A Hierarchical Architecture for Time- and Event-Triggered Real-Time Systems [⋆]

Jorge Real[1(✉)], Sergio Sáez[2], and Alfons Crespo[1]

[1] Instituto de Automática e Informática Industrial
[2] Instituto Tecnológico de Informática
Universitat Politècnica de València
Camí de Vera, s/n, 46022 València, Spain
{jorge,ssaez,alfons}@disca.upv.es

Abstract. This paper proposes an architecture for combining the execution of time- and event-triggered real-time task sets. This makes it possible for the designer to choose the most appropriate mechanism depending on the role and nature of each task in the system. The proposed architecture allows one to choose the priority levels at which time- and event-triggered tasks are executed. This gives the designer an additional degree of freedom to make compromise decisions upon contradicting timing requirements, such as granting reduced jitter and at the same time providing prompt service to non-periodic events, for example. The proposed model is accompanied with a Ravenscar implementation of the time-triggered scheduler and a library of utilities for specifying time-triggered schedules and reusing time-triggered task patterns.

Keywords: Real-time systems. Time-triggered scheduling. Ravenscar tasking profile. High-integrity systems. Embedded systems.

1 Introduction

Real-time computer systems can be broadly described in terms of activities, or tasks, that must be executed along time, together with timing restrictions such as their frequency of execution or the deadlines by which the tasks must be completed. Compliance with those timing restrictions must be guaranteed in advance, which in turn requires a deterministic, analysable method to schedule the execution of tasks along time.

There are two major approaches for designing and scheduling real-time systems, namely time- and event-triggered. In time-triggered (TT) systems, a pre-elaborated, offline static plan dictates the exact points in time when each task must execute and for how long it is allowed to execute. In event-triggered (ET) systems on the other hand, tasks are released for execution as a consequence of events, whose times of occurrence are not necessarily known in advance. These events may indeed be related with time,

[⋆] This work has been partly supported by Spanish Government and FEDER funds (AEI/FEDER, UE) under grant (TIN2017-86520-C3-1-R) (PRECON-I4); and by European Commission project AQUAS (ECSEL-JU, Contract 737475).

such as the recurrent expiration of a timer cycle, but they can also be asynchronous, such as entering a particular system state, or receiving particular sensor or user inputs. At run time, the scheduler makes online decisions about which task to execute, typically based on the priorities assigned to tasks whose activation events have occurred and need be handled.

Both TT and ET designs exhibit advantages and disadvantages with respect to each other, and none can be deemed superior in general. TT systems excel in terms of run-time predictability, they are schedulable by construction, but they have a higher design complexity and lack flexibility. Inconveniently, timing aspects may need to be reflected in the functional code of TT tasks, for example when their execution time is too long to be fitted in a single slot of the schedule and they need be split into several pieces of code. TT systems can also deal with asynchronous events, but they need some form of sampling to detect their occurrence, or more sophisticated techniques to make the system react faster to them.

ET systems on the other hand, allow designers to conveniently separate functional and timing concerns, for example a preemptive scheduler can split the execution of a long task at run time based on its priority. An event-triggered system naturally incorporates non-periodic tasks as well. However they are not predictable, and require specific schedulability analysis techniques. This unpredictability affects also timing aspects, introducing variable release delays due to interference from higher-priority tasks. Variable delays must be avoided for example in digital control applications, since they impact the performance of control algorithms. They are also harmful for synchronous communication between elements of a distributed system. More extensive comparisons between time- and event-triggered systems are given in [1, 2].

This paper proposes an architecture for the combined execution of TT and ET task sets, with the aim to obtain the best of both approaches in systems formed by tasks with different sensitiveness to release delays. Control and communication tasks would fall in the subset of delay-sensitive tasks and they would be scheduled with an offline, TT plan; whereas less delay-sensitive tasks (human-machine interface, logging, etc.) would be subject to an online scheduler, typically a priority-based one.

The idea of combining TT and ET tasks has been proposed before in the form of a technique called *slot shifting* [3]. As the name suggests, the technique consists in shifting the slots of an offline TT plan at run time to accommodate aperiodic and sporadic requests, while preserving deadline guarantees for the TT workload. This is accomplished by using precise information about the actual arrival of non-periodic events and knowledge about the slack time present in the offline schedule. The purpose of slot shifting is to speed up the handling of sporadic and aperiodic events, but this is at the expense of slightly varying the placement of slots in the time-triggered plan. We are interested in a technique that grants minimal and ideally constant release jitter, for improved digital control and communications, hence our approach does not modify the offline TT plan.

Combining the ET and TT approaches makes it possible to choose which release mechanism to apply to which application tasks. This allows designers to use the most appropriate one depending on the role and nature of each task in the system. The proposed architecture is hierarchical, because it allows one to choose the priority levels at which TT and ET tasks are executed. This gives the designer an additional degree of freedom to make compromise decisions upon contradicting timing requirements, such as granting reduced jitter and at the same time providing prompt service to non-periodic events, for example. We include a Ravenscar implementation of the proposed approach, together with a library of helping patterns. The Ravenscar profile [4] is a subset of the tasking model of the Ada programming language [5], restricted in favour of determinism and analysability for high-integrity real-time systems that could be certifiable to the highest integrity levels. In a previous publication [6], we proposed a similar architecture for a non-restricted Ada tasking model. In [7] we presented a first version of the Ravenscar implementation. This paper describes an evolved version of that work, proposing a richer model, solving pending issues related with the use of shared resources among the TT and ET subsets, and introducing a new implementation of a library of utilities to make the approach more usable.

The rest of this paper is organised as follows. Section 2 describes the general system model. Section 3 details the model for the time-triggered part of the proposed architecture. Section 4 proposes an Application Programming Interface to use the TT scheduler and example patterns are given in Section 5. Section 6 describes a library of utilities to ease the usability of the proposed architecture, including an interface for the description of TT plans and use of predefined TT task patterns. Experimental results measuring release jitter are presented in Section 7. Finally, Section 8 presents our conclusions.

2 System Model

We consider a real-time computer system is composed by a set of tasks, each task undertaking a particular function of the system. Tasks have timing requirements to meet, especially a deadline by which they must complete their execution. During system operation, tasks are triggered repeatedly to execute their function. According to their triggering mechanism, tasks may be either time- or event-triggered, depending on their particular role and timing requirements. A time-triggered task is triggered when a static, offline schedule so dictates. An event-triggered task is triggered upon the occurrence of its associated release event. The whole set of tasks is hence split in two subsets, the TT and the ET subsets. Typically, control and communication tasks will be in the TT subset since they require precise and predictable timing, whereas aperiodic, sporadic and also jitter-tolerant periodic tasks will belong in the ET subset.

Both kinds of tasks, ET and TT tasks, are scheduled using a priority based scheduler. Ultimately, they are all implemented by priority-scheduled tasks. A high priority level is reserved for TT tasks, where they are scheduled according to the current TT plan. The plan determines at which points in time a task must be released and for how long they are allowed to run. Gaps in the plan, i.e. time intervals with no scheduled TT task, represent intervals for the execution of lower-priority, ET tasks.

The TT plan will normally be executed at the highest priority level, to minimise the release delays of TT tasks and to avoid interference from other tasks during their execution. However, it is also possible to reserve the highest priority level for ET tasks if there is a requirement for prompt handling of one or more particular asynchronous events. For simplicity, we will assume everywhere else, without loss of generality, that the highest-priority level is used exclusively by TT tasks.

From the schedulability analysis point of view, the TT plan may be regarded as a source of interference for the lower-priority ET tasks: a higher-priority TT plan can be analysed as a transaction with offsets whose interference on lower-priority tasks can be quantified [8].

Tasks of both subsets may share data in mutual exclusion. The protected object abstraction of the Ada programming language [5], together with Ceiling Locking policy, as enforced by the Ravenscar profile, guarantee consistent access to shared resources among concurrent tasks. Resource sharing between TT tasks does not need the use of mutual exclusion protocols, but this is only true in the case where TT tasks complete their execution in a single time slot. If a TT task needs to use several slots to complete its job, then mutual exclusion must be guaranteed in its access to shared resources. Data sharing between ET and TT tasks definitely needs protection because there may be preemptions between them.

Our proposed model does not impose any restrictions or additions to the scheduling or implementation of the ET workload. We will therefore discuss the TT model in more detail. The following section describes the TT model, i.e., the components and semantics of a TT plan.

3 Time-Triggered Model

A time-triggered plan consists of an ordered sequence of non-overlapping *time slots*. Each slot has a given duration, in seconds. The total duration of the plan is the sum of its slot durations. At run time, the plan is repeated cyclically. Figure 1 is the graphical representation of a plan with 14 slots numbered from 0 to 13. The total total duration is 80 ms and the duration of each slot is indicated by the time scale: the duration of slot 0 is 5 ms, slot 1 is 8 ms, etc.

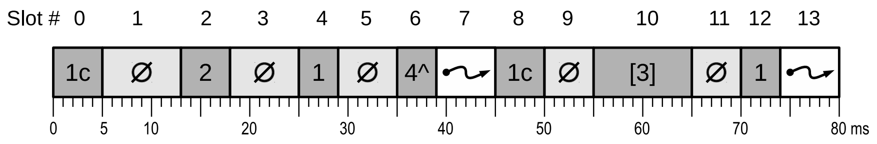

Slot # 0 1 2 3 4 5 6 7 8 9 10 11 12 13
1c
Ø
2
Ø
1
Ø
4^
1c
Ø
[3]
Ø
1
0 5 10 20 30 40 50 60 70 80 ms

Fig. 1. A 14-slot time-triggered plan. Slots with sequence numbers 2, 4 and 12 are *regular* slots for works 1 and 2, as indicated; slots 0 and 8 are *continuation* slots for work 1; slot 6 is a *sync* slot with identifier 4; slot 10 is an *optional* slot for work 3; slots 7 and 13 are *mode-change* slots. The rest (1, 3, 5, 9, 11) are *empty* slots.

At run time, a TT scheduler is invoked whenever the next slot is planned to start, which is when the current slot finishes[1]. There are several types of slots. The type of slot and its duration define a time window that is either dedicated to executing a particular TT task (running with a high priority), or made available for the preemptable execution of lower-priority tasks. We define the following types of slots:

- An ***empty slot*** defines a time interval during which no TT task is planned for execution. Empty slots therefore represent time windows that are granted to lower-priority tasks. In Figure 1, slots 1, 3, 5, 9 and 11 are empty slots.
- A ***mode-change slot*** is similar to an empty slot in the sense that it has no associated TT task to execute. Additionally, it defines a time in the plan where it

[1] Note that it is possible to optimise the TT scheduler implementation avoiding invocations of the scheduler for empty slots starting just after a work slot whose corresponding TT activity was completed earlier than the end of the slot.

is possible to substitute the current plan with a new one. By placing mode-change slots in the plan, the designer determines the exact times when a mode change can occur. If there is a pending mode-change request to process at the start of a mode-change slot, then the new plan will start at the end of the mode-change slot. This ability to change mode at defined points in time introduces a degree of flexibility that off-line, static schedules do not possess by nature. In Figure 1, slots 7 and 13 are mode-change slots, represented with a curved arrow.

The previous types of slots, empty and mode-change, exclude the execution of any TT task. Their duration is therefore available for non-TT workload and they actually represent the only time windows that are granted to lower priority tasks, from a schedulability analysis point of view. The following types of slots, regular and optional, imply the activation of a task at the TT level.

– A ***regular slot*** defines a time interval (the slot duration) reserved for the execution of a TT task (a *work*). The TT task is denoted by a Work_Id, a positive integer that identifies the particular work to execute during the slot duration. The underlying TT scheduler will make the work start to execute as soon as feasible after the start time of the slot. In Figure 1, slots 2, 4 and 12 are regular slots corresponding to works with Work_Id 1 or 2 as indicated.
  The duration of a regular slot must be sufficient, by design, to accommodate the worst-case execution time of the work it serves. Since all systems need be designed with some slack, the TT work will most often finish earlier than the slot duration, leaving the extra time to lower-priority tasks. If, due to a design or run-time error, a work overruns its regular slot, this is considered a severe fault and an exception is raised. In our implementation, the scheduler raises Ada's Program_Error exception, since an overrun violates the schedulability assumptions of the model.
  A TT task must always be ready to use its allocated regular slots in the plan. Failing this, the scheduler will raise Program_Error as well. The following type of slot is more permissive in this regard.
– An ***optional slot*** is like a regular slot except that it can be omitted. A TT task may decide to use or not to use an assigned optional slot in the plan. If it does use it (the task is ready to start when the optional slot arrives) then it has the same semantics as a regular slot, including overrun control at the end of the slot. But if the task is not waiting for the optional slot when the slot starts, it is not considered an error and the slot duration is made available for ET tasks. In Figure 1, slot 10 is an optional slot for work 3, indicated with square brackets around the Work_Id. Optional slots are useful for tasks that may or may not require to use their allocated slot, such as a communication task when it has nothing to say; or a sporadic task whose activation event has not occurred.

Both regular and optional slots may have the additional property of *continuation*. Slots with this property can be used for scheduling long TT tasks that may require more than one slot in the plan for completing each of their iterations.

– A regular slot with continuation, or simply a ***continuation slot***, is a regular slot in which the associated work does not need to complete by the end of the slot, and it can continue its execution using future slots. Failing to finish by the end of a continuation slot is not an overrun. Instead, the work is held at the end of the slot and continued at the start of the next slot in the plan that is marked with its Work_Id. There may be one or more consecutive continuation slots for a given work, defining a sliced sequence of slots. Overrun will only be checked when the plan reaches a regular slot for this work after the sequence o continuation slots. We refer to the last, non-continuation slot of a sequence as the *terminal slot* of the sequence. A sliced sequence is therefore represented in the plan by a series of continuation slots ended with a terminal slot. In Figure 1, continuation slots are marked with a Work_Id plus a letter *'c'*, indicating *continuation*. There are two sliced sequences for Work_Id number 1 in Figure 1, one defined by slots 0 and 4 (continuation and terminal) and the other using slots 8 and 12. This type of slot is useful to split a large TT task into smaller pieces in a way that is essentially transparent to the task code. Continuation slots require asynchronously holding and resuming a running TT task, which in turn requires support from the runtime system. This is the reason why our implementation of the TT scheduler is an extension of the runtime system.

Special care must be taken when defining the semantics for holding a sliced TT task. Holding a task must not occur while it is accessing shared data, which could be in an inconsistent state at the end of the continuation slot. A hold operation may need to be deferred until the end of a shared data operation. This deferral could delay the effective start of the next slot, which goes against our requirement for reduced and fixed release jitter. To avoid any impact on the start time of the next slot, continuation slots can use a *padding* time that would absorb the worst-case duration of a critical section. The padding time is the anticipation of the hold operation with respect to the end of the current continuation slot. If a padding time is defined, the hold operation will be advanced to the end of the slot minus the padding time. Knowing the worst-case duration of critical sections, one can use it as the padding time of the continuation slots of a sliced sequence, hence keeping the cost of the hold operation within the duration of the continuation slot, and not introducing delays on the start of future slots.

– An optional slot with continuation, or simply an ***optional continuation slot***, combines the properties of continuation and optionality. With this type of slots, one can plan a sliced sequence that is either taken or not, as a whole. An optional sliced sequence consists of an optional continuation slot, then zero or more continuation slots, and then a terminal slot.

Finally, we define a slot type that is intended for the use of ET tasks, rather than TT works.

– A synchronisation slot, or a ***sync slot*** for short, is one that can be used by ET tasks to synchronise their execution with the time base of the plan. ET tasks may need to adjust their activation to the plan pace, for example if they need to consume data produced by the TT workload. This type of slot allows an ET task to wait for the appropriate time in the plan to do so. An ET task can wait until a sync slot arrives and then continue execution at its assigned priority level. Sync slots have an associated identifier, but not related with any TT task. The task identifier of a sync slot is called the Sync_Id, instead of Work_Id, to make it clear that the arrival of a sync slot does not release a TT task, but an ET task that runs at a priority different from the priority of the TT plan. There is no requirement that an ET task be waiting for the slot when it arrives. If an ET task requests to wait for the arrival of a sync slot that has already occurred, then it will be immediately released. The semantics is similar to waiting for a time in the past. Multiple arrivals of unused sync slots are however not queued.

Figure 2 summarises the types of slots in the model. A time slot is the root abstraction and it has a single attribute, common to all types of slots: the slot duration. Empty and mode-change slots are concretions of the time slot abstract type that need no further attributes. A sync slot is a time slot with an added attribute, the Sync_Id, an identifier of the ET task that uses the sync slot. Work slots are the class of slots that may be used by TT tasks. Work slots add three attributes to the basic time slot: a Work_Id, identifying the TT task that uses the slot; a boolean *Is_Continuation* that determines the semantics of the slot to be regular (with overrun detection) or continuation (with hold); and a Padding time for the case of continuation slots. These two attributes (*Is_Continuation* and *Padding*) are writable because regular and optional slots can be dynamically changed into their continuation counterparts while the slot is being used. The usefulness of this feature will be illustrated in Section 5.

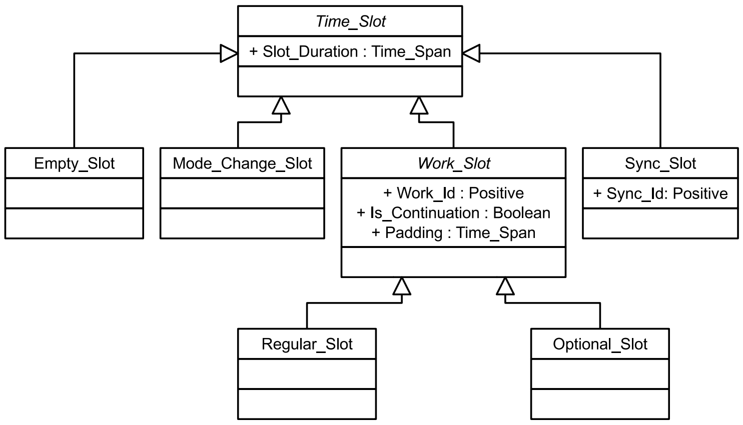


Fig. 2. Types of slots defined in the TT model. *Continuation* and *optional continuation* slots are particular modes of regular and optional slots, respectively, for which the Is_Continuation attribute is True.

4 Application Programming Interface

The model described in sections 2 and 3 is supported by package XAda.Dispatching.TTS, where TTS stands for Time-Triggered Scheduling. XAda stands for *eXtended Ada*, a package for experimental extensions to the Ada standard libraries. In addition to this package, we have modified some runtime packages of the *GNAT 2019 CE* cross compiler for ARM, in particular, the STM32F4 Ravenscar runtime, full version; and we provide use libraries that will be described in Section 6. All the software involved in this project is available from [9].

Package XAda.Dispatching.TTS offers the following seven subprograms for using the TT scheduler:

Set_Plan Procedure to enforce a TT plan. It takes the TT plan as the only parameter.
The first call to Set_Plan immediately starts the given TT plan. Subsequent calls

will have effect at the end of the next mode-change slot in the currently running plan. Calling Set_Plan when there is a pending mode change replaces the previous request – calls are not queued. Set_Plan can be called from any task, either TT or ET. During a mode-change slot, only ET tasks can call Set_Plan, since there is no TT work running in the slot. The latest call to Set_Plan during a mode-change slot will take effect at the end of the slot.

Wait_For_Activation This procedure is called from TT tasks to wait for the arrival of their next slot in the plan. It takes a Work_Id as an input parameter and returns in a second parameter the start time of the slot in the TT plan. This time is useful for measuring jitter, for example. Callers to this procedure execute the code after the call at the TT priority level. A call to Wait_For_Activation implicitly informs the scheduler that the TT task has completed its last activation, which facilitates overrun detection.

Wait_For_Sync A caller to this procedure waits until the arrival of a particular sync slot in the plan, or continues immediately if the sync slot has already occurred in the current cycle of the plan. It is intended for use from ET tasks waiting for a sync slot, hence it accepts a Sync_Id as an input parameter, not a Work_Id. A second parameter returns the slot start time in the TT plan, as in Wait_For_Activation. Arrivals of several sync slots are not queued.

Continue_Sliced This parameterless procedure informs the TT scheduler that the currently running TT task wants to continue running under a sliced regime, i.e., using continuation slots. This call effectively transforms the current regular slot into a continuation slot, without altering the plan, and just for the current instance of the regular slot. The procedure has no effect if the current slot is already a continuation slot. A call to Continue_Sliced from an ET task, or from a TT task other than the corresponding with the current work slot is an error.

Leave_TT_Level A parameterless procedure used from TT tasks to inform the TT scheduler that they want to continue executing outside of the plan for a while, under the priority-based scheduler and at a different priority to the TT priority (normally lower). This is useful to execute parts of the task that are not subject to strict jitter requirements or that may be difficult to integrate in the TT plan.

Get_Last_Plan_Release This function returns the last start time of the currently running TT plan.

Get_First_Plan_Release This function returns the time when the current plan started, i.e., the start time of the first slot in the first iteration of the current plan. The times returned by Get_First_Plan_Release and Get_Last_Plan_Release are useful to relate the current real-time clock value with the TT plan time. During the first iteration of the plan, both functions return the same value.

XAda.Dispatching.TTS is a generic package. It takes the number of Work_Ids and Sync_Ids as generic parameters, as well as the TT plan priority level. In this manner, all data structures are adjusted to the size required by the application using the package. For example, waiting tasks (both ET and TT tasks) use Ada's suspension objects, one suspension object per task. The genericity of the package allows us to use just the amount of suspension objects needed by the application.

Package XAda.Dispatching.TTS also defines the types needed for representing TT plans. There are types for representing the hierarchy of slots depicted in Figure 2 and a TT plan is an array of those slots. Slot types are tagged records, hence they are extensible by the user. Listing 1 corresponds to the specification part of the TTS package defining the slots and TT plan types. In Section 6, higher-level libraries are proposed for further increasing the usability of the TT scheduler, both in terms of easing the definition of TT plans and for reusing common TT programming patterns. The next section discusses a number of useful patterns for TT tasks.

Listing 1. Generic parameters and data types declared in the specification of TTS.

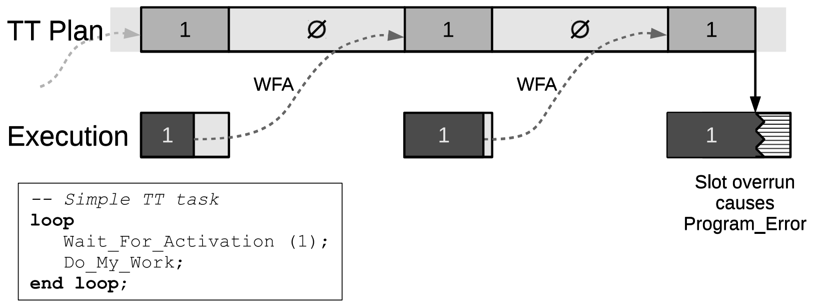
```
pragma Profile (Ravenscar);
-- Context clauses omitted
generic
   Number_Of_Work_IDs : Positive;
   Number_Of_Sync_IDs : Positive := 1;
   TT_Priority        : System.Priority := System.Priority'Last;
package XAda.Dispatching.TTS is

   -- TT tasks use a Work_Id of this type to identify themselves when they call the scheduler
   type TT_Work_Id is new Positive range 1 .. Number_Of_Work_IDs;

   -- ET tasks use a Sync_Id of this type to identify themselves when they call the scheduler
   type TT_Sync_Id is new Positive range 1 .. Number_Of_Sync_IDs;

   -- An abstract time slot in the TT plan.
   type Time_Slot is abstract tagged record
      Slot_Duration : Ada.Real_Time.Time_Span;
   end record;
   type Time_Slot_Access is access all Time_Slot'Class;

   -- An empty time slot
   type Empty_Slot is new Time_Slot with null record;
   type Empty_Slot_Access is access all Empty_Slot'Class;

   -- A mode change time slot
   type Mode_Change_Slot is new Time_Slot with null record;
   type Mode_Change_Slot_Access is access all Mode_Change_Slot'Class;

   -- A sync slot
   type Sync_Slot is new Time_Slot with
      record
         Sync_Id : TT_Sync_Id;
      end record;
   type Sync_Slot_Access is access all Sync_Slot'Class;

   -- An abstract work slot
   type Work_Slot is abstract new Time_Slot with
      record
         Work_Id         : TT_Work_Id;
         Is_Continuation : Boolean := False;
         Padding         : Ada.Real_Time.Time_Span := Ada.Real_Time.Time_Span_Zero;
      end record;
   type Work_Slot_Access is access all Work_Slot'Class;

   -- A regular slot
   type Regular_Slot is new Work_Slot with null record;
   type Regular_Slot_Access is access all Regular_Slot'Class;

   -- An optional work slot
   type Optional_Slot is new Work_Slot with null record;
   type Optional_Slot_Access is access all Optional_Slot'Class;

   -- Types representing/accessing TT plans
   type Time_Triggered_Plan is array (Natural range <>) of Time_Slot_Access;
   type Time_Triggered_Plan_Access is access all Time_Triggered_Plan;

   -- API procedures and private part not shown here
end XAda.Dispatching.TTS;
```

5 Task Patterns

The model and API described in sections 2 to 4 open the possibility for implementing a number of useful task patterns. This section describes patterns for both TT tasks, to be executed during work slots in the plan (refer to Figure 2), and ET tasks that have some interaction with the plan, although they execute during non-work time slots. All TT tasks run at the priority defined in the instantiation of package XAda.Dispatching.TTS, whereas ET tasks run at their individually defined priority.

5.1 Patterns Using Regular Slots

The simplest TT pattern we can think of is a task that accommodates all its execution time within the duration of one work slot. We call this a *Simple TT Task* pattern. The task structure is simply an infinite loop where it waits for the arrival of its next slot and then executes its sequence of statements, just before suspending itself again until the arrival of its next slot.



```
-- Simple TT task
loop
   Wait_For_Activation (1);
   Do_My_Work;
end loop;
```

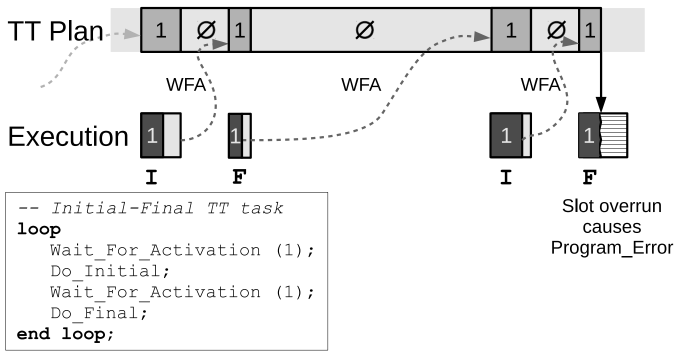
Fig. 3. Simple TT Task pattern. The task uses Work_Id = 1.

Figure 3 shows the pattern structure and represents the execution of three iterations of a simple TT task. The task calls Wait_For_Activation to wait for the arrival of the next regular slot in the plan for the TT task with Work_Id = 1 (for example). At the beginning of each slot, the scheduler releases the work and lets it run at the TT priority. Lower-priority tasks are therefore disabled to run while the simple TT task is running. When the task completes within the slot duration (first and second cases of Figure 3), it is put back to wait as a result of calling Wait_For_Activation. The time not used by the TT task (represented in lighter grey) becomes available for lower-priority tasks.

If, for whatever reason, the execution time of a simple TT task exceeds the slot duration, this is considered a hard deadline violation and a Program_Error exception is raised. The TT scheduler is in charge of making this check at the end of every regular slot.

A simple TT task may have its own local state, which is kept across successive releases. It can also share data with other simple TT tasks, because this type of task executes in mutual exclusion with other simple TT tasks (there are no overlapping slots). If the task needs to share data with pre-emptable PB tasks (or sliced TT tasks, as we'll see later), then it needs to do it in mutual exclusion, which in Ada is guaranteed by protected objects, for example. In such case, the TT task may experience blocking that must be taken into account when deciding the slot duration.

The *Initial-Final* pattern (I-F, for short) is for TT tasks that can be subdivided in two parts, both with strict jitter requirements and both requiring overrun control. This pattern can be easily obtained by sequential composition of two simple TT patterns, as shown in Figure 4, which shows the execution of two iterations of an I-F task. The loop is split in two parts, first the initial and then the final, and both use the same Work_Id when calling Wait_For_Activation – this is not necessarily a restriction, but just to keep the plan more human-readable. Note that the slots for the initial and final parts need not have the same duration. Overrun is checked for both the initial and the final parts.



```
-- Initial-Final TT task
loop
   Wait_For_Activation (1);
   Do_Initial;
   Wait_For_Activation (1);
   Do_Final;
end loop;
```

Fig. 4. I-F pattern, a TT task split in two parts, initial and final.

Regarding data sharing, the considerations we made for the Simple TT task pattern apply also to the case of the I-F task pattern. Note however that communication between the initial and final parts does not need to be protected if it is supported by data that is local to the task implementing the pattern.

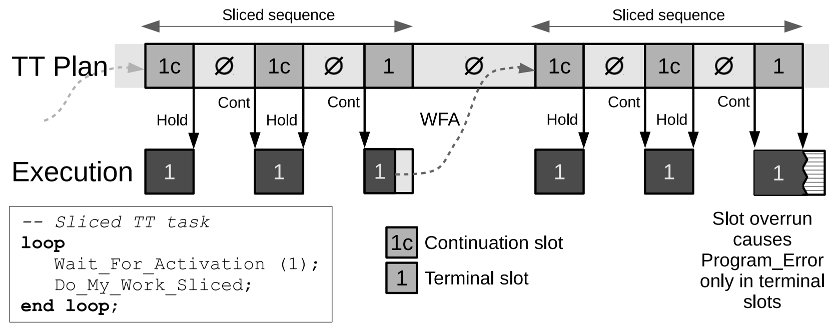
A further variation of I-F is the *Initial-Mandatory-Final* pattern (I-M-F, for short), which uses three consecutive regular slots to perform a logically related sequence of actions. This scheme is typical of digital control systems, where the initial part acquires some environment data, the mandatory part makes calculations using the acquired data, and the final part applies the results of the mandatory part to actuators in the controlled plant. The pattern structure would be defined by three calls to Wait_For_Activation using the same Work_Id, each preceding the statements of the initial, mandatory and final parts. The same considerations regarding overrun detection and data sharing we made for simple and I-F tasks, apply to I-M-F tasks: overrun is checked for each and every part of the task.

In fact, this pattern can be generalised to a form I-{M}-F, where there are one or more slots dedicated to execute overrun-controlled parts of the mandatory section. If we do not want overrun control in all the intermediate slots, then we need to use continuation slots (Section 5.2). If the mandatory part is not always really required, then one can resort to optional slots (Section 5.3).

.eps

5.2 Patterns Using Continuation Slots

Continuation slots allow one to break a long running TT task into slices in a way that is transparent to the application code, i.e., it does not require the task to make explicit calls to Wait_For_Activation at particular points of its execution. Slicing is dynamic rather than statically hardcoded. The TT scheduler will hold and resume the task at points dictated by the plan. Like a simple TT task, a sliced task just calls Wait_For_Activation and then performs its work. But execution of the work may be split across several consecutive continuation slots allocated in the plan to the sliced task. A sliced TT task requires one or more continuation slots, ending with a regular slot, the terminal slot of the sliced sequence. The terminal slot enforces overrun checking at the end of the sliced sequence.



```
-- Sliced TT task
loop
   Wait_For_Activation (1);
   Do_My_Work_Sliced;
end loop;
```

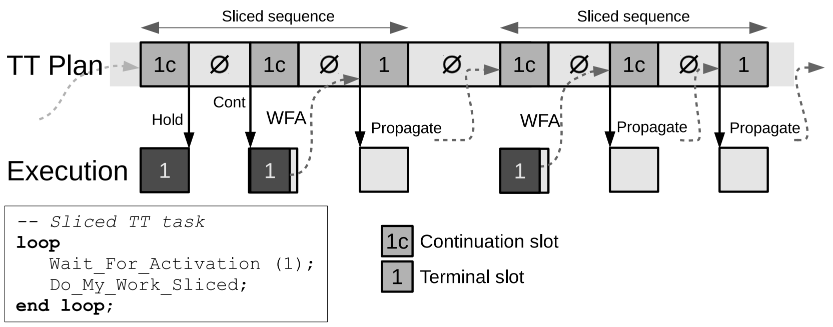
Fig. 5. Sliced task pattern.

Figure 5 shows the execution of two iterations of a *Sliced TT Task*. This task can make use of up to three consecutive slots. This is reflected in the plan as two continuation slots (marked '1c') and one terminal, regular slot (marked simply '1'). The pattern does not differ in structure from the simple TT task described in Section 5.1, but the semantics are totally different due to the use of continuation slots. In other words, one needs to look at the plan to distinguish a simple TT task from a sliced TT task.

In the two iterations shown in Figure 5, the task is held (by the TT scheduler) at the end of exhausted continuation slots, and resumed for continuation at the start of its next slot in the plan. Exceeding the duration of a continuation slot is not an overrun situation. In the first iteration, the task completes its work within its terminal slot (a regular slot). In the second, the task overruns the terminal slot, hence a Program_Error exception is raised.

There are indeed other possible situations in the execution of a sliced task. Given the pattern structure, the task will call Wait_For_Activation as soon as it completes the Do_My_Work_Sliced sequence of statements. This may well happen during a continuation slot, before the terminal slot of the sequence. The task may use up to three slots to complete an iteration, but it could actually take less. If that is the case, then the scheduler needs to take the call to Wait_For_Activation as a wait until the next sliced sequence, not necessarily the next slot marked '1', in this example. An early wait for activation must therefore be *propagated* until the next continuation slot after the next terminal slot, i.e., the next start of a sliced sequence.

Figure 6 shows the possibl.epse cases of early completion of a three-slot sliced task. In the first case, the task completes in the second slot of the sequence. When the terminal slot of this sequence arrives, the scheduler has to avoid waking up the task at the start of the slot and checking for overrun at the end, because this iteration of the task has completed. The effect of Wait_For_Activation must be postponed and the task must remain blocked waiting for the start of a new sequence of slots. This is marked as "Propagate" in Figure 6. In the second case, the task completes even earlier, during the first slot of the sequence. The call to Wait_For_Activation must be propagated twice. A new sliced sequence starts after the next regular (terminal) slot.



```
-- Sliced TT task
loop
   Wait_For_Activation (1);
   Do_My_Work_Sliced;
end loop;
```

Fig. 6. Two iterations of a three-slot sliced task completing before the terminal slot, requiring propagation of early Wait_For_Activation calls.

The need to hold and resume a running task has implications that are relevant when the sliced task shares data with other TT or ET tasks. Since the task can be held asynchronously, this data sharing needs to be protected by some mutual exclusion means, such as Ada's protected objects in our case. But holding the task while it is running a protected action is not acceptable, and letting it finish a long protected action could enlarge the release jitter of the next slot. Our approach to face this issue is to give the TT scheduler's internal Hold and Continue operations the semantics of the homonymous subprograms defined in the Ada (non Ravenscar) package Ada.Asynchronous_Task_Control. In particular, if a task is held while it is executing a protected operation, then the effect of Hold is posponed until the end of the protected action.

Postponement of Hold may delay the start of the next slot, which contradicts our requirement for short (and fixed) release jitter for TT tasks. To mitigate or even avoid blocking the start of the next slot during the completion of an ongoing protected operation, there are several possible ways to proceed:

1. Protected operations should be as short as possible, to minimise the blocking time they introduce. Even so, this does not really solve the issue.
2. One can use empty slots after every continuation slot to absorb the potential blocking time on TT tasks. However, this may be difficult or impossible for certain plans

and still requires to take the potential extra time into account in the schedulability analysis.

3. Charge the cost of Hold to the continuation slot used by the sliced task. Instead of letting the hold operation execute beyond the end of the continuation slot, hold the task earlier than the end of the slot, with sufficient time to absorb the cost of Hold, even when the task is held while executing a protected operation. This is the reason for the existence of the *padding* attribute for continuation slots, described in Section 3. Figure 7 graphically describes the concept of padding.

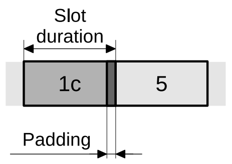
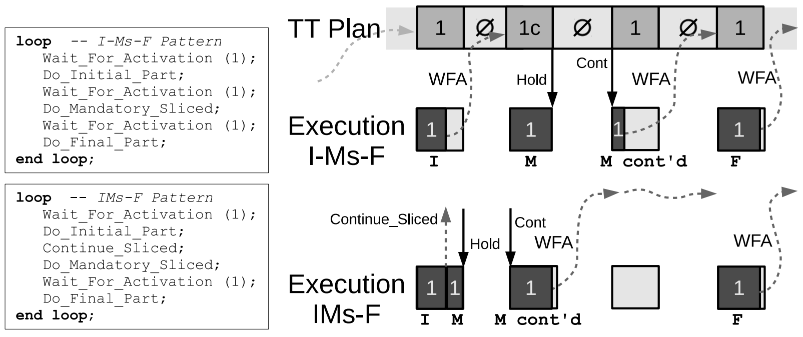


Fig. 7. Concept of padding time. In this example, the cost of Hold is charged to the continuation slot '1c' and not to the next regular slot with Work_Id = 5.

Sliced sequences can also be combined with parts supported by regular slots. The *Initial-Mandatory_Sliced-Final* pattern (I-Ms-F) is a variant of the I-M-F pattern where the mandatory part is sliced. The *InitialMandatory_Sliced-Final* pattern (IMs-F, note the missing dash between the 'I' and 'M' parts) is a slight, though important modification of I-Ms-F that allows the mandatory part to start executing immediately after the initial part, without waiting for the next slot in the plan. Both patterns have the same representation in the plan, taking one regular slot for the initial part (so that it is subject to overrun control), then one or more continuation slots ending with a terminal slot for the sliced mandatory part, plus one regular slot for the final part.

The IMs-F pattern requires using the scheduler's Continue_Sliced API subprogram. Since IMs-F allows the mandatory sliced part to start as soon as the initial part is done, during the first regular slot, we are effectively transforming the semantics of the Initial part's regular slot into that of a continuation slot. The scheduler must therefore be informed of the termination of the initial part so that, if the initial part is not done by the end of the slot, then there is an overrun; but if it has completed, then the slicing regime has started and the hold/continue mechanism has to apply to the already started sliced mandatory part.

Figure 8 shows these two patterns (I-Ms-F and IMs-F) for a sliced mandatory part of two slots. The top line represents the TT plan, common to both patterns. The structures of the patterns are shown to the left. The middle row represents a normal execution of an I-Ms-F task. After completing the initial part before the end of the first slot, the task calls Wait_For_Activation, hence delaying the start of the mandatory part to the next slot. Since the first slot is regular, the initial part runs under overrun control. The sliced mandatory part takes the next continuation slot and a part of the second continuation slot and then waits for the arrival of the terminal slot, which it uses to execute the final part. In the IMs-F pattern, at bottom of Figure 8, the TT task calls Continue_Sliced as soon as it completes its initial part. If the scheduler has not received this call by the end of the first slot, then the initial part has overrun;


```
loop  -- I-Ms-F Pattern
   Wait_For_Activation (1);
   Do_Initial_Part;
   Wait_For_Activation (1);
   Do_Mandatory_Sliced;
   Wait_For_Activation (1);
   Do_Final_Part;
end loop;
```

```
loop  -- IMs-F Pattern
   Wait_For_Activation (1);
   Do_Initial_Part;
   Continue_Sliced;
   Do_Mandatory_Sliced;
   Wait_For_Activation (1);
   Do_Final_Part;
end loop;
```

Fig. 8. Two variants of tasks with a sliced mandatory part: I-Ms-F and IMs-F.

otherwise, the initial part was completed during the first slot and the running task is held/continued as a sliced subsequence of this pattern. The final part of the pattern requires a previous call to Wait_For_Activation, as already described for other patterns with a final part.

A final consideration regarding continuation slots and their use by sliced TT tasks is that mode changes should not occur in the middle of sliced sequences, because that would break their logic. This must be avoided by appropriate placing of mode-change slots in the plan. This restriction can be checked by static analysis of the plan, but detection at runtime, although not impossible, could impose an excessive overhead from the TT scheduler.

5.3 Patterns Using Optional Slots

As described in Section 3, an optional slot is like a regular slot, except that it can be omitted by the associated work. A task not showing up for an optional slot is not an error.

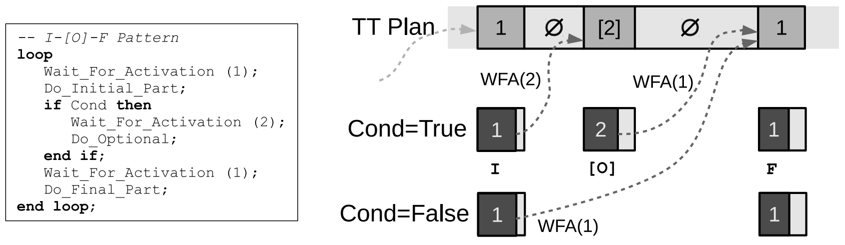
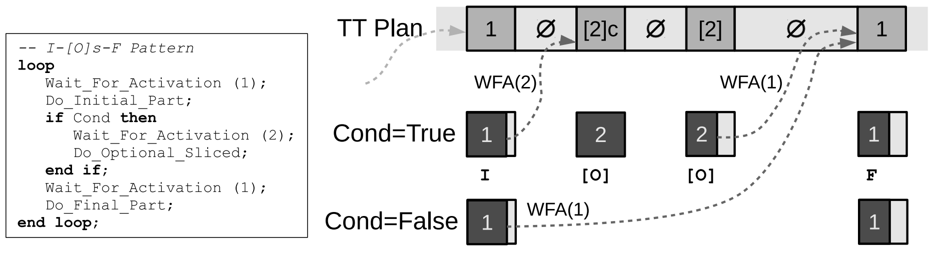

```
-- I-[O]-F Pattern
loop
   Wait_For_Activation (1);
   Do_Initial_Part;
   if Cond then
      Wait_For_Activation (2);
      Do_Optional;
   end if;
   Wait_For_Activation (1);
   Do_Final_Part;
end loop;
```

Fig. 9. The I-[O]-F pattern, with Initial, Optional and Final parts. The optional part requires its own, different Work_Id.

An example pattern is one with mandatory initial and final parts, plus an intermediate optional part. This would be an *Initial-Optional-Final* pattern, or I-[O]-F. A use example would be a control application that may need to apply an algorithm or not, depending on data collected during the initial part, such as an artificial vision application depending on the complexity of the image being processed. An I-[O]-F pattern is represented in Figure 9. The pattern first waits for a regular slot for the initial part. Then, depending on some condition (Cond), it may take the optional part when the optional slot arrives, or go straight to wait for the next regular slot for the final part. To make it clear that the task is waiting for a regular or an optional slot, the slots are marked with different Work_Ids, 1 and 2 in this example. A TT task may use more than one Work_Id — it is incorrect however that more than one task use the same Work_Id. From the I-[O]-F pattern, it is straightforward to derive the *Initial-Mandatory-Optional-Final* pattern (I-M-[O]-F), where there is a mandatory phase after the initial, and an optional part for refinement of a simple control action obtained in the mandatory part.

Optionality may also be a property of a sliced sequence, not necessarily of a single slot. Figure 10 represents the *Initial-Optional_sliced-Final* pattern (I-[O]s-F), where the optional part is actually a sliced sequence. The pattern is identical to the I-[O]-F case, but the difference resides in the plan. We still need to use two distinct Work_Ids, one for the optional part and another one for the compulsory parts. All slots in the sliced sequence are optional continuation slots, as described in Section 3, and the terminal slot of the sequence must be optional as well.



```
-- I-[O]s-F Pattern
loop
   Wait_For_Activation (1);
   Do_Initial_Part;
   if Cond then
      Wait_For_Activation (2);
      Do_Optional_Sliced;
   end if;
   Wait_For_Activation (1);
   Do_Final_Part;
end loop;
```

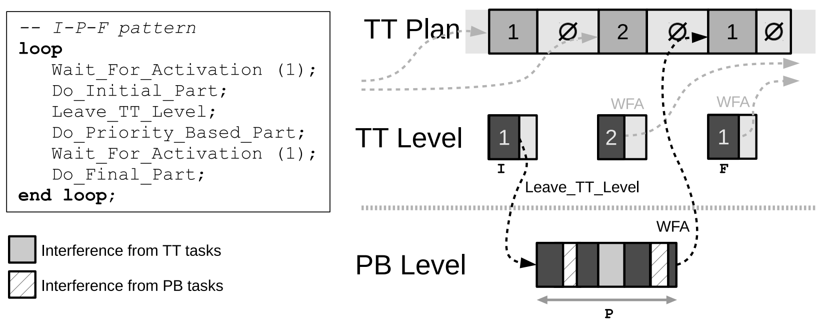
Fig. 10. The I-[O]s-F pattern, with Initial, Optional sliced and Final parts. In the plan, the optional sequence uses one (or more) optional continuation slots, ending with an optional slot as terminal. A distinct Work_Id is needed for the optional part.

5.4 Patterns Using Non-TT Parts

Although ET tasks are scheduled by the underlying priority scheduler and they do not require slots in the TT plan, it is possible to conceive task patterns that use a combination of TT and non-TT parts that operate in coordination with the plan. For example, the *Initial-Priority_based-Final* pattern (I-P-F) has initial and final parts with allocated work slots in the plan, plus an intermediate part that is scheduled in competition with the rest of ET tasks. The priority-scheduled part does not require slots, which facilitates the design of specially crowded plans. We just need to make sure that there is sufficient time available between the initial and final parts for the worst-case response time of the priority-based part, after applying appropriate schedulability analysis.

This ability to combine TT and non-TT parts within the same task pattern is supported specifically by the API subprogram Leave_TT_Level described in Section 4. TT tasks run at the priority specified for the TT plan. Using Leave_TT_Level, a TT task is asking the scheduler to lower its priority to another value.

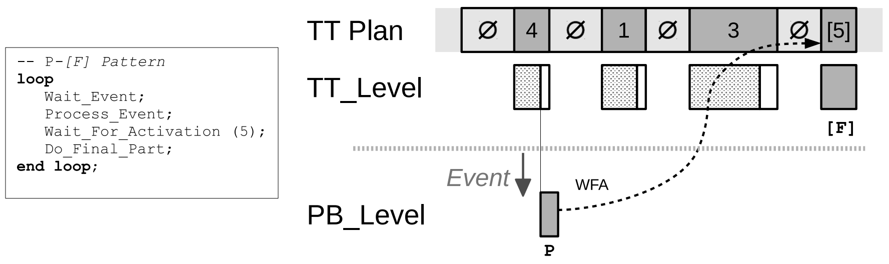
Figure 11 shows the I-P-F pattern and an example of execution. The TT plan includes slots only for the Initial and Final parts of the task.



```
-- I-P-F pattern
loop
   Wait_For_Activation (1);
   Do_Initial_Part;
   Leave_TT_Level;
   Do_Priority_Based_Part;
   Wait_For_Activation (1);
   Do_Final_Part;
end loop;
```

Fig. 11. The I-P-F pattern, with Initial, Priority-Based and Final parts. Calling Leave_TT_Level sends the task to a predefined priority level to continue executing under a priority based (PB) scheduling regime. The TT priority is recovered for the final part as a result of calling Wait_For_Activation

A combination of priority-based and TT parts is also used by the *Priority_based-_Final* pattern of Figure 12 (P-[F]). The priority-based part could be, for example, waiting for the occurrence of an event that it would then process at the required priority level. Only when the event occurs, the optional slot for the final part would be taken. It could then be used to communicate the results of processing the event, when the plan so requires. Note also that reaction to the event occurs as soon as the selected priority level is idle. This property favours prompt responsiveness to asynchronous events in combination with tight jitter for the TT workload.



```
-- P-[F] Pattern
loop
   Wait_Event;
   Process_Event;
   Wait_For_Activation (5);
   Do_Final_Part;
end loop;
```

Fig. 12. The P-[F] pattern, with a Priority-Based and an optional Final part.

5.5 Patterns Using Sync Slots

Sync slots are used by ET tasks to synchronise their releases with the TT plan. An ET task can simply wait for the arrival of a sync slot, without inheriting the TT priority when it arrives. Waiting for a sync slot via a call to Wait_For_Sync is similar to suspending the caller until that slot arrives. If the slot already occurred before the call and it was not sensed (no task had called Wait_For_Sync for that sync slot), then the task does not suspend. The semantics is thus as if the task executed an absolute delay until a time in the past. Successive arrivals of unused sync slots are not queued.

Consider as an example a low-priority task that collects data that has been logged during an iteration of the plan. The task then transmits those data using some non-real-time connection. All the task needs to know is when the data are available, possibly make a local copy, and then transmit at its own pace. But production of those data occurs at the pace of the TT plan, because it is produced by TT tasks (or TT parts of combined patterns). A sync slot is all the task needs to be notified of the availability of a new batch of data to transmit. Figure 13 shows the *Synchronised_Priority_Based-Final* pattern (SyncP-F). The task retrieves data produced by TT tasks 1, 2 and 3 during their respective regular slots. Availability of data is indicated in the plan with sync slot 6^.

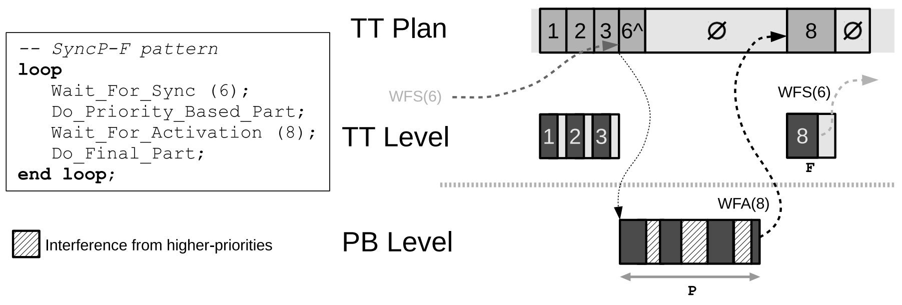

```
-- SyncP-F pattern
loop
   Wait_For_Sync (6);
   Do_Priority_Based_Part;
   Wait_For_Activation (8);
   Do_Final_Part;
end loop;
```

Fig. 13. The SyncP-F pattern.

5.6 Summary

The types of slots defined in Section 3 enable a number of useful programming patterns, both for implementing *pure* TT tasks and for coordinating ET and TT tasks in a predictable manner. Section 5 has described a number of these patterns using all types of slots defined beforehand and the API subprograms defined in Section 4. What follows is a list of aspects to consider at the time of using the described TT model.

Regular slots

- A given Work_Id must always be used by the same task. This applies also to the other types of TT work slots, continuation and optional.
- There is overrun control at the end of the slot, hence the slot duration must accommodate the worst-case execution time of the allocated TT work.

- Each slot is reserved to exactly one TT task. This differs from other models, such as the cyclic executive [10], where a slot can host more than one TT work. In our model, a slot overrun can always be traced back to a single TT task.

Continuation slots

- They provide a transparent means to split the execution of a long running TT task across several slots. There is no need for the programmer to break up the task into sub-actions that fit a given slot duration.
- Resource sharing between sliced TT tasks and other TT or ET tasks must be protected (mutually exclusive), given that sliced tasks are subject to holding and resuming, which is akin to preemption.
- A sliced sequence is formed by one or more continuation slots plus a final, regular slot, for overrun control of the entire sliced sequence.
- Mode-change slots should not be placed in the middle of sliced sequences.
- Using the API facility Continue_Sliced, one can change the execution regime of the running TT task during a regular slot, to make it start running sliced.

Optional slots

- They can be skipped: *no-show* is not an error when the slot is optional.
- They are adequate for optional parts of tasks and for the handling, at the TT level, of time-unrelated events, which may have occurred or not.
- A sliced sequence as a whole may be optional. This is indicated in the plan with a starting *optional continuation* slot.

Sync slots

- They are a means to insert points in the plan that are detectable by ET tasks, which allows them to operate in coordination with the plan.
- A given Sync_Id must always be used by the same task.
- The duration of sync slots may absorb the scheduler overhead, so that the next slot in the plan does not incur an added release delay.

Mode-change slots

- Calling the API subprogram Set_Plan places a mode-change request to the TT scheduler.
- A mode switch occurs if there is a pending mode-change request at the beginning of a mode-change slot (from a previously running TT task) or a request arrives during the slot (coming from an ET task). The latest call to Set_Plan will be in effect at the end of the mode-change slot.
- In the absence of a pending mode-change request, a mode-change slot is equivalent to an empty slot.
- A non-null duration of the mode-change slot is useful to absorb the scheduler overhead and avoid it to delay the intended start of the new plan. It may also be required to facilitate the schedulability of the ET workload during the mode-change transition [11] [12].

6 TT Utilities and Patterns Libraries

The package XAda.Dispatching.TTS, part of whose specification was given in Listing 1, is a means to use the proposed TT model. An instantiation of this generic package provides a scheduler that runs the TT workload at a given TT_Priority and whose memory footprint is adjusted to the strict needs of a system using a given number of Work_Ids and Sync_Ids. From that point, the programmer must then define the TT plans of the application and implement the TT tasks.

Defining TT plans — One plan must be defined for each operational mode of the application and for each processor when there is more than one. The plans must be of type Time_Triggered_Plan, an array of slots (see Listing 1). Slots must be created, each of its own type in the Time_Slot hierarchy (Work_Slot, Sync_Slot, etc.)

Implementing the tasks — The tasks behind the given Work_Ids and Sync_Ids must also be created. Each task will have a structure (at which points of the loop the task needs to use TT scheduler services) and functional code. The code implementing the task functionality is totally application-dependent, but the task structure can be reused across applications for tasks that follow the same pattern – e.g., one of those discussed in Section 5.

To facilitate the use of our Ravenscar implementation of the TT scheduler in these two facets, we have incorporated two helping libraries: TT_Utilities and TT_Patterns. The former helps with the definition of plans, making them also more readable in the source code, whereas the latter provides task types that implement TT patterns. We look at them to show how to use the TT scheduler in XAda.Dispatching.TTS. At the time of writing, they are very basic libraries; but expressive enough to build a complete example.

Listing 2 shows the main part of the specification of package TT_Utilities. The parameter of this generic package is an instance of package XAda.Dispatching.TTS. The utilities are therefore bound to a particular TT scheduler, with a given TT priority and number of work and sync identifiers. The enumeration type Slot_Type represents the kinds of slots defined in the model. Note that the identifier Terminal is just for readability, since terminal slots are regular slots by definition.

Listing 2. Specification of package TT_Utilities

```
with Ada.Real_Time, XAda.Dispatching.TTS;
use Ada.Real_Time;

generic
   with package TTS is new XAda.Dispatching.TTS(<>);
package TT_Utilities is

   -- Time_Slot kinds for building TT plans
   type Slot_Type is (Empty, Mode_Change, Regular, Terminal, Continuation,
                      Optional, Optional_Continuation, Sync);

   -- Time_Slot constructor
   function TT_Slot (
     Kind           : Slot_Type;
     Slot_Duration  : Time_Span;
     Slot_Id        : Positive := Positive'Last;
     Padding        : Time_Span := Time_Span_Zero) return TTS.Time_Slot_Access;

   -- Other declarations ...
end TT_Utilities;
```

The function TT_Slot creates and initialises a slot of a given kind and returns an access to it (the Ada form of a pointer) of type Time_Slot_Access. This is an *access-to-class-wide* type, hence it is compatible with all types of slots rooted at the abstract Time_Slot (refer to Listing 1). The parameters that must be passed to function TT_Slot are the Kind of slot to construct, one of those defined by Slot_Type, and its corresponding Slot_Duration, of the type Time_Span defined in the standard package Ada.Real_Time. Two additional parameters can be supplied if required: a positive integer Slot_Id to be used as Work_Id or Sync_Id of the slot, and an optional Padding time for continuation slots (refer to Figure 7). A precondition (not shown in Listing 2) ensures that the slot duration is not negative and that the Slot_Id value is within the range of Work_Id or Sync_Id, depending on what the identifier will be used for.

As an example, consider the plan depicted in Figure 14. It consists of 22 slots, numbered from 0 to 21 at their top. The time scale is in milliseconds. The plan takes 2 seconds per cycle. Slots marked 1 and 3 correspond to simple TT tasks. There are two sliced sequences in the plan, both following the IMs-F pattern described in Section 5.2. The first sequence, marked with continuous arrows, uses slots 4, 7, 11 and 15 with Work_Id = 2. The second sequence, marked with dashed arrows, uses slots 5, 9 and 13 and a Work_Id = 4. Slots 17 and 20 are used by an I-F task with Work_Id =5. The optional slot 19 is for a P-[F] task (see Section 5.4) that uses Work_Id = 6 for the final part. There are two sync slots, 3 and 12, marked with Sync_Ids 2 and 1, respectively. The rest are empty slots of varied durations and a final 80 ms mode-change slot. Listing 3 uses TT_Utilities to build the plan of Figure 14. The constant ms is just for readability, to avoid multiple calls to function Milliseconds in the plan declaration.

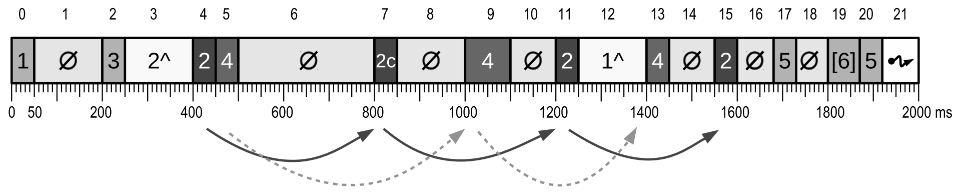


Fig. 14. Example TT plan.

Listing 3. Declaration of the TT plan of Figure 14.

```
ms : constant Time_Span := Milliseconds (1);

TT_Plan : aliased TTS.Time_Triggered_Plan :=
  ( TT_Slot (Regular,       50*ms, 1),  -- #00 Single slot for 1st seq. start
    TT_Slot (Empty,        150*ms   ),  -- #01
    TT_Slot (Regular,       50*ms, 3),  -- #02 Single slot for 2nd seq. start
    TT_Slot (Sync,         150*ms, 2),  -- #03 Sync point for sporadic task SP1
    TT_Slot (Regular,       50*ms, 2),  -- #04 Seq. 1, IMs part
    TT_Slot (Regular,       50*ms, 4),  -- #05 Seq. 2, IMs part
    TT_Slot (Empty,        300*ms   ),  -- #06
    TT_Slot (Continuation,  50*ms, 2),  -- #07 Seq. 1, continuation of Ms part
    TT_Slot (Empty,        150*ms   ),  -- #08
    TT_Slot (Terminal,     100*ms, 4),  -- #09 Seq. 2, terminal of Ms part
    TT_Slot (Empty,        100*ms   ),  -- #10
    TT_Slot (Terminal,      50*ms, 2),  -- #11 Seq. 1, terminal of Ms part
    TT_Slot (Sync,         150*ms, 1),  -- #12 Sync Point for ET Task 1 + Empty
    TT_Slot (Regular,       50*ms, 4),  -- #13 Seq. 2, F part
    TT_Slot (Empty,        100*ms   ),  -- #14
    TT_Slot (Regular,       50*ms, 2),  -- #15 Seq. 1, F part
    TT_Slot (Empty,         80*ms   ),  -- #16
    TT_Slot (Regular,       50*ms, 5),  -- #17 I part of end of plan
    TT_Slot (Empty,         70*ms   ),  -- #18
    TT_Slot (Optional,      70*ms, 6),  -- #19 F part of synced ET Task 1
    TT_Slot (Regular,       50*ms, 5),  -- #20 F part of end of plan
    TT_Slot (Mode_Change,   80*ms) );   -- #21
```

The other available helping library is TT_Patterns (see Listing 4), which offers types for representing task states and implements the patterns described in Section 5.

Listing 4. Fragment of specification of package TT_Patterns.

```
with Ada.Real_Time;
with XAda.Dispatching.TTS;

generic
   with package TTS is new XAda.Dispatching.TTS(<>);
package TT_Patterns is

   type Task_State is abstract tagged record
      Release_Time: Ada.Real_Time.Time := Ada.Real_Time.Time_First;
      Work_Id : TTS.TT_Work_Id;
      Sync_Id : TTS.TT_Sync_Id;
   end record;

   -- Simple Task State. Initialize + Main_Code
   type Simple_Task_State is abstract new Task_State with null record;
   procedure Initialize (S : in out Simple_Task_State) is abstract;
   procedure Main_Code (S : in out Simple_Task_State) is abstract;
   type Any_Simple_Task_State is access all Simple_Task_State'Class;

   -- Similar for Initial_Final Task State. Initialize + Initial_Code + Final_Code ...
   type Any_Initial_Final_Task_State is access all Initial_Final_Task_State 'Class;

   -- Initial_Mandatory_Final Task State. Initialize + Initial_Code + Mandatory_Code + Final_Code ...
   type Any_Initial_Mandatory_Final_Task_State is access all Initial_Mandatory_Final_Task_State 'Class;

   -- Initial_OptionalFinal Task State. Initialize + (S) Initial_Code + [Condition] Final_Code
   type Initial_OptionalFinal_Task_State is abstract new Task_State with null record;
   procedure Initialize (S : in out Initial_OptionalFinal_Task_State ) is abstract;
   procedure Initial_Code (S : in out Initial_OptionalFinal_Task_State ) is abstract;
   function Final_Is_Required (S : in out Initial_OptionalFinal_Task_State ) return Boolean is abstract;
   procedure Final_Code (S : in out Initial_OptionalFinal_Task_State ) is abstract;
   type Any_Initial_OptionalFinal_Task_State is access all Initial_OptionalFinal_Task_State 'Class;

   -- TT PATTERNS --
   task type Simple_TT_Task (Work_Id     : TTS.TT_Work_Id;
                             Task_State  : Any_Simple_Task_State;
                             Synced_Init : Boolean);

   task type Simple_Synced_ET_Task (Sync_Id     : TTS.TT_Sync_Id;
                                    Task_State  : Any_Initial_Final_Task_State ;
                                    Synced_Init : Boolean);


   task type Initial_Final_TT_Task (Work_Id     : TTS.TT_Work_Id;
                                    Task_State  : Any_Initial_Final_Task_State ;
                                    Synced_Init : Boolean);

   task type Initial_Mandatory_Final_TT_Task (Work_Id     : TTS.TT_Work_Id;
                                              Task_State  : Any_Initial_Mandatory_Final_Task_State ;
                                              Synced_Init : Boolean);;

   task type InitialMandatorySliced_Final_TT_Task (Work_Id     : TTS.TT_Work_Id;
                                                   Task_State  : Any_Initial_Mandatory_Final_Task_State ;
                                                   Synced_Init : Boolean);;

   task type Initial_OptionalFinal_TT_Task ( Initial_Work_Id  : TTS.TT_Work_Id;
                                             Optional_Work_Id : TTS.TT_Work_Id;
                                             Task_State       : Any_Initial_OptionalFinal_Task_State ;
                                             Synced_Init      : Boolean);

   task type SyncedInitial_OptionalFinal_ET_Task (Sync_Id     : TTS.TT_Sync_Id;
                                                  Work_Id     : TTS.TT_Work_Id;
                                                  Task_State  : Any_Initial_OptionalFinal_Task_State ;
                                                  Synced_Init : Boolean);
end TT_Patterns;
```

Package TT_Utils accepts a TT scheduler as generic parameter, hence it is also bound to that particular instance of XAda.Dispatching.TTS. For an application to use TT_Utilities and TT_Patterns, it must create instances of both using the same instance of a TT scheduler as generic parameter. For example, an application using the plan depicted in Figure 14 at the second highest priority would start as shown in Listing 5.

Listing 5. An instantiation of the TT scheduler and helper libraries.

```
with XAda.Dispatching.TTS;
with  TT_Utilities ;
with TT_Patterns;
 ...
package body TTS_Example_Application is

   Number_Of_Work_Ids : constant := 6;
   Number_Of_Sync_Ids : constant := 2;

   -- Instantiation of scheduler
   package TTS is new XAda.Dispatching.TTS
     (Number_Of_Work_Ids, Number_Of_Sync_Ids, Priority'Last - 1);

   -- Instantiation of helper libraries
   package TT_Util is new TT_Utilities (TTS); use TT_Util;
   package TT_Patt is new TT_Patterns (TTS); use TT_Patt;
    ...
end TTS_Example_Application;
```

Back to the patterns library, there are two main parts in Listing 4. The first part declares *task state types*, whereas the second declares the *task types* behind the supported patterns. A task state is associated to each application task that instantiates one of the patterns. The Task_State contains minimal information about the task: its latest release time (useful to measure release jitter) and the task's Work_Id and/or Sync_Id. The abstract Task_State type contains these fields in this particular version because we found them very useful to measure release jitters and to log the activity of tasks at runtime. This abstract root type can be extended with extra fields, such as application-dependent local task data.

The Simple_Task_State is usable by the Simple_TT_Task pattern. Procedure Initialize and Main_Code must be implemented by the user. The former is initialisation code for the task, if required. If the task does have an initialisation, then one can choose to execute the initialisation code immediately after task creation, or use an isolated slot just for the initialisation. This is chosen with the boolean parameter Synced_Init. So the Simple_TT_Task pattern has three discriminants: the used Work_Id, the Task_State, and this boolean for the initialisation phase.

In addition to Simple_TT_Task state, there are other three types declared: Initial_Final_Task_State, Initial_Mandatory_Final_Task_State and Initial_OptionalFinal_Task_State. These are to represent task states of patterns using more and more parts, hence more and more subprograms for their different phases. These types and their related subprograms are abstract, and must be implemented by the library user with application-specific code. Once the task state is defined, we need to select a task pattern for the the task and instantiate from its corresponding type.

As an example, we consider the first IMs-F task of the plan in Figure 14, with Work_Id = 2. As a task with three parts of code, the appropriate task state type to derive from is Initial_Mandatory_Final_Task_State. If we wanted to use any variables local to the task, which are obviously absent in the patterns, then we should extend the abstract task state with as many local variables as needed. Suppose we want this IMs-F task to use a local variable Counter. Let us also assume that the task does not require execution of a dedicated slot for executing its initialisation code. In this context, the task state would be implemented as shown in Listing 6. The listing shows the instantiation of the task state along with instantiation of the IMs-F pattern as well. It also shows the implementation of the application-specific code, just to illustrate the use local and global variables.

Listing 6. Creation of a TT task for the first IMs-F sequence of the example system.

```
-- Type declaration as an extension of the abstract type with local data
type First_IMF_Task is new Initial_Mandatory_Final_Task_State
  with
   record
      Counter : Natural := 0;
   end record;
-- No initialisation needed, hence null subprogram
procedure Initialize (S : in out First_IMF_Task) is null;
-- Specs of code subprograms
procedure Initial_Code (S : in out First_IMF_Task);
procedure Mandatory_Code (S : in out First_IMF_Task);
procedure Final_Code (S : in out First_IMF_Task);

-- Instantiation of task state
Wk2_State : aliased First_IMF_Task;

-- Instantiation of task pattern
Wk2 : InitialMandatorySliced_Final_TT_Task
  (Work_Id      => 2,
   Task_State   => Wk2_State'Access,
   Synced_Init  => False);

-- A global variable
Var_1 : Natural := 0;

-- Subprograms for Initial, Mandatory and Final parts
procedure Initial_Code (S : in out First_IMF_Task) is
begin
   S.Counter := Var_1;
end Initial_Code;

procedure Mandatory_Code (S : in out First_IMF_Task) is
begin
   while S.Counter < 200_000 loop
      S.Counter := S.Counter + 1;
   end loop;
end Mandatory_Code;

procedure Final_Code (S : in out First_IMF_Task) is
begin
   Var_1 := S.Counter;
end Final_Code;
```

The body of package TT_Patterns contains the structural code of all supported patterns, leaving the application-specific details of each task in the code they provide in their task state. Listing 7 shows the implementation of the IMs-F pattern used by our example task Wk_2. The code for this task type first takes note of the Work_Id used by the task in its Task_State. If the task was created with attribute Synced_Init = True, then it would wait for its separate initialisation slot. Otherwise Initialize is called immediately and then the task loop starts. Calls to Wait_For_Activation and Continue_Sliced are placed around calls to the user-provided code in the Task_State passed to the task at task creation time.

Listing 7. Implementation of the IMs-F pattern, in the body of package TT_Patterns.

```
task body InitialMandatorySliced_Final_TT_Task is
begin
   Task_State.Work_Id := Work_Id;
   if Synced_Init then
      TTS.Wait_For_Activation (Work_Id, Task_State.Release_Time);
   end if;
   Task_State. Initialize ;

   loop
      TTS.Wait_For_Activation (Work_Id, Task_State.Release_Time);
      Task_State. Initial_Code ;

      TTS.Continue_Sliced;
      Task_State.Mandatory_Code;

      TTS.Wait_For_Activation (Work_Id, Task_State.Release_Time);
      Task_State.Final_Code;
   end loop;
end InitialMandatorySliced_Final_TT_Task ;
```

7 Experimental Results

We have conducted tests to measure the scheduler overhead in our implementation of the TT model. The scheduler overhead causes deviation from the plan, since it delays the effective start of planned tasks. In other words, overhead contributes to release jitter. We are proposing to use a TT scheduler for jitter-sensitive tasks, so we need to show that the scheduler overhead is small.

We have built a plan[2] that includes transitions from six different kinds of slots to regular, optional and sync slots. Regular and optional are the kinds of slots that can host the start of a TT task, hence it is in these slots where release jitter occurs and needs be measured. Sync slots are also included in the experiment to measure the overhead imposed on synchronising an ET task with the TT plan. Since we are specifically observing release jitter, the tasks used in this test do nothing but measure and print their release jitter. The only exception is a sliced task that consumes CPU so that we can measure the cost of holding a sliced task. The fact that Wait_For_Activation and Wait_For_Sync return the start time of the slot in the plan is most helpful to measure jitter by simply reading the clock and subtracting the planned release time – all times are slightly inflated by the time taken to read the clock, which is about half a microsecond in our platform.

We have used the GNAT CE 2019 compiler with the *arm-eabi* toolchain for the ARM architecture. From the available runtimes, we have chosen the so called *ravenscar-full-stm32f4* runtime, the less restrictive one. We have modified the default runtime tick period of 1 ms to improve the runtime response to timing events. This is crucial, since the TT scheduler is driven by timing events. We have tried two different periods, 5 μs and 10 μs to observe the effect of this particular system parameter on the results.

Table 1 gives the maximum jitter values we have collected in the two setups. Cells in the table show pairs of times $(a\ /\ b)$ in microseconds. The first value corresponds to measurements with the 5 μs tick period, the second with 10 μs. Raw measurements are very stable in both setups, with null standard deviation for almost all series measured. Depending on clock configuration, the maximum measured jitter was 23.20 μs or 36.70 μs, for the 5 μs or 10 μs configuration, respectively.

[2] The code for this experiment is available as program Main_C in [9].

	Starting slot		
Ending slot	Regular	Optional	Sync
Empty	19.80 / 35.18	19.80 / 35.18	**23.20** / 33.67
Mode Change	19.74 / 25.13	19.74 / 25.13	23.14 / 23.61
Regular	20.46 / 35.85	20.55 / 35.94	19.04 / 34.42
Sync	19.88 / 35.26	14.88 / 35.26	18.19 / 33.66
Held continuation	21.58 / 36.61	21.67 / **36.70**	19.80 / 35.19
Optional	20.55 / 35.94	20.46 / 35.85	19.04 / 34.42

Table 1. Maximum jitter in microseconds when switching from an ending slot to a starting work or sync slot. Measurements obtained on an STM32F4 Discovery board with clock interrupt period configured as 5 μs / 10 μs in the *ravenscar-full-stm32f4* runtime.

As can be seen from Table 1, the tick period does have an effect on the results: all jitters are larger in the 10 μs configuration. Note that a shorter tick period involves more frequent clock interrupts, which means higher overall runtime overhead, hence a compromise is required for each system between overall runtime overhead and responsiveness to timing events. The periodic clock interrupt has a nastier effect of hiding the actual time cost of switching slot. For example, one would expect that holding a sliced task would translate to higher jitter for the next slot, but this is not what Table 1 shows for the 5 μs configuration.

To grasp an idea of what the overheads would be in an *ideal* system, with no periodic clock interrupts, we have repeated the experiments using the original GNAT runtime with 1 ms clock interrupt period, well beyond the measured jitters. The raw results are all affected by one extra clock cycle, i.e., all are of the form 1.XXX ms. Then we have subtracted exactly 1 ms from all raw values and obtained the results shown in Table 2, which make more sense. For example, now the maximum jitter occurs when the ending slot is a continuation slot with sliced work in progress, which is consistent with the need to hold the task. The shortest ideal jitters occur when the starting slot is not a TT work slot, which takes shorter for the scheduler to process since there is no possible overrun to be checked.

	Starting slot		
Ending slot	Regular	Optional	Sync
Empty	14.60	14.60	13.09
Mode Change	14.55	14.55	13.04
Regular	15.20	15.29	13.77
Sync	14.61	14.61	13.01
Held continuation	15.95	**16.04**	14.53
Optional	15.29	15.20	13.77

Table 2. *Ideal* jitters in microseconds after eliminating the effect of clock granularity.

To further minimise release jitter, the TT scheduler may easily apply an optimisation that consists in anticipating the start time of slots by a fixed amount, slightly less than the minimum measured jitter. In this manner, the scheduler overhead is not paid for at the start, but at the end of the slot. Overhead is unavoidable, but moving its effect to the end of the slot improves *release* jitter.

Our conclusion from these experimental jitter measurements is positive both for their low values and their repeatability.

8 Conclusions

The architecture presented in this paper, and its corresponding Ravenscar implementation, make it possible to run a mixed workload of time- and event-triggered tasks on top of an efficient priority-based scheduler. At the application level, the workload is subdivided in two sets, one formed by the TT tasks, the other by the ET tasks. In this manner, the application can be designed to take advantage of one or the other scheduling approach. The TT set contains only tasks that are specially sensitive to release jitter, such as control or communication tasks, so that they are predictably scheduled by an offline plan. Excluding other tasks from the plan eases the offline plan design. The rest of tasks are left to the priority-based scheduler, which will make all scheduling decisions online, based on task priorities. The relative priority of the TT plan can be established, leaving room for ET application tasks that require prompt service (e.g., sporadic tasks) or tasks that are for whatever reason difficult to integrate in an offline schedule. Other authors have proposed a combined ET-TT architecture, but they focused on prompt service to sporadic tasks at the cost of shifting the release of planned TT tasks. For the purpose of reducing release jitter of TT tasks, it is best to give precedence to the TT plan instead of delaying the start of jitter-sensitive tasks to favour others.

The proposed TT model is flexible, defining different slot types of arbitrary duration. It provides the concept of slicing, which further facilitates the design of plans with long TT tasks. Due to the semantics defined for the Hold operation and its relation with ongoing protected actions, sliced tasks may share resources with other tasks – provided they use some protection mechanism, such as protected objects. The concept of padding time for continuation slots helps keeping the potential blocking time confined to the originating sliced task, and avoids delaying the release of a subsequent task, which could be a jitter-sensitive TT task. The concept of synchronisation points, in the form of sync slots, allows ET tasks to coordinate their execution with the plan and execute at their own priority without taking work slots in the plan. Optional slots introduce further flexibility in the model, allowing the application to take or skip particular plan slots.

The results obtained from jitter measurements in our Ravenscar implementation of the TT scheduler confirm the applicability of the model with promising results. We have proposed an illustrative set of TT, ET and combined task patterns that take advantage of the model for typical control and communication tasks.

Besides an implementation of the TT scheduler itself and some extensions and modifications to the runtime systems, we have developed libraries that provide convenient types and operations to facilitate the definition of TT plans and to use the proposed patterns. This is a first step towards increasing the usability of the model. Other extensions and tools are being considered for further increasing its applicability, such as automatic generation of code and skeletons, plan builders, etc.

References

1. Kopetz, H.: Event-Triggered versus Time-Triggered Real-Time Systems. In Springer-Verlag, ed.: Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, London, UK (1991) 87–101
2. Albert, A.: Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. In: Embedded World Conference, Nürnberg (February 2004) 235–252
3. Isovic, D., Fohler, G.: Handling Mixed Sets of Tasks in Combined Offline and Online Scheduled Real-Time Systems. Real-Time Systems **43**(3) (November 2009) 296–325
4. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in high-integrity systems. Technical Report YCS-2017-348, University of York (June 2017)
5. ISO/IEC-JTC1-SC22-WG9: Ada Reference Manual ISO/IEC 8652:2012(E), http://www.ada-europe.org/manuals/LRM-2012.pdf (2012)
6. Real, J., Sáez, S., Crespo, A.: Combining Time-Triggered Plans with Priority Scheduled Task Sets. In Bertogna, M., Pinho, L.M., Quiñones, E., eds.: Reliable Software Technologies – Ada-Europe 2016. Volume 9695 of Lecture Notes in Computer Science., Cham, Springer International Publishing (2016)
7. Real, J., Sáez, S., Crespo, A.: Combined Scheduling of Time-Triggered and Priority-Based Task Sets in Ravenscar. In Casimiro, A., Ferreira, P., eds.: Reliable Software Technologies – Ada-Europe 2018. Volume 10873 of Lecture Notes in Computer Science., Cham, Springer International Publishing (June 2018)
8. Palencia, J.C., González-Harbour, M., Gutiérrez, J.J., Rivas, J.M.: Response-Time Analysis in Hierarchically-Scheduled Time-Partitioned Distributed Systems. IEEE Transactions on Parallel and Distributed Systems **28**(7) (July 2017) 2017–2030
9. Real, J., Sáez, S.: Time-Triggered Scheduling in Ravenscar (source code, version 0.3.0). DOI: 10.5281/zenodo.3490505 (July 2019)
10. Baker, T.P., Shaw, A.: The cyclic executive model and Ada. In: Proceedings IEEE Real Time Systems Symposium 1988, Huntsville, Alabama. (1988) 120–129
11. Real, J., Crespo, A.: Offsets for scheduling mode changes. In: Proceedings of the 13th Euromicro Conference on Real-Time Systems. Delft, The Netherlands, IEEE Computer Society Press (2001) 3–10
12. Real, J., Crespo, A.: Mode Change Protocols for Real-Time Systems: A Survey and a new Proposal. Real-Time Systems **26**(2) (March 2004) 161–197