and reader $T_R = 5$, as in Figure 3 left: while τ_W may repeatedly write the considered label L, these updates are not visible to the concurrently executing reader, until a *publishing point* $P_{W,R}^n$, where the value is updated for the next reader instance. This point corresponds to the first upcoming writer release that directly precedes a reader release, i.e., where no other write release appears before the arrival of the following reader instance. We call *publishing instance* the writing instance that updates the shared value for the next reading instance, i.e., the writer's job that directly precedes a publishing point. Note that not all writing instance are publishing instances. See Figure 3, where publishing instances are marked in bold red.

It is also convenient to define *reading points* $Q_{R,W}^n$, which correspond to the arrival of the reading instance that will first use the new data published in the preceding publishing point $P_{R,W}^n$. Figure 3 shows publishing and reading points for a case where the writer task has a higher (a) or smaller (b) period than the reader task.

Let $T_{\max} = \max(T_W, T_R)$. Publishing and reading points for two communicating tasks can be easily computed as a function of the task periods, as shown in the next theorem.

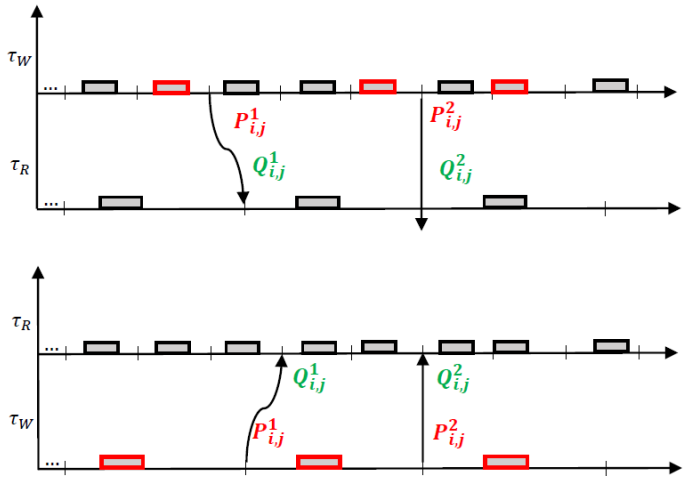

Figure 3. Publishing and reading points when the reader has a higher (a) or smaller (b) period than the writer.

Theorem 1. *Given two communicating tasks, the publishing points of the writing task τ_W can be computed as*

$$P^n_{W,R} = \left\lfloor \frac{nT_{\max}}{T_W} \right\rfloor T_W, \quad \forall n \in [1, n_{W,R}], \tag{8}$$

while the reading points of the reading task τ_R can be computed as

$$Q^n_{W,R} = \left\lceil \frac{nT_{\max}}{T_R} \right\rceil T_R, \quad \forall n \in [1, n_{W,R}], \tag{9}$$

where $n_{W,R}$ is the number of jobs released in a hyperperiod by the task with the longest period, i.e.,

$$n_{W,R} = \frac{LCM(T_W, T_R)}{T_{\max}} = n_{R,W}. \tag{10}$$

Proof. If the writer τ_W has a smaller period than the reader τ_R, i.e., $T_W \leq T_R$ (as in Figure 3 left), there is one publishing and one reading point for each *reading* instance. There are $LCM(\tau_W, \tau_R)/\tau_R = n_{w,r}$ such instances. Reading points trivially correspond to each reading task release, i.e.,

$$Q^n_{W,R} = nT_R, \quad \forall n \in [1, n_{W,R}],$$

while publishing points correspond to the last writer release before such a reading instance, i.e.,

$$P^n_{W,R} = \left\lfloor \frac{nT_R}{T_W} \right\rfloor T_W, \quad \forall n \in [1, n_{W,R}].$$

Otherwise, when the writer τ_W has a larger period than the reader τ_R, i.e., $T_W \geq T_R$ (as in Figure 3 left), there is one publishing and one reading point for each *writing* instance. Again, there are $LCM(\tau_W, \tau_R)/\tau_W = n_{w,r}$ such instances. Publishing points trivially correspond to each writing task release, i.e.,

$$P^n_{W,R} = nT_W, \quad \forall n \in [1, n_{W,R}],$$

while reading points correspond to the last reader release before such a writing instance, i.e.,

$$Q^n_{W,R} = \left\lceil \frac{nT_{\max}}{T_R} \right\rceil T_R, \quad \forall n \in [1, n_{W,R}].$$

It is easy to see that, in both cases $T_W \leq T_R$ and $T_W \geq T_R$, the formula for $P^n_{W,R}$ and $Q^n_{W,R}$ are generalized by Equations (8) and (9). Note that when $T_W = T_R$, $P^n_{W,R} = Q^n_{W,R} = nT_W$. □

Let $I_{W,R}$ denote the set of labels written by τ_W and read by τ_R. For each of these labels, the reading task τ_R creates a local copy to which it has exclusive access. Let $\dot{I}_{W,R}$ denote the set of τ_R-local copies of the labels contained in $I_{W,R}$. A communication-specific runnable is to be inserted to update $\dot{I}_{i,j}$ at the end of the communication period, i.e., by the latest completing task before a publishing point.

We hereafter treat separately the cases of (i) harmonic synchronous communication and (ii) non-harmonic synchronous communication. The asynchronous case will be considered as future work since the given AMALTHEA model does not provide enough information to properly model interrupts and adaptive variable-rate (AVR) tasks [7].

1) Harmonic Synchronous Communication (HSC): Two communicating tasks τ_W and τ_R have harmonic periods if the period of one of them is an integer multiple of the other. When a harmonic synchronous communication (HSC) is established, the following relations hold: $LCM(T_W, T_R) = T_{\max}$, $n_{W,R} = n_{R,W} = 1$ and $P^n_{W,R} = Q^n_{W,R} = nT_{\max}$, i.e., publishing and reading points are integer multiples of the largest period of the communicating tasks.

Consider the example in Figure 4, where two tasks τ_l and τ_s, with $T_l/T_s = 2$, both read shared labels L_1 and L_2. Moreover, τ_l writes to L_1, while τ_s writes to L_2. The proposal suggests that τ_s and τ_l are to read $L_{s,1}$ and $L_{l,2}$ instead of the original labels. These copies are to be updated by either runnable τ_s^{last} or runnable τ_l^{last} depending on whichever job finishes last before the next publishing point. In other words, the responsibility to update the copies is given either to the reader or to the writer, depending on which one completes last in the communication interval. The first reader instance after the publishing point is the first one that accesses the updated value. Such a value will be used by all reading instances until the next reading point.

Unlike the implicit communication, only one task pays the overhead for maintaining the determinism in the communication. Assuming such a task is τ_i, its worst-case execution time can be computed as

$$C_i^{total} = \sum_{r \in [1, \gamma_i]} C_i^r + \sum_{\ell \in I_i \cup O_i} \xi_\ell(x), \tag{11}$$

where τ_i is assumed to update all its shared labels. Better estimations are possible considering which task effectively finishes last in each communication period, making the analysis significantly more complex.

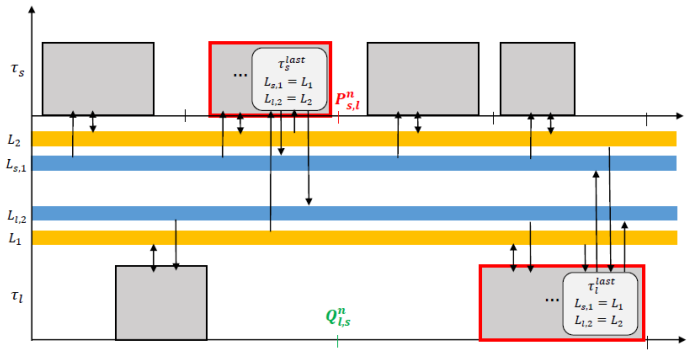

Figure 4. LET harmonic communication

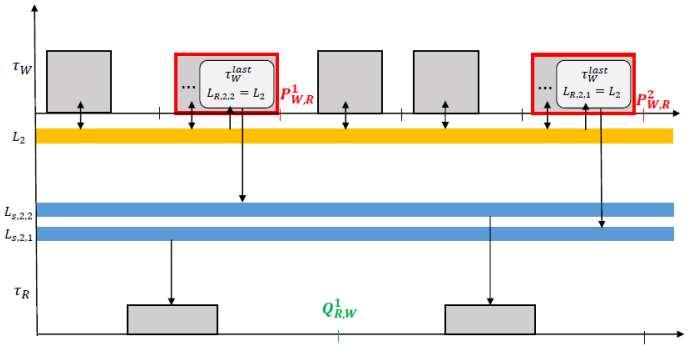

Figure 5. Non harmonic T_W communicates with T_R

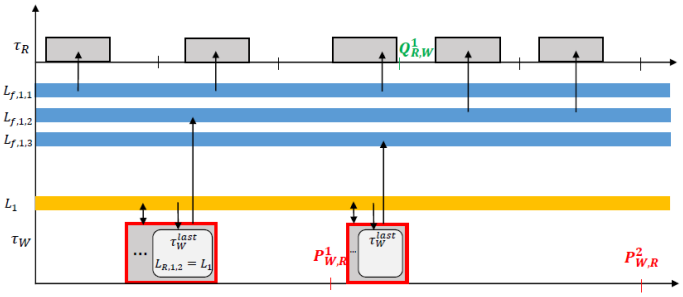

Figure 6. Non harmonic T_W communicates with T_R

The additional memory occupancy is given by the local copies created for shared labels, i.e., all labels in I_i for all tasks τ_i.

2) Non-Harmonic Synchronous Communication (NHSC): When two communicating tasks do not have harmonic periods, a non-harmonic synchronous communication (NHSC) is established. The general formulas of Section III-B apply.

Like in the HSC case, the reading task of a shared label accesses a local copy instead of the original label. However, due to the misaligned activations of the communicating tasks, at least two copies of the same shared label are needed in a NHSC. A task-specific runnable is to be inserted at the end of the writer in order to update the copies of $I_{W,R}$ before the publishing point. If only one copy was used, a task could be writing it while the reader is reading it, leading to an inconsistent state. With two copies, instead, a reader may read a local copy, while the writer may freely write a new value for the next reading instance in a different buffer.

For example, consider a reading task τ_R and a writing task τ_W communicating through a shared variable L, with $2T_R = 5T_W$ as in Figure 5. There are two τ_R-local copies, $L_{R,1}$ and $L_{R,2}$, of the shared label L. The reading task τ_R reads from one of these copies instead of the original label. These copies are to be updated by the last runnable τ_W^{last} of the writing task. Note that τ_W directly writes to L instead of a local copy.

There might also be cases where three copies per labels are needed in order to fulfill the LET determinism. Consider Figure 6 where $5T_R = 2T_W$. Then, τ_R reads either from $L_{R,1}$, $L_{R,2}$ or $L_{R,3}$ instead of the original label L. These three copies are to be updated by runnable τ_W^{last}. Note that τ_W directly accesses L.

An extra copy of L is needed due to the fact that the value computed by the second writing instance may be available before the next reading point, depending on the execution time of τ_W. If such publishing instance updates $L_{R,1}$, instances before the next reading point may read inconsistent values. If it updates $L_{R,2}$, the update by the previous publishing instance would be overwritten without being used, and the next reading instance would either read an inconsistent value, or a value that does not correspond to a release time, violating the LET paradigm. A third buffer is therefore needed to store the new value avoiding conflicts.

In general, this happens when a publishing instance has a best-case finishing time that may precede the next reading point. Let us define $w^n_{W,R}$ as the window of time between a publishing point $P^n_{W,R}$ and the next reading point $Q^n_{W,R}$. Then, using Equations (8) and (9),

$$w^n_{W,R} = \left\lceil \frac{nT_{\max}}{T_R} \right\rceil T_R - \left\lfloor \frac{nT_{\max}}{T_W} \right\rfloor T_W. \qquad (12)$$

If the best-case response time of a publishing instance may be smaller than the corresponding $w^n_{W,R}$, a third buffer is needed to store the new value. Depending on the above condition, the additional memory occupancy due to the local copies is two or three times the size of the labels in I_i for all tasks τ_i.

IV. Effect Chain

In this section, we present an analysis of the third challenge, computing end-to-end latencies of effect chains taking into account age and reaction semantics.

An effect chain is a producer/consumer relationship between runnables working on shared labels. Effects chains are assumed to be triggered by an event or a task release. The first task in the chain produces an output (i.e., writes to a shared label) for another task following in the event chain. This second task reads the shared label to write an output to

a different shared label, which may be then read by a third task, and so on. When the last task produces its final output the event chain is over.

In [8], four different communication semantics are described to characterize the timing delays of effect chains. In particular, the *age latency* is defined as "the delay between the last input that is not overwritten until the last output with this input". Age is also referred to as last-to-last (L2L) delay. In other words, the age latency denotes the largest delay between the start of the event chain and its end, i.e., from the first input in the chain until the last output related to this input. It measures for how long may an input continue influencing the final output of the event chain. This metric is particularly important for control applications, such as, fuel injection control or cruise control.

The *reaction latency*, also referred to as first-to-first (F2F) in [8], is the delay between the first input that produces an output until the arrival of a new output computed with a new stimulus. In other words, the reaction latency denotes the largest delay between the start of the event chain and the first output produced with a different input, i.e., from the first input in the chain until the first output not related to this input. It measures how much time it may take for a new event to propagate to the end of the event chain. This metric allows estimating the reactivity to new inputs, e.g., measuring the delay of a button-to-action event. Figure 7 depicts both semantics.

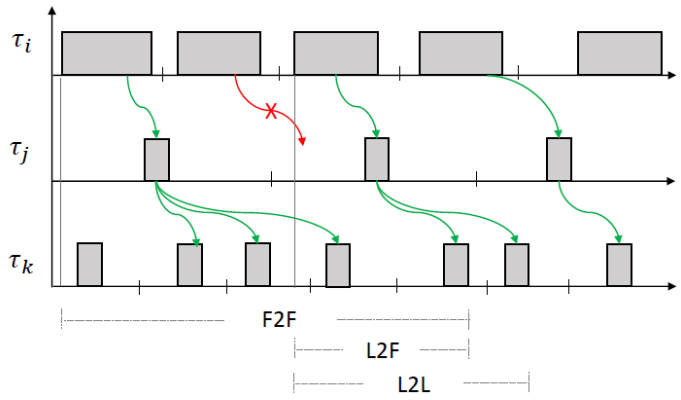

Figure 7. Age and reaction semantics

Before computing end-to-end age and reaction latencies of an effect chain, we first compute the maximum delay ϕ_i^r between two instances of the same runnable τ_i^r belonging to two consecutive jobs, i.e., the maximum delay between two consecutive instances of a runnable accessing (reading or writing) a particular label. In Figure 8, ϕ_i^r is derived as a function of the best-case start time s_i^r and the worst-case response time R_i^r of runnable τ_i^r.

$$\phi_i^r = T_i - s_i^r + R_i^r - (\epsilon_1 + \epsilon_2). \tag{13}$$

Assuming the first and the second runnable instance access the shared label at the beginning and at the end of their execution, respectively, it follows $\epsilon_1 = \epsilon_2 = 0$, and

$$\phi_i^r = T_i - s_i^r + R_i^r. \tag{14}$$

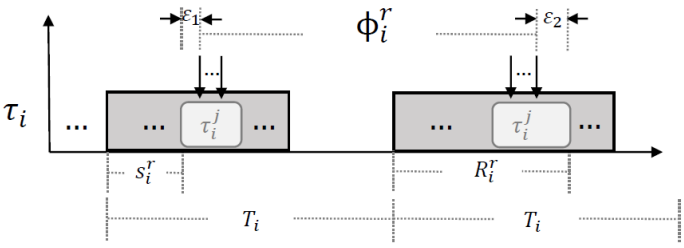

Figure 8. Worst-case delay ϕ_i^r between two consecutive instances of runnable τ_i^r.

In the following, we compute age and reaction latencies for the considered communication semantics.

A. Explicit Communication

Consider an effect sub-chain where a runnable τ_W^i writes to a label L which is in turn read by another runnable τ_R^j. We hereafter compute the following latencies: worst-case sub-chain propagation delay $\delta_{W,R}^{i,j}$, worst-case age latency $\alpha_{W,R}^{i,j}$ and worst-case reaction latency $\rho_{W,R}^{i,j}$. To do this, we consider different worst-case settings where all the following conditions hold:

C1. τ_W^i stores L right after τ_R^j started loading it.
C2. A first read occurs at s_R^j, while the next one is after ϕ_R^j time-units.
C3. A first write occurs at s_W^i, while the next one is after ϕ_W^i time-units.
C4. All following reading accesses are separated by T_j time-units.

If $\phi_R^j < \phi_W^i$, the worst-case sub-chain propagation delay $\delta_{W,R}^{i,j}$ corresponds to ϕ_R^j, as shown in Figure 9. If instead $\phi_R^j \geq \phi_W^i$, $\delta_{W,R}^{i,j}$ corresponds to ϕ_W^i, as shown in Figure 10. Therefore,

$$\delta_{W,R}^{i,j} = \min\{\phi_R^j, \phi_W^i\}. \tag{15}$$

To compute the age latency $\alpha_{W,R}^{i,j}$, we again separately consider the cases with $\phi_R^j < \phi_W^i$ and $\phi_R^j \geq \phi_W^i$. The corresponding settings leading to the worst-case age latency are shown in Figure 11 and 12, respectively. Therefore,

$$\alpha_{W,R}^{i,j} = \begin{cases} \phi_R^j + \left\lfloor \frac{\phi_W^i - \phi_R^j}{T_R} \right\rfloor T_R, & \phi_R^j < \phi_W^i \\ \phi_W^i, & \phi_R^j \geq \phi_W^i, \end{cases} \tag{16}$$

or, equivalently,

$$\alpha_{W,R}^{i,j} = \min\left\{\phi_R^j, \phi_W^i\right\} + \left\lfloor \frac{\max\{\phi_W^i - \phi_R^j, 0\}}{T_R} \right\rfloor T_R \tag{17}$$

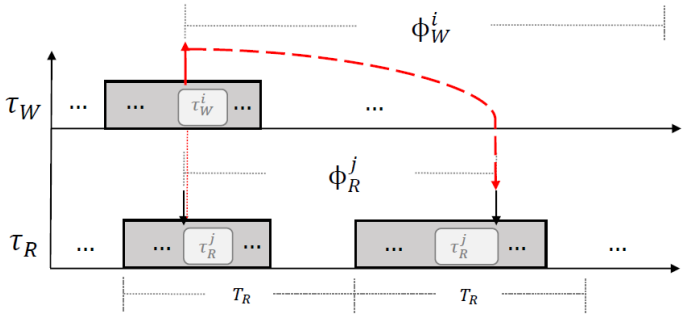

Figure 9. Worst-case sub-chain propagation delay when $\phi_R^j < \phi_W^i$

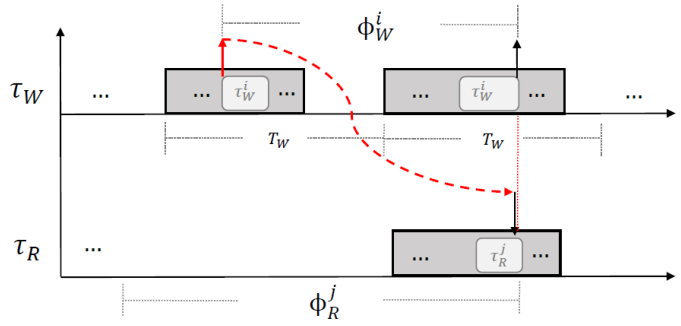

Figure 10. Worst-case sub-chain propagation delay when $\phi_R^j \geq \phi_W^i$

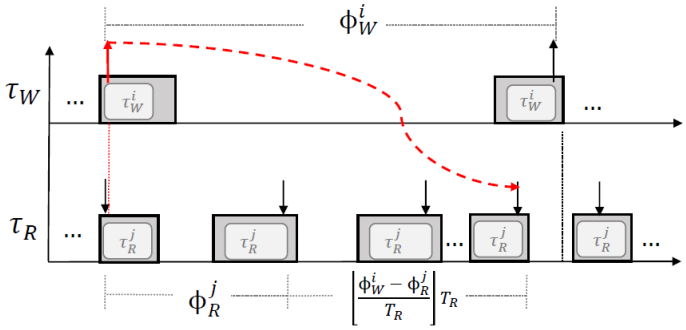

Figure 11. Age latency $\alpha_{W,R}^{i,j}$ when $\phi_R^j < \phi_W^i$.

To compute the worst-case reaction latency $\rho_{W,R}^{i,j}$, more subcases need to be considered. The worst-case situation with $\phi_R^j < \phi_W^i$ is depicted in Figure 13, where we compute the reaction latencies of both the first and the second writing events. The reaction latency of the first writing event is $T_W + \phi_R^j$, while that for the second one is $\phi_R^j + \left\lfloor \frac{\phi_W^i - \phi_R^j}{T_R} \right\rfloor T_R + T_R$.

The worst-case situation with $\phi_R^j \geq \phi_W^i$ is depicted in Figure 14, where we compute the reaction latencies of two different writing events, highlighted in bold red in the figure. The reaction latency of the first writing event is

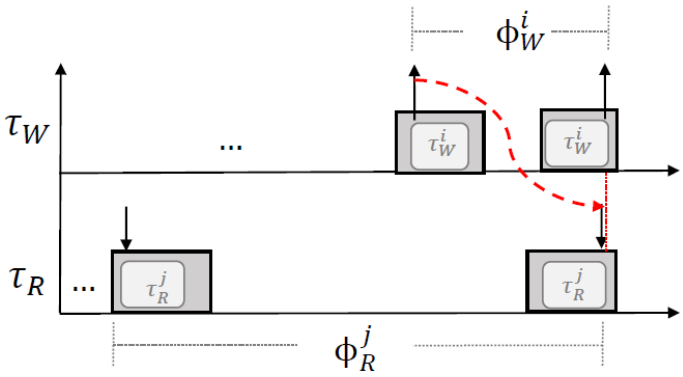

Figure 12. Age latency $\alpha_{W,R}^{i,j}$ when $\phi_R^j \geq \phi_W^i$.

$T_W + \left\lfloor \frac{\phi_R^j - \phi_W^i}{T_W} \right\rfloor T_W + \phi_W^i$, while that for the second one is $\phi_W^i + T_R$.

Merging both cases,

$$\rho_{W,R}^{i,j} = \begin{cases} \max\left\{T_W + \phi_R^j, \phi_R^j + \left(\left\lfloor \frac{\phi_W^i - \phi_R^j}{T_R} \right\rfloor + 1\right) T_R\right\}, & \phi_R^j < \phi_W^i \\ \max\left\{T_R + \phi_W^i, \phi_W^i + \left(\left\lfloor \frac{\phi_R^j - \phi_W^i}{T_W} \right\rfloor + 1\right) T_W\right\}, & \phi_R^j \geq \phi_W^i \end{cases} \tag{18}$$

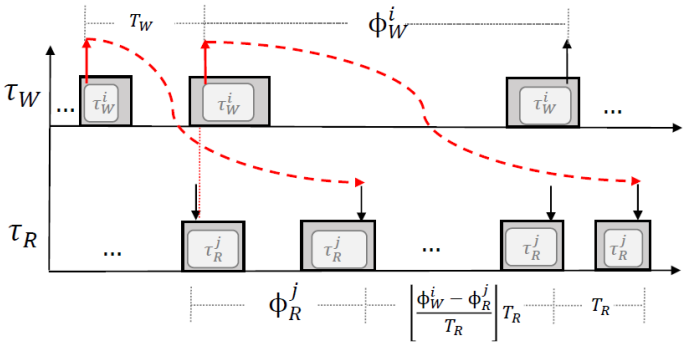

Figure 13. Reaction latency $\rho_{W,R}^{i,j}$ when $\phi_R^j < \phi_W^i$.

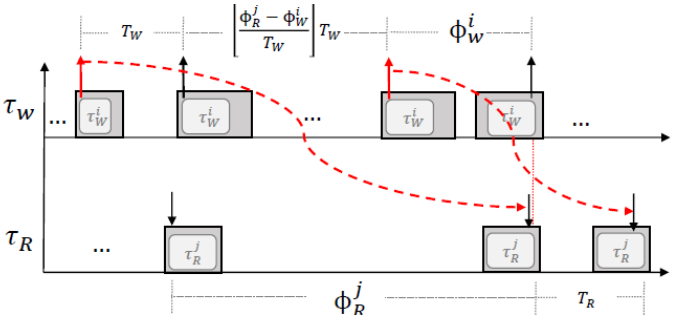

Figure 14. Reaction latency $\rho_{W,R}^{i,j}$ when $\phi_R^j \geq \phi_W^i$.

B. Implicit

As explained in Section III-A, AUTOSAR's implicit communication introduces two extras runnables at task boundaries in charge of reading and publishing the shared labels. From an end-to-end latency perspective, the implicit communication can be considered as a particular case of the explicit semantic, considering τ_W^{last} and τ_R^o as writing and reading runnables, respectively.

For instance, the worst-case sub-chain propagation delay $\delta_{W,R}^{i,j}$ for any pair of communicating runnables τ_W^i and τ_R^j is equal to $\delta_{W,R}^{last,o}$, plus an extra delay Δ_R due to the fact that τ_R publishes all its shared labels at the end of its execution. Figure 15 shows the case with $\phi_R^0 < \phi_W^{last}$. It is easy to see that $\Delta_R = R_R - R_R^0$, where R_R and R_R^0 represent the worst-case response time of τ_R and τ_R^0, respectively. A similar situation has been verified to happen in the scenarios considered for all other worst-case latency settings. In order to get a new set of relations for this type of communication, it is then sufficient to add Δ_R to Equation 15, 17 and 18.

$$\delta_{W,R}^{i,j} = \delta_{W,R}^{last,o} + \Delta_R = \min\{\phi_R^o, \phi_W^{last}\} + \Delta_R. \tag{19}$$

$$\alpha_{W,R}^{i,j} = \alpha_{W,R}^{last,o} + \Delta_R \tag{20}$$

$$\rho_{W,R}^{i,j} = \rho_{W,R}^{last,o} + \Delta_R \tag{21}$$

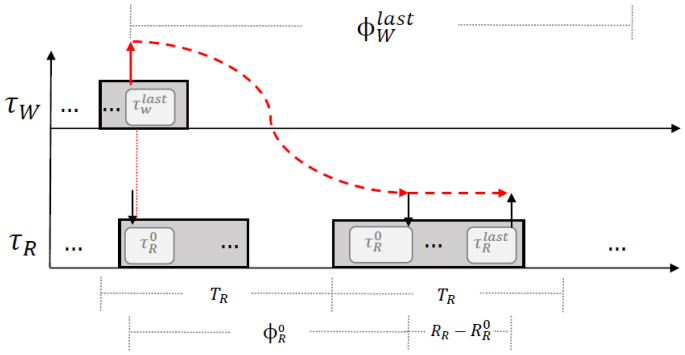

Figure 15. E2E implicit semantic

C. LET

Given an effect chain (EC) involving multiple tasks, there is a fixed number of possible communication chains in a hyperperiod, starting from the end of the period of the first task and finishing with the release of the last one in the EC. We call these chains *basic paths*. The number of *basic paths* can be computed as

$$\zeta = \frac{LCM_{i=1}^{\eta}(T_i)}{T_{\max}}$$

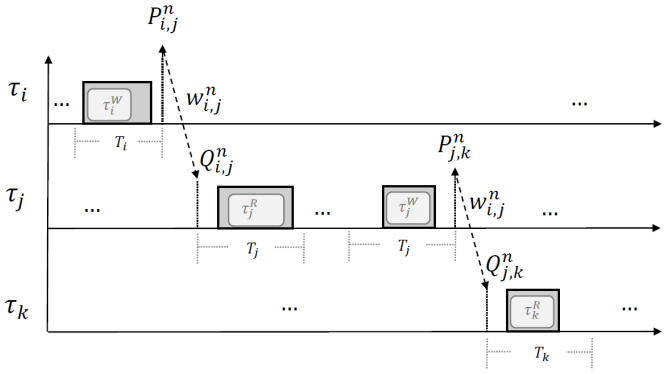

Figure 16. E2E LET semantic

Where η represents the number of tasks composing an effect-chain. Note that if all tasks in the EC are harmonic, then $\zeta = 1$.

Given a sub-chain composed of two communicating tasks τ_W and τ_R, let $n_{W,R}^{EC}$ denote the number of jobs released in the hyperperiod of the EC by the task with the longest period in the sub-chain, i.e., $n_{W,R}^{EC} = \frac{LCM_{i=1}^{\eta}(T_i)}{max(T_W, T_R)}$.

Consider an EC composed of three tasks τ_i, τ_j and τ_k, as depicted in Figure 16. The length θ_{EC}^n of the n-th basic path of the EC can be computed as $\theta_{EC}^n = w_{i,j}^n + P_{j,k}^n - Q_{i,j}^n + w_{j,k}^n$ $\forall n \in [1, \zeta]$. Once the length of the ζ basic paths is known, see Algorithm 1, the last-to-first propagation delay $\delta(\rho)$, can be computed as

$$\delta(EC) = T_i + \max_{n\in[1,\zeta]}\{\theta_{EC}^n\} + T_k \tag{22}$$

The end-to-end age latency can be computed as

$$\alpha(EC) = \max_{n\in[1,\zeta]}\left\{\theta_{EC}^n + Q_{j,k}^{n+1} - Q_{j,k}^n\right\} + T_i, \tag{23}$$

where $Q_{j,k}^n = P_{j,k}^n + w_{j,k}^n$. The reaction latency as

$$\rho(EC_{F2F}) = \max_{n\in[1,\zeta]}\left\{\theta_{EC}^n + P_{i,j}^n - P_{i,j}^{n-1}\right\} + T_k, \tag{24}$$

where $P_{i,j}^n = Q_{i,j}^n - w_{i,j}^n$.

D. Challenge effect chains

In this section, the worst-case age and reaction delays are computed for the task set given for the FMTV challenge. For space reasons, only the delays for the LET semantic are shown. We considered the effect chains in Figure 17.

Effect chain 1. As shown in Figure 17, all runnables involved in the considered effect chain belong to the same task. However, the last runnable in the chain is always executed first. This EC may be treated as one composed of two tasks with the same period: τ_{10ms_1} and τ_{10ms_2}, where the latter contains R_{10ms_107} and the former R_{10ms_149}, R_{10ms_243} and R_{10ms_272}. Then,

$$\begin{aligned}\alpha(EC) = \rho(EC) &= T_{10ms\_1} + T_{10ms\_2} \\ &= 10ms + 10ms = 20ms\end{aligned}$$

Algorithm 1 θ^n_{EC} Parameters' Calculation

if $\zeta == n^{EC}_{i,j}$ **then** ▷ τ_i and τ_j define the *basic path*
  while $0 < n \leq \zeta$ **do**
    while $0 < m \leq n^{EC}_{j,k}$ **do**
      $d \leftarrow P^m_{j,k} - Q^{n-1}_{i,j}$
      if $d > 0$ **then** $break$
      $m \leftarrow m + 1$
    $points \leftarrow (w^{n-1}_{i,j}; n-1; Q^{n-1}_{i,j}; m; P^m_{j,k}; w^m_{j,k})$
    $n \leftarrow n + 1$
else ▷ τ_j and τ_k define the *basic path*
  while $0 < n \leq \zeta$ **do**
    while $0 \leq m \leq n^{EC}_{i,j}$ **do**
      $d \leftarrow P^n_{j,k} - Q^m_{i,j}$
      if $d \leq 0$ **then** $break$
      $m \leftarrow m + 1$
    $points \leftarrow (w^{m-1}_{i,j}, m-1; Q^{m-1}_{i,j}; n; P^n_{j,k}; w^n_{j,k})$
    $n \leftarrow n + 1$
return $points$

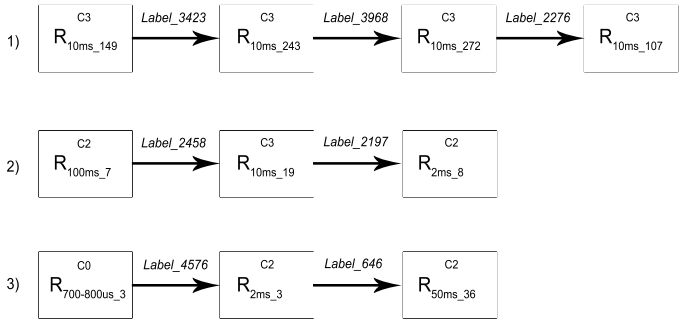

Figure 17. Amalthea effect chains

Effect chain 2. Since this effect chain presents a harmonic communication among all its tasks, then $\zeta = 1$ and

$$\alpha(EC) = \theta^1_{EC} + Q^2_{10ms,2ms} - Q^1_{10ms,2ms} + T_{100ms} = 10ms + 100ms + 100ms = 210ms$$

$$\rho(EC) = \theta^1_{EC} + P^1_{100ms,10ms} - P^0_{100ms,10ms} + T_{2ms} = 10ms + 100ms + 2ms = 112ms$$

.

Effect chain 3. This effect chain contains an ISR. Based on the given AMALTHEA model, the interrrupt is treated as a sporadic task with a period between 700us and 800us. Since 799us is the period length that maximizes the end-to-end age and reaction delays, with $n = 179$ and $n = 180$ respectively, and the EC is a *NHSC*, we obtain

$$\alpha(EC) = \max_{n\in[1,799]} \left\{\theta^n_{EC} + Q^{n+1}_{2ms,50ms} - Q^n_{2ms,50ms}\right\} + 799us = \theta^{179}_{EC} + Q^{180}_{2ms,50ms} - Q^{179}_{2ms,50ms} + 799us = 2798us + 50000us + 799us = 53,597ms$$

$$\rho(EC) = \max_{n\in[1,799]} \left\{\theta^n_{EC} + P^n_{799us,2ms} - P^{n-1}_{799us,2ms}\right\} + 50ms = \theta^{180}_{EC} + P^{180}_{799us,2ms} - P^{179}_{799us,2ms} + 50ms = 2461us + 50337us + 50ms = 102,798ms$$

V. Conclusion

In this paper we presented a solution for the FMTV challenge 2017. We proposed a formal implementation for the implicit and LET communication paradigms, analyzing the impact introduced in terms of memory footprint and communication delay. Moreover, we introduced a precise calculation of two effect-chain propagation delay semantics: *age* and *reaction*.

As a future work, we plan to extend the analysis of LET taking into consideration adaptive variable rate tasks (AVR) and interrupt service routines (ISR) .

A Java implementation is available for the algorithms described in this paper. Due to space constraints, details of results and tools are not covered in this paper but may be downloaded from the Github repository[1].

Acknowledgment

This work was supported by the HERCULES Project, funded by European Union's Horizon 2020 research and innovation program under grant agreement No. 688860

References

[1] A. Hamann, D. Ziegenbein, S. Kramer, and M. Lukasiewycz, "2017 formals methods and timing verification (fmtv) challenge," 2017, pp. 1–1. [Online]. Available: https://waters2017.inria.fr/challenge/

[2] A. Hamann, D. Ziegenbein, S. Kramer, and M. Lukasiewyz, "Demo abstract: Demonstration of the fmtv 2016 timing verification challenge," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–1.

[3] I. Sañudo, P. Burgio, and M. Bertogna, "Schedulability and timing analysis of mixed preemptive-cooperative tasks on a partitioned multicore system," in *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'16), in conjuction with the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016), Toulouse, France, July 2016*.

[4] The AUTOSAR consortium, "AUTOSAR: The Software Component Template." [Online]. Available: http://www.autosar.org

[5] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, Jan 2003.

[6] T. A. Henzinger, B. Horowitz, and M. Kirsch, "Embedded control systems development with giotto," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '01. New York, NY, USA: ACM, 2001, pp. 64–72. [Online]. Available: http://doi.acm.org/10.1145/384197.384208

[1] https://github.com/nachoSO/ChallengeWaters

[7] A. Biondi and G. Buttazzo, "Engine control: Task modeling and analysis," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 525–530.

[8] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*. IEEE Communications Society, 2009.