Deterministic Implementation of Periodic-Delayed Communications and Experimentation in AADL

Fabien Cadoret*, Thomas Robert*, Etienne Borde*, Laurent Pautet*, and Frank Singhoff†
*Institut Telecom;TELECOM ParisTech; LTCI - UMR 5141
Paris, France
firstname.lastname@telecom-paristech.fr
†Lab-STICC, UMR 6285, Université de Bretagne Occidentale, UEB
Brest, France
frank.singhoff@univ-brest.fr

***Abstract*—The design of hard real-time embedded systems has to comply with strong requirements with respect to time determinism and resource consumption. However, interacting tasks may induce pessimism in schedulability analysis or introduce significant overheads in memory usage.**

In this paper, we restrict the execution and communication models to enforce an efficient and predictable implementation. To ensure determinism, a message sent by an emitting task is delivered at its deadline. We take advantage of a wait-free specialized message queues to provide predictable and efficient implementation. The integration of such mechanisms is assisted by a model driven engineering framework[1].

***Keywords*-scheduling theory; real-time middleware; model driven engineering**

I. Introduction

Most of real-time embedded software applications involve concurrent execution of tasks that are not independent. In practice, these dependencies come from data flows or shared states between tasks (for instance shared hardware devices embodied by buffers). The implementation these constructs may vary depending on the expected inter-task interaction model (shared objects, message queues...).

In this context, a known issue with dependent tasks is that the implementation of their interactions deeply affects the design process, either decreasing the quality of schedulability analysis, or introducing significant overheads in terms of resources consumption, in particular memory and execution resources. These implementations can be classified according to their access methods to shared data objects: blocking or non blocking (lock-free [8], or wait-free [5] as usually admitted [1]). Each of these access methods raises particular issues, either on the side of schedulability analysis (increasing its complexity), or on the side of resources consumptions.

Blocking accesses, usually based on shared objects protected by locks, have a low impact in terms of memory consumption. The apriori validation of such mechanisms, due to over-approximation of Worst-Case Execution Times (WCET), leads to a sub-efficient usage of the execution resources. Moreover, such mechanisms often lead to handle priority inversion issues by on-line priority control or inheritance protocols. These mechanisms entail non-trivial scheduling anomalies [15], [4] that significantly increase the complexity of schedulability analysis.

Thus, to our knowledge, schedulability analysis are still considering rather coarse over-approximations of delays due to shared resources. Moreover, lock mechanisms themselves have an execution time that should not be neglected, especially when privileged execution domains are enforced by the execution platform (a mechanism now popular in real time operating systems). The Ravenscar profile [2] provides relevant guidelines to ensure a correct (*i.e.* without anomalies) usage of lock-based protected objects. Restrictions proposed in this profile ensure that associated schedulability analysis are not over-pessimistic, and that application timing is fully predictable.

In order to reduce the pessimism of schedulability analysis, non-blocking mechanisms have been studied [8], [1], [9], [5]. Non-blocking solutions are usually categorized into lock-free and wait-free mechanisms. Lock-free mechanisms deal with shared objects by retrying to access an object until it becomes available. This technique has a low impact on memory consumption, but increases significantly the WCET of tasks thus leading to an important overhead in terms of execution resources consumption. On the other hand, wait-free mechanisms deal with concurrent accesses by duplicating shared data. Helpers are provided to manage accesses to such data in case of interference.

Several interesting proposals have been made to find lower and upper bounds on memory consumption overhead when using wait-free implementations [9], [5]. Similarly, studies about the implementation of lock-free mechanism aim at reducing temporal overheads introduced by such approaches [1]. However, in the worst-case, such solutions introduce an overhead (both in terms of memory and execution resources) that is not acceptable for real-time embedded systems.

In this paper, we study an alternative track that adapts the

[1]This work was partially funded by the FUI/PARSEC project.

guidelines of the Ravenscar profile for non-blocking message passing communications. Our restrictions to the task and communication model aim at satisfying the following objectives:

Objective 1: Maintain schedulability analysis to an acceptable level of complexity when integrating overheads due to task communications.

Objective 2: Provide a fine grain evaluation of resource consumption in terms of memory and computation time for communication mechanism implementation.

Objective 3: Ensure that the overhead introduced by communications is reasonable compared to existing alternative approaches.

Indeed, wait-free or lock-free shared objects are designed envisioning rich usages scenarii (multiple readers and writers at the same time, dynamic message and task loads...). However, from our experience, in most hard real time embedded systems, these objects are used in simple usage scenarii. The complexity rather comes from the computation of the number of accesses and the restrictions on available resources. Our restrictions to the task and communication model are based on these considerations.

The contributions of this paper are:

1) a set of restrictions on inter-tasks communications enabling a wait-free implementation of shared message queues that would be fully predictable from the timing point of view, with a low storage overhead
2) a proof of concept of the usability of this implementation, using Model Driven Engineering (MDE) techniques to (i) capture the execution and communication semantics we rely on, (ii) automate the resources consumption analysis, and (iii) generate the implementation of wait-free accesses to the shared message queues

This paper is organized as follows: section II introduces the communication model. Then, we describe in section III our approach to implement the communication primitives without any locking primitives. We also provide in this section formula used to assess their cost in terms of memory and execution time. In section IV, we depict how this communication model can be automatically derived from task set specification to assist the evaluation of the schedulability analysis, and to generate the implementation. Section V compares our result to existing similar approaches. Section VI concludes this paper.

II. Task and communication Model

In next subsections, we describe the task and communication model we propose to rely on. Then, the three objectives described in section I are refined in terms of constraints on its implementation.

A. Task model with delayed communications

The key idea is to take advantage of existing dependencies between communications and task execution model. The task and communication model we present hereafter is built upon preemptive fixed priority scheduling of independent periodic tasks. It is extended with predictable message passing communication.

Task model: A system is made of a fixed set of periodic tasks, defined with the following attributes:

- T_i: refers to task i in the task set
- P_i: period of task T_i
- C_i: worst case execution time of task T_i
- D_i: deadline of task T_i

The period of a task is the fixed duration between two of its successive release times. In the sequel, we call a job, the execution of the sequence of statements started after each release time t, and completed before $t + D_i$.

We assume that periodic tasks are synchronous: all first task release times occur at the same time. For simplicity, we assume for all tasks that its deadline is smaller or equal to its period ($\forall i : D_i \leq P_i$).

Communication model: The communication model we consider in this paper is a variant of message passing communications modeled with ports.

- Communication links are modeled by ports and connections to enable the various configurations regarding the number of sender and receiver.
- A task can "put" a message on emitting ports executing the *send* statement.
- A task can "get" a message on receiving ports executing the *receive* statement.
- Any message, put on an emitting port p, is eventually received on receiving ports connected to p.

This model is refined to ensure deterministic communications between tasks. We recall the notion of delivery time defined as the time at which a message is available on a receiving port. Then, we consider the following additional constraints:

- Emitting and receiving ports are statically bound to tasks.
- During each job, exactly one message is sent on each emitting port of the task.
- A message sent by a job is delivered to the receiving task at its release time following the emitting job deadline. More formally, a message sent to T_i by a job released at time t is considered delivered at $\lceil \frac{t+D_j}{P_i} \rceil \cdot P_i$.
- Any message delivered to T_i at t, should be removed from the receiving port (i.e. can no longer be received) at $t + D_i$. The message is said outdated.
- Messages delivered simultaneously to a task are received in the order of emitting job deadlines. When emitting job deadlines are simultaneous, a pre-defined order noted $\prec$, e.g. task priorities, is used.

The model is said "periodic-delayed" as messages are periodically sent and their delivery is delayed until sender job deadlines. Such a communication model can be modeled in AADL [16]. AADL is an architectural description language used in several modeling processes dedicated to the design and the verification of embedded systems.

To illustrate this task and communication model, we consider the time-line depicted in Figure 1. It illustrates communications between three tasks: T_1,T_2, and T_3, with $i = 1..3, P_i = D_i$, and $P_1 = 5$, $P_2 = 7$, $P_3 = 10$. T_1 and T_2 send periodically messages to T_3 according to the communication model described above.

As illustrated on this figure, exactly one message is sent during each task job for T_1 and T_2. Notice that $a1$,$a2$,$b1$ are only delivered at time 10. Yet, $b2$ will not be delivered before 20, even though job $T2.2$ (that produces $b2$) already finished its execution when $T3.2$ starts in this scenario. This model allows ensuring deterministic time for message reception independently of task interleaving and actual execution time. Notice also, that $b1$ has to be returned to the receiver before $a2$ to enforce a deterministic order on message from the point of view of the receiver. Finally, $a1, b1, a2$ would be discarded at completion time of $T3.2$, even if these messages were not consumed during this job.

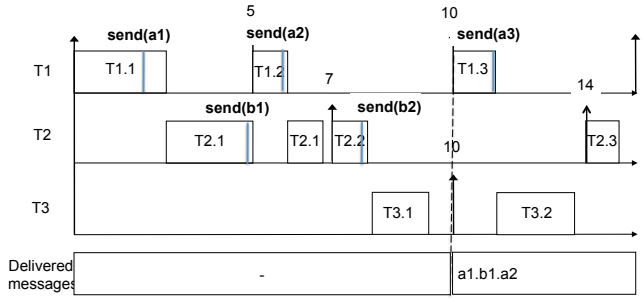

Figure 1. Sample execution

B. Validation strategy

Given this communication model, the three objectives of the previous section can be refined in simpler sub-goals.

To reach *objective 1* (keep schedulability analysis tractable):

- The implementation of communications should not used any lock.
- The execution time and call occurrences of communication statements should be bounded. It should ensure computations due to communication are bounded.

To reach *objective 2* (evaluate resource consumption overhead), we propose to:

- instantiate statically (as opposed to at execution time or at initialization time) the communication mechanisms and data structures implementing our model
- represent these additional components in a global model of the system architecture

To reach *objective 3* (master the overhead due to tasks interactions), it is necessary to:

- determine when messages need to be stored, can be discarded from the data structure, and should be delivered. It is necessary to determine the amount of memory required by the structure
- ensure exclusive access to data with minimal or no data duplication, and computational overhead

In next section, we present an implementation of the communication model including proof elements w.r.t. its correctness. Then, a discussion is carried out to explain how those sub-goals are fulfilled.

III. Deterministic and Efficient Implementation

As previously explained, we focus on a restricted communication model for tasks interacting on the same computing node. Implementing such a communication model boils to implement a queue structure that enforces the correct message queuing and dequeuing policy. Our main concern is actually to enable concurrent accesses of multiple writers to the message queue of a single reader without using a lock.

A. The wait-free queue structure and logic

Both the queue structure and logic determine the characteristics of the implementation with respect to message safety (no message loss), and message delivery order.

In the remainder of the paper, we assume messages are of fixed or bounded size. Thus, we focus on the number of messages in the queue structure and not on its memory size. As outdated messages can be discarded, there is actually a maximum number of stored messages. In the section III-C, we describe how to statically compute this number to prevent message loss and wrong delivery order.

Thus, the implementation of the wait-free queue can be reduced to a fixed array of slots to store messages. This array must have a sufficient but fixed capacity, denoted L. Slots are uniquely identified by integer indexes ranging from 0 to $L-1$. Each time, *send* and *receive* operations are invoked, indexes are computed to determine which slot should be written or read. Yet, these operations as well as index computations may be executed concurrently. Our first objective is to ensure no message is lost or corrupted despite this concurrency.

A way to prevent conflicts between *send* operations is to compute an unique index in which a message can be stored. This specific index ensures an exclusive access to the designated slot for a known time interval. If the exclusion interval is chosen large enough, then write accesses can be fully ordered. Write operations only overwrite outdated messages (e.g. messages no longer needed).

Index computation can be modeled as follow : each time a *send* operation is invoked a sequence number i_m

is provided such that the message is stored in the array at index i_m $modulo$ L. Thus, the safety requirement (no lost message) can be specified as follow: for any pair of distinct messages if their sequence numbers are equal modulo L, it means that at least one of them is outdated.

Producing sequence numbers that are not affected by the concurrency between several *send* operations is necessary. Because the queue structure has a single reader, the reading order can be used to build these sequence numbers. Thus, successive *receive* operations result in accesses to slots of consecutive indexes.

We first provide formulas to compute these sequence numbers as a strictly increasing sequence of integers (e.g. unbounded). Then we describe the conditions under which the safety requirement holds.

B. Send and Receive slot selection

As said previously, slot selection is done thanks to a numbering of messages. This numbering is built according to the order in which they must be received, as specified by the communication model. In practice, the sequence number generated for a message is the cardinal of the set of messages received strictly before it.

The content of this set has to be clearly identified. Some notations are introduced to ease its description.

- In the model, a queue q has a set of sender tasks denoted PT_q, and a single receiver task, T_r.
- Deadline of T_j in its k^{th} job is denoted $JD(j,k)$.
- An arbitrary $\prec$ order is used to order distinct tasks.
- $|X|$ is the cardinal of X, X being a set.

We first formalize the reception order.

Definition 1 (Reception total order):

Let m_1, m_2 be two messages sent toward the same queue. T_{s1} (resp. T_{s2}) sends its message m_1 (resp. m_2) in its k_1^{th} job (resp. k_2^{th}).

m_1 is received strictly before m_2 iff both statements hold

- The k_1^{th} job deadline of T_{s1} is lower or equal to the k_2^{th} job deadline of T_{s2}: $JD(s1,k_1) \leq JD(s2,k_2)$ (condition $C1$),
- Job deadlines are distinct, or are identical and T_{s1} precedes T_{s2} in the fixed order $\prec$ (condition $C2$).

This definition provides the guidelines to compute the cardinal of the set of messages received before message m_2. First, we compute the number of pairs $(s1,k_1)$ such that $JD(s1,k_1) \leq JD(s2,k_2)$. This number corresponds to the cardinal of the set of jobs satisfying $C1$. Then, we subtract the number of jobs violating condition $C2$.

Jobs satisfying $C1$: Let $SEJD(q,t)$ the cardinal of the set of messages received before t on q. In other words, it corresponds to the cardinal of the set of sender job deadlines earlier to t. Remember that $D_i \leq P_i$.

$$SEJD(q,t) = \sum_{j \in PT_q} \left(\left\lfloor \frac{t - D_j}{P_j} \right\rfloor + 1 \right) \qquad (1)$$

To find the slot index used in a *send* operation invoked by the k^{th} job of T_j, we compute $SEJD(q, JD(j,k))$, the number of messages sent by jobs with deadlines expiring simultaneously or before T_j job deadline.

Jobs violating $C2$: We now consider the set of tasks having a job with a deadline equal to T_j job deadline. We designate as $Followers$ of task T_j all the tasks from this set which are successors to T_j according the fixed order $\prec$.

Let $\mathbb{N}$ be the set of naturals (positive integers). Note that if $\frac{t-D_s}{P_s}$ is a natural, it means that there exists k such that $t = k \cdot P_s + D_s$.

We first define the $Collide$ function to identify whether a task has a job deadline at t.

$$Collide(s,t) = \begin{cases} 1 & if \quad \frac{t-D_s}{P_s} \in \mathbb{N} \\ 0 & otherwise \end{cases}$$

Then $Followers(q,j,k)$ is defined as follows:

$$Followers(q,j,k) = \sum_{s \in PT_q, j \prec s} Collide(s, JD(j,k))$$

When a *send* operation is invoked by the k^{th} job of T_j, its sequence number is noted $MSN(q,j,k)$, (i.e. Message Sequence Number) and defined as follows:

$$\begin{aligned} MSN(q,j,k) \quad = \quad & SEJD(q, k \cdot P_j + D_j) \\ & - Followers(q,j,k) \end{aligned} \qquad (2)$$

Slot Selection for Send: Hence, the corresponding message is stored in slot $MSN(q,j,k)$ $modulo$ L. Note that this slot selection does not require to share any state variable between sender tasks, and will thus be implemented without lock.

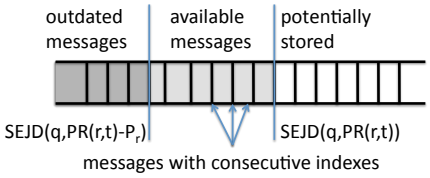

Figure 2. Status of messages w.r.t to their indexes at t

Slot Selection for Receive: Implementing the receive primitive requires to specify message availability. $PR(r,t)$ denotes the latest release time of T_r with respect to t (i.e. the release time of the currently activated job):

$$PR(r,t) = \lfloor \frac{t}{P_r} \rfloor \cdot P_r$$

Figure 2 depicts the status of messages with respect to their sequence numbers at a given time t. Messages available for reception at t must have been sent from job with deadlines expiring before $PR(r,t)$. Thus, their indexes are smaller or equal to $SEJD(q, PR(r,t))$. Yet,

messages corresponding to jobs with deadlines smaller than $PR(r,t) - P_r$ are outdated (see dark grey area on figure 2). Finally, *receive* deliver messages with MSNs ranging from $SEJD(q, PR(r,t) - P_r) + 1$ to $SEJD(q, PR(r,t))$.

Next section describes how to compute the size of the fixed message array in order to ensure the safety property.

C. Correctness and Overhead analysis

In this section, we determine the size of the fixed array and we demonstrate why the sequence numbers MSN allow to safely access the stored messages. We first determine the range of sequence numbers used at t to store non-outdated messages. If the size of this interval is strictly smaller than the array size, then each slot is associated with at most one message produced before t.

The lower bound has already been computed in previous section. Let T_r be the receiving task. The lower bound of sequence numbers used to store messages not outdated at t is $SEJD(q, PR(r,t) - P_r)$+1, denoted $ILB(q,t)$.

The upper bound requires more attention. Let T_s be a sending task. If its *send* operation is invoked at t, then to comply with our communication model, the message delivery occurs at $PR(s,t) + D_s$. Hence, its sequence number is lower or equal to $SEJD(q, PR(s,t) + D_s)$. Let D_{max} be the largest deadline of sending tasks. Note $SEJD(q,t)$ is a monotonically increasing function w.r.t. t. As $PR(s,t) \leq t \leq PR(r,t) + P_r$, $SEJD(q, PR(s,t) + D_s)$ is always lower or equal to $SEJD(q, PR(r,t) + P_r + D_{max})$, denoted $IUB(q,t)$ later.

As a consequence of the two previous results, the size of the interval of indexes used to store non-outdated messages is lower than the maximum of $IUB(q,t) - ILB(q,t) + 1$. If this size is chosen as the $max_{t \in \mathbb{R}^+}(IUB(q,t) - ILB(q,t) + 1)$, it leads to a sufficient condition to ensure message safety. The exact value can be computed by numeric solvers. We can also provide a very close bound of this formula but its proof is beyond the scope of this paper. Hence, we provide another formula for a slightly higher value that can be easily computed by hand. We use the fact that $\lfloor \frac{a+b}{c} \rfloor - \lfloor \frac{a}{c} \rfloor \leq \lfloor \frac{b}{c} \rfloor + 1$, for positive integers a, b, c. Let apply this result to $a_j = PR(r,t) - P_r - D_j$, $b_j = 2 * P_r + D_{max}$, and $c_j = P_j$.

$$
\begin{aligned}
SEJD(q, PR(r,t) + P_r + D_{max}) &= \sum_{j \in PT_q} \left\lfloor \frac{a_j + b_j}{c_j} \right\rfloor \\
SEJD(q, PR(r,t) - P_r) &= \sum_{j \in PT_q} \left\lfloor \frac{a_j}{c_j} \right\rfloor
\end{aligned}
$$

$$
IUB(q,t) - ILB(q,t) + 1 \leq \sum_{j \in PT_q} \left(\left\lfloor \frac{2 * P_r + D_{max}}{P_j} \right\rfloor + 1 \right) \quad (3)
$$

We can assess how pessimistic this bound is. $SEJD(q, 2 * P_r)$ corresponds to a number of messages exactly produced over two first periods of the receiving task. It is lower to the optimal value of the array size. In our example of figure 1, the lower bound is 7 when the bound from equation 3 is 10.

We provide all the guidelines required to instantiate the wait-free queue such that both safety and orderliness are achieved. Next section presents how these guidelines are integrated in an automatic generation process.

IV. Modelling and analyses with a MDE approach

In previous sections, we have presented the theoretical and algorithmic contribution of this paper: (i) a deterministic and analysable execution and communication model; and (ii) the mathematical definition for configuring wait-free queue that implements this execution and communication model. The Architectural Analysis Description Language (AADL) has been selected in order to capitalize this work and integrate it into previous works on assisted software engineering for embedded systems [3]. In practice, this contribution has been integrated in RAMSES[2], a model transformation and code generator framework for AADL.

A. AADL in a Nutshell

AADL is an architecture description language that was mainly experimented in the domain of real-time embedded systems. First class elements of the language are strongly-typed components that belong to different categories (*e.g.* process, thread, data, subprogram, etc...). Interactions between components are expressed thanks to predefined features (*e.g.* data ports, event data ports, event ports, data accesses, etc) that come with a standardized semantics (yet mostly expressed using natural language). AADL components are composed hierarchically according to standardized composition rules. Features of subcomponents have to be connected to represent interaction links. Finally, properties can be associated to any AADL element (component, subcomponent, feature, connection, etc) to precise the execution and communication semantics.

The reasons for choosing AADL lies in its capability to model both our design model (execution and communication model), and a model of its implementation [3]. Indeed, the execution and communication semantics presented in section II-A matches with the semantics of a subset of AADL. Briefly, this subset can be defined as follows:

- The "Dispatch_Protocol" property is set to "Periodic" for each thread component: tasks are periodic,
- The "Period", and "Deadline" properties are set for each thread component, such that $deadline \leq period$,
- The "Timing" property is set to "Delayed" for output ports of tasks: messages are sent at deadline.

Note that the default value of the AADL "Output_Rate" property already states that one message is produced per

[2]Refinement of AADL Models for the Synthesis of Embedded Systems: http://penelope.enst.fr/aadl

activation of the producer tasks. Similarly, we use "AllItems" as the default value for the property "Dequeue_Protocol", which means that all the messages available at release time of the recipient will be considered as consumed at the end of its job.

To conclude, the execution and communication model defined in section II-A can be modeled in AADL using standard components and property sets. However, even though this semantics exists from the beginning of the AADL definition, there exists no code generation process that (i) implements this semantics with a wait-free queue, and (ii) produces an AADL representation of this implementation. We propose a pragmatic solution to these issues in next subsection.

B. Assessing Memory and execution time overhead

When deploying the implementation of the queue presented in section III, we ineluctably introduce overheads. Considering memory foot-print, the overhead is mostly due to the size of messages and the array used to store them (*i.e.* data structure of the wait-free queue). The worst-case memory and time consumption due to the execution of helper functions ($delivered$, $send$, $receive$) can be assessed for a given task set. The computation method is very similar for time and memory resources. We focus on execution time in the remainder of this section.

Considering task execution time, each sending task would execute once the $send$ primitive but several times the $receive$ primitive. Thus, assessing the overhead due to the execution of $send$ can be done using WCET evaluation techniques on the generated code.

On the other hand, the overhead on the receiving side requires to evaluate the execution time of the $receive$ primitive, but also the number of calls to this primitive. In case the AADL model provides a description of the subprograms that are called in a thread, this number can be computed from this behavioral model. Otherwise, we can use the fact that, for a given task, the $receive$ primitive will be called at most $max_{k\in\mathbb{N}}(SEJD(q,(k+1)\cdot P_i) - SEJD(q,k\cdot P_i) - 1)$.

As both overheads vary due to task set characteristics, (i) we propose to model the task set structure in AADL, and (ii) we derive automatically a refined version of this architecture in which the components used to implement the queue are modeled. Finally, the information they are carrying are collected to assess overheads.

We illustrate the refinement from design model to implementation model through a brief presentation of both models (see listing 1 and listing 2) w.r.t. the example of Figure 1.

The design model (listing 1) defines three tasks `T1`, `T2` and `T3`, with their period, deadline and priority (see lines 3 to 8 on listing 1). Tasks `T1` and `T2` (of type `producer`) have an out event data port (`p_out`) with value "delayed" for property "timing" (see lines 16-17 on listing 1). This property value is used to select the delayed delivery semantic

```
1  process implementation proc.impl
2  subcomponents
3    T1: thread producer.impl {Period => 5 ms;
4      Deadline => 5 ms; Priority => 3; };
5    T2: thread producer.impl {Period => 7 ms;
6      Deadline => 7 ms; Priority => 2; };
7    T3: thread consumer.impl {Period => 10 ms;
8      Deadline => 10 ms; Priority => 1; };
9  connections
10   cnx1: port T1.p_out -> T3.p_in;
11   cnx2: port T2.p_out -> T3.p_in;
12 end proc.impl;
13
14 thread producer extends Periodic_Thread
15 features
16   p_out: out event data port MyType
17     { Timing => Delayed; };
18 end producer;
19
20 thread implementation consumer.impl
21 calls
22   seq: { call1: subprogram consumer_job; };
23 connections
24   cnx1: parameter p_in -> call1.param_in;
25 end consumer.impl;
26
27 subprogram consumer_job
28 features
29   param_in: in parameter MyType;
30 properties
31   Execution_Time => 1 ms .. 3 ms;
32   Source_Stack_Size => 40 Kbytes;
33 end consumer_job;
```

Listing 1. Design Model: Example

of messages according to AADL semantics. Output ports of `T1` and `T2` are both connected to the input port (`p_in`) of task `T3` (see lines 10-11 on listing 1).

When task `T3` (of implementation `consumer.impl`) is activated, it executes the subprogram `consumer_job` and passes the available message delivered on port `p_in` to the subprogram that perform computations (see lines 22 and 24 on listing 1). Finally, we provide in listing 1 the definition of subprogram `consumer_job`: note that this subprogram comes with a definition of its WCET (3 ms), which by extension is the WCET of task `T3`.

The specification model is used to instantiate communication components to assess their cost in terms of resources consumption. Listing 2 illustrates the result of the model transformation that produces the AADL model of the implementation of the wait-free queue resulting from the task set specified in listing 1. The implementation model provides a refinement of the design model, in which data structures and subprograms have been instantiated in order to explicitly model communication actions. In the mapping between the design and implementation models, ports and connections are transformed into data accesses, data subcomponents, and subprogram calls. This set of model elements represent both the wait-free queue and the accessors to this queue.

In listing 2, communication components are:

- `T3_array`, that implements the data structure of the wait-free queue itself, with a size computed according to equation 3 described in section III-C,
- `delivered_p_in_T3`, that returns the indexes (`First` and `Last`) of available data in the queue using an implementation of the SEJD function presented in section III-B.

Note that the values of WCET and memory footprint instantiated in this model result from (i) the computation of resource consumption per execution of the generated code (done with usual techniques to assess memory footprint and execution time), and (ii) the computation, for the worst-case, of the number of execution of this code. In the specification model, calls to the *receive* primitive are not explicitly represented but hidden in the implementation of the $consumer\_job$ subprogram. As a consequence, the resource consumption of the `delivered_p_in_T3` subprogram also takes into account the resource consumption due to multiple executions of the *receive* primitive.

```
1  process implementation proc.impl
2  subcomponents
3   ... -- unchanged subcomponents e1, e2 and r;
4   T3_array: data MyType[10]; -- message buffer
5  connections
6   ...
7   c3: data access T3_array->T3.p_in_array;
8  end proc.impl;
9
10 thread proc_T3 extends Periodic_Thread
11 features
12  p_in_array: requires data access MyType[10];
13 end proc_T3;
14
15 thread implementation proc_T3.impl
16 subcomponents
17  RLB: data Integer {Initial_Value=>('-1');};
18  RUB: data Integer {Initial_Value=>('-1');};
19 calls
20  s:{call1:subprogram delivered_p_in_T3;
21     call2:subprogram consumer_job; };
22 connections
23  c1:data access p_out_array->call1.Array;
24  c2:parameter RUB->call1.Last;
25  c3:parameter RLB->call1.First;
26  c4:parameter call1.Val->call2.param_in;
27 end proc_T3.impl;
28
29 subprogram delivered_p_in_T3
30 features
31  Val: out parameter MyType;
32  Array: requires data access MyType[10];
33  Last: in out parameter Integer;
34  First: out parameter Integer;
35 properties
36  Execution_Time => 50 .. 60 us;
37  Source_Stack_Size => 5 Kbytes;
38 end delivered_p_in_T3;
```

Listing 2. Implementation Model: Example

Finally, this model can be submitted to an existing method available to check the schedulability of an AADL model [17]. Of course, this type of method is very accurate on the implementation model, since it represents tasks that are independent from a scheduling point of view.

V. Related works

The task and communication model proposed in this article is time-triggered: computations and communications are triggered at predefined dates. From this time-triggered computational model, we have shown how to implement it in order to ease schedulability analysis.

In the sequel, we present approaches that proposed similar computational models, and the means to implement.

One of the very first time-triggered approaches was MARS (Maintainable Architecture for Real-Time Systems) [11]. MARS was the seminal work of TTA (Time-Triggered Architecture) [10]. The MARS approach is based on a concept of global clock among a distributed system. This clock allowed to produce a total ordering of all events in the system, and thus allowed an efficient analysis of distributed real-time systems which is usually difficult to achieve. As opposed to our contribution, TTA users must explicitly design the system while we propose to automatically map parts of it into TTA-like components.

Giotto is another example of time-triggered approach [7]. Systems are modeled with a language providing the notions of modes, sensors, actuators and periodic tasks. Tasks communicate through ports in a deterministic manner: time at which data are sent or read through a port are statically known as they are predefined by the Giotto language. Giotto illustrates the independency of scheduling and communications in time-triggered system: with Giotto, scheduling of tasks is chosen at system compilation time thanks to compiler directive, and then, is not related to time-triggered communications. The queue mechanism is not provided by Giotto, only the last message sent would be kept for each sender in each receiving period.

Similarly to Giotto, OASIS [12] provides both time-triggered oriented communications and a tool chain to implement applications. OASIS applications are written with an extended C language called Psi-C. Psi-C allows programmers to express when communications will occur. Compared to our approach, OASIS requires a detailed description based on Psi-C. OASIS also requires a dedicated execution platform. In contrary, we chose a Ravenscar-like execution platform. As most of the real-time operating system have Ravenscar features, we can expect that our implementation models generator can be used with a larger number of execution platforms.

Finally, the work presented here is strongly related to AADL [16] and its seminal work, Meta-H[18]. AADL is a very rich architecture language and it proposes several computational models: AADL provides Ravenscar communications, time-triggered communications and many others such as remote procedure call or ARINC 653 communication

mechanisms. An AADL model may mix all those mechanisms, allowing designers to model complex architectures. In this case however, analysis can be difficult to achieve. This motivated our choice to only focus on a time-triggered computation model. In [6], the author shows how to perform schedulability and dimensioning analysis when AADL models handle time-triggered communications. In the context of synchronous languages and GALS architectures, [13] propose to use the Signal language and its Polychrony framework to achieve both verification and implementation with the same type of AADL models. One of the challenge raised by this type of approach is to ensure that the synchronous programs will be correctness-preserving implemented in an asynchronous execution platform [14]. In this work, as most of the approaches presented in this section, time-triggered communications are independent of thread scheduling: with Polychrony, AADL thread scheduling is supposed to be computed off-line by a dedicated tool. Implementation models are generated with the standard Polychrony tools in contrary to our approach in which, thanks to a MDE process, generation of the implementation models can be adapted by specific generators to cope with specific requirements.

VI. Conclusion

In this paper, the proposed contribution enable resource efficient integration of a wait-free specialized message queues in a predictable way.

In previous work, either the schedulability analysis or resource consumption were an issue. In order to avoid this difficult choice, we followed an alternative path restricting the underlying communication model. The restriction is still relevant for many communication patterns requiring a strong determinism on communications. Section III and IV detailed how Objectives 2 and 3 are fulfilled to our point of view. The core concept is to avoid synchronization taking advantage of our ability to predict how messages need to be stored.

In this contribution, we focus on message-based communication between tasks sharing the same memory space. In future works, we want to use similar approach to handle sharing buses to extend the approach to actually distributed architecture. Another track of investigation is to loosen the hypothesis made on the communication model for which the approach can be adapted at a reasonable cost.

References

[1] J. H. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Trans. Comput. Syst.*, vol. 15, no. 2, pp. 134–165, May 1997. 1

[2] A. Burns, B. Dobbing, and G. Romanski, "The ravenscar tasking profile for high integrity real-time programs," in *Reliable Software Technologies Ada-Europe*, ser. Lecture Notes in Computer Science, L. Asplund, Ed. Springer Netherlands, 1998, vol. 1411, pp. 263–275. 1

[3] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet, "Design patterns for rule-based refinement of safety critical embedded systems models," in *ICECCS*, 2012. 5

[4] Y.-S. Chen, L.-P. Chang, T.-W. Kuo, and A. K. Mok, "Real-time task scheduling anomaly: observations and prevention," in *SAC*, 2005, pp. 897–898. 1

[5] H. Cho, B. Ravindran, and E. D. Jensen, "Space-optimal, wait-free real-time synchronization," *IEEE Trans. Comput.*, vol. 56, no. 3, pp. 373–384, Mar. 2007. 1

[6] P. H. Feiler, "Efficient embedded runtime systems through port communication optimization," in *ICECCS*, 2008, pp. 294–300. 8

[7] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Proceedings of the IEEE*. Springer-Verlag, 2000, pp. 166–184. 7

[8] M. Herlihy, "A methodology for implementing highly concurrent objects," *ACM Trans. Program. Lang. Syst*, vol. 15, no. 5, pp. 745–770, 1993. 1

[9] H. Huang, P. Pillai, and K. G. Shin, "Improving wait-free algorithms for interprocess communication in embedded real-time systems," in *Proceedings of the General Track of USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 303–316. 1

[10] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," in *Proceedings of the IEEE*, vol. 91, Jan. 2003. 7

[11] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The mars approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, Jan. 1989. 7

[12] S. Louise, M. Lemerre, C. Aussaguès, and V. David, "The oasis kernel: A framework for high dependability real-time systems," in *HASE*, 2011, pp. 95–103. 7

[13] Y. Ma, H. Yu, T. Gautier, J.-P. Talpin, L. Besnard, and P. L. Guernic, "System synthesis from aadl using polychrony," in *in the 2011 Electronic System Level Synthesis Conference San Diego, California, USA. June, 2011.*, pp. 1–6. 8

[14] D. Potop-Butucaru, Y. Sorel, R. de Simone, and J.-P. Talpin, "From concurrent multi-clock programs to deterministic asynchronous implementations," *Fundam. Inform.*, vol. 108, no. 1-2, pp. 91–118, 2011. 8

[15] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *WCET*, 2006. 1

[16] SAE, *Architecture Analysis & Design Language v2.0 (AS5506)*, September 2008. 3, 7

[17] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand, "Investigating the usability of real-time scheduling theory with the cheddar project," *Real-Time Systems*, vol. 43, no. 3, pp. 259–295, 2009. 7

[18] S. Vestal and P. Binns, "Scheduling and communication in metah," in *IEEE Real-Time Systems Symposium, Raleigh Durham, NC*, 1993, pp. 194–200. 7