A specification framework for mixed-criticality scheduling protocols

ALAN BURNS, University of York, UK
CLIFF B JONES, Newcastle University, UK

This paper presents a general formal framework for describing the relationship between a criticality-aware scheduler, a set of application jobs that are assigned different criticality levels, and an environment that generates both work and faults that the run-time system must control. The proposed formalism extends the rely-guarantee approach, which facilitates formal reasoning about the functional behaviour of concurrent systems, to address *real-time* properties. The exposition of the general framework is supplemented by a seven step approach that enables it to be instantiated to deliver the formal specification of any proposed mixed-criticality scheduling protocol. The expressive power of the approach is explored via a non-trivial instantiation.

ACM Reference Format:
Alan Burns and Cliff B Jones. 2025. A specification framework for mixed-criticality scheduling protocols. *ACM Trans. Embedd. Comput. Syst.* 1, 1 (August 2025), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

There is extensive literature on scheduling approaches and more recent proposals have tackled tolerance to some degree of excess run-time demand beyond the normal load. To enhance the robustness of schedulers, in 2007 Vestal proposed [46] that jobs should be distinguished by allocating them different 'criticality' levels. If a fault such as overly frequent job triggering occurs during execution, computation effort can then be allocated to the most critical jobs at the expense of those of lesser criticality. Since 2007, over 1000 papers on the theme of Mixed Criticality Systems (MCS) have been published with a wide range of scheduling protocols being proposed [11, 12]. It should be noted that not all standards and papers on MCS assign the same meaning to 'criticality'; this is an issue explored by Graydon and Bate [24], Esper et al. [23], Paulitsch et al. [44], Ernst and Di Natale [22], Wilhelm [47], Jiang [29–31], Lee and Kim [39] and Reghenzani and Fornaciari [45]. In this paper we do not rely on any particular notion of criticality, rather we facilitate consistent and coherent usage however the term is defined.

Two previous papers developed formal specifications for particular scheduling approaches, [36] addresses 'Earliest Deadline First' scheduling with execution time overruns and [16] tackles 'Fixed Priority' scheduling with early arrival faults. The aim of developing these formal specifications was threefold:

(1) It provides an unambiguous definition of the scheduling protocol and the assumptions implied on the hardware platform and the environment in which the system will execute.
(2) It allows a schedulability test to be developed – this enables the task set of the application to be checked for compliance with the requirements of the scheduling approach.
(3) It gives rise to a specification that provides a basis for the development of actual code of a run-time scheduler.

Authors' Contact Information: Alan Burns, University of York, Department of Computer Science, York, UK, alan.burns@york.ac.uk; Cliff B Jones, Newcastle University, School of Computing, UK, cliff.jones@ncl.ac.uk.

Manuscript submitted to ACM

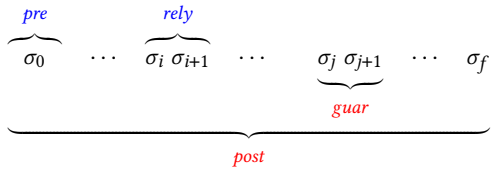

Fig. 1. Relating pre/rely/guarantee/post conditions over a sequence of states σ_i

These earlier papers illustrate that considerable effort is needed to develop a bespoke specification. Hence, in this paper, §2 and 3 offer a general framework for mixed-criticality scheduling approaches that can then form the starting point for the specification of any relevant scheduling discipline; §4 demonstrates how this framework can be instantiated to a specification for a combination of the two previously published schedulers and §5 addresses schedulability tests.

The remainder of this introductory section outlines background material.

1.1 Background material

Any formal description has to employ notation; that which is used in this paper should present few difficulties to any reader familiar with set notation and logic. To make this paper self-contained, notes are provided in Appendix A. Most of the notation concerns objects used in the formal models. Of particular importance are 'states' of the formal description: rather than build a definition in terms of functions, many modelling techniques describe systems as state transitions. Here, states are changed by 'operations' which are specified rather than given as programs. Pre/post conditions suffice for the specification of sequential operations because the relation of the starting and finishing states defines what users need to know.

Interference is a key issue with concurrency: with shared variable concurrency, the values of variables can be changed whilst a specified piece of code is executing. The intention behind rely/guarantee conditions [32–34] was to record information about interference and offer rules that supported top-down decomposition of specified components into concurrent sub-components. (See [26] for a more tutorial description and [20] for a thorough discussion of compositional development of concurrent programs.) Fig. 1 relates the components of an extended specification:

- the pre condition defines the set of initial states σ_0; this can be assumed by the developer;
- a developer must accept that the state can change during execution but such changes (e.g. σ_i/σ_{i+1}) can be assumed to be bounded by the rely condition;
- execution of the created code must satisfy the post condition which is a relation between the initial state σ_0 and final state σ_f;
- any state changes during execution of the created code must also be bounded; for example, the transition by the code of the specified component from σ_j to σ_{j+1} must satisfy the guarantee condition.

Common examples or interference assertions involve orderings such as an operation relying on the value of a variable monotonically decreasing ($x' \leq x$ or $s' \subseteq s$); trivial examples are equalities but rely or guarantee conditions that include a boolean flag are more useful (e.g. $flag \implies y' = y$). The use of rely-guarantee conditions in the scheduling application is more interesting: the overall objective is that jobs should complete by their deadlines; in order to achieve this the scheduler must be designed on the assumption that jobs do not exceed their load estimates whereas jobs work under the assumption that they are allocated resources in a timely manner.

In, for example [15], rely-guarantee thinking was shown to offer a way to describe (and/or develop) fault-tolerant systems. Of more relevance to the material below is the research that has shown rely conditions being used to record assumptions about components that are, in fact, not under the control of the developer (see for example [15]). As has been shown in [16, 36], rely/guarantee conditions can be used to express the assumptions between a run-time scheduler and the jobs that it has to control (see matching of *rely-Scheduler*/*guar-Job* in §3.3 below). Detailed proof rules for rely/guarantee specifications are not required in the current paper; only the matching of the corresponding clauses is needed.

The role of predicate restrictions on objects is described in Appendix A; one important application of this idea is the concept of data-type invariants on state-like objects (e.g. *inv-State* in §3.1). Because all instances of such states are required to satisfy their invariant, they play the role of general rely-guarantee conditions. The contribution of this to progress arguments is covered in §3.6.

One characteristic of Cyber-Physical Systems (CPS) is that their behaviour is best understood (and specified) at different time scales (from microseconds or less to hours or more). Time is clearly a crucial notion in the specification of real-time systems, but it is usually represented, in modelling schemes for example, on a single indexed axis. Such an approach fails to separate the structural properties of the system, forces different temporal notions onto the same flat description, and fails to support the separation of concerns that the different time scales of the system exhibit. Just as the functional properties of a system can be modelled at different levels of abstraction or detail, so too can its temporal properties be representable in different, but provably related, time scales.

To make clearer use of 'time', with the aim of producing more dependable computer-based systems, Burns and Hayes [14] proposed a framework that explicitly identifies a number of distinct 'time bands' in which a system under study is situated. Within this framework, each band is defined by a granularity G (e.g. a second) and a precision ρ (e.g. 10ms); and actions within the band either have duration (e.g. an integer number of seconds) or are considered to be instantaneous events. Events, although deemed to be instantaneous at one band, are projected onto finer bands where they have duration — with this duration being constrained to be no greater than the precision of the original band. This notion of an instantaneous event is used in §3.3 below to represent a context switch that moves a newly released job to the set of active jobs: from the point of view of the application, this takes no time, although, of course, at a time band below that of the application some bounded duration (denoted as τ) is allowed.

Precision is also used to define equality: two events, $e1$ at time $t1$ and $e2$ at time $t2$ occur at the same time ($t1 =_\rho t2$) if they are contained within ρ when mapped to a sufficiently detailed finer time band. This notion of equality is used in §3.2.

These two uses of precision, taken together, define a three time band model for a typical embedded system. The application may have its temporal properties (deadlines, periods etc.) defined in a millisecond band, the implementation of the Scheduler and related support functions is perhaps best placed within a microsecond band. In this band, equality between internal and external notions of time can be assumed; but in a nanosecond band their differences can be expressed and potentially measured. The precision in the top band (as used in §3.3) is perhaps 10 μs; the precision of the middle band (as used to define temporal equality in §3.2) is perhaps 10 ns.[1] The finest band does not have a precision since there is no lower band on which it is projected.

[1]Although this three band temporal structuring is assumed by the proposed framework, the actual values of the defining parameters are not fixed and will vary between hardware platforms and will certainly alter in the future.

2 Scheduling

The purpose of scheduling is to plan and control the execution of *Job*s to meet deadlines; this can only be achieved with assumptions about the load that jobs can impose. Almost inevitably, this gives rise to a static 'planning' phase in which a scheduling 'discipline' is selected followed by use of a run-time system that implements the chosen discipline. The current paper distinguishes the off-line planning process and how it feeds into the requirements on the run-time scheduler whose detailed specification provides a precise reference point for developing implementations.

Because multiple *Job*s can overlap, each requiring resources at run-time, it is necessary to have a run-time *Scheduler* that selects which *Job* is to run. These choices are governed by the planning discipline from the static planning phase.

The overall combination of *Planning*/*Scheduler* can be viewed as:

$$Time \parallel \{Framing\ ;\ Confirming\ ;\ \{Scheduler \parallel Job_1 \parallel Job_2 \parallel \cdots \parallel Job_k\}\}$$

During the *Framing* part of *Planning* the hardware platform will be chosen, the load parameters determined and the scheduling discipline identified. As part of this process, a schedulability test will be obtained (either from first principles or from existing practice). Ideally this test will be formally derived from the scheduling approach (see §3.8). During *Confirming*, the schedulability test is applied to the actual load data of the application. If the test fails, *Confirming* is aborted; if it passes, then at some future time the system is cleared to begin execution. Fault-Tolerant behaviour must also be checked for feasibility (e.g. relaxed worst-case execution times can be accommodated) see §2.2.

Mixed-criticality makes it possible to assign different levels of criticality to the jobs: in the best case, all jobs meet their deadlines but levels of fault-tolerant behaviour can allocate extra resources to jobs of higher *Criticality* so that they meet (possibly more generous) deadlines at the expense of jobs that are less critical.

It was noted in the Introduction that there has been an extensive collection of scheduling papers published since the notion of Mixed-Criticality was introduced by Vestal in 2007. While many interesting scheduling ideas are contained in these papers, in general, they often lack the detail that is needed if these schemes are to be implemented and then employed in critical embedded systems. For example, the assumptions made about how the environment can behave are often missing or at best provided in an incomplete manner. In the following a formal framework is provided that allows a complete and unambiguous description of the run-time behaviour of the scheduler, the actions of the application's jobs and the system's state to be defined.

The following framework specification is an extended and generalised version of the two specific specifications previously published [16, 36]. It has been derived from a detailed examination of the Mixed-Criticality literature to ensure that it has the necessary expressive power and ease of use to be generally applicable. The specification addresses the following entities: *Tasks*, *Jobs*, *Modes*, rely and guarantee conditions, Invariants, *State*, *Time*, *Clocks*, and the *Scheduler* with its *Release*, *ModeUp* and *Overrun* methods.

2.1 Tasks define types of jobs

It is standard to employ a notion of 'task' to define type information of 'jobs'; each task gives rise to a sequence of jobs: any particular application has a defined set of *TaskId*s and *TaskInfo* objects provide static information about jobs relating to their task. The sequence of jobs is either periodic (sometimes called regular or 'time-triggered') or sporadic (also called aperiodic or 'event-triggered'). In addition each task has a k field that defines the criticality of the task (*Criticality* is a partially ordered set of values). *TaskInfo* also contains information about the load its jobs impose on the run-time system. This leads, (using the record and mapping notation described in Appendix A), to the following object descriptions (which are completed in §2.2):

$$TaskMap = TaskId \xrightarrow{m} TaskInfo$$

$$\begin{array}{lll} TaskInfo :: & periodic & : \mathbb{B} \\ & k & : Criticality \\ & loadm & : \cdots Load \cdots \end{array}$$

The 'load' that the jobs of a task can impose on the runtime system are captured in objects of type *Load*. These records contain a *D* field that is a relative deadline: these deadlines are expressed as *Duration*s; a job is required to terminate before that duration from when it is released. Two other aspects of *Load* are of the same type: an estimate of worst-case execution time (WCET) is in the *C* field and the expected arrival gap between successive jobs from the same task is in the *T* field.

Fault tolerance and mixed-criticality give rise to a notion of firmness of deadlines. Published schemes often fail to clearly distinguish between jobs that only have soft deadlines, jobs that should meet their deadlines (it is a fault if they do not) and those that are provided with extra resources to ensure that they indeed do so (unless the overload condition become too severe):

$$Firmness = \{\textsc{Soft}, \textsc{Brittle}, \textsc{Hard}\}$$

The three values of the *Firmness* parameter are as follows: Soft – not forced to meet the deadline; Brittle – must meet the deadline if no job overruns its Worst-Case Execution Time (WCET denoted by the symbol *C*) or is released for execution before the time determined by *T*; and Hard – must meet the deadline even when it overruns *C* or is released too early. The *D* parameters define the temporal requirement which must be satisfied if the deadline is defined to be Brittle or Hard. The *R* value represents the longest response time that any job of that task can experience, it is computed as part of the *Confirming* process in many forms of schedulability analysis — see §5.

$$\begin{array}{lll} Load :: & D & : Duration \\ & fness & : Firmness \\ & C & : Duration \\ & T & : Duration \\ & R & : Duration \end{array}$$

2.2 Fault-tolerance: modes

If a job exceeds its estimated worst-case execution time (*C*), or if an event-triggered job is released before its expected release gap (*T*), it is considered to be a fault and a fault-tolerant scheduler must be able to avoid critical jobs with Hard deadlines being starved of resource. Both of these forms of fault arise from invalid assumptions made during Planning concerning resource usage. They can, for example, be caused by failures in analysis (e.g. WCET estimates being in error – too low), or as a consequence of other forms of error and recovery such as the need to re-execute a job after a failed validity check or the execution of an exception handler.

Handling fault-tolerance requires a notion of system *Mode*. Each mode defines a level of performance and jobs of tasks with a higher criticality rating are protected, while tasks of a lower criticality have their level of service reduced (perhaps to zero). The deadline of a job may be Hard in one mode, Brittle in another and Soft in a third. To capture these properties the *Load* parameters are made mode specific, leading to the following completed definition of *TaskInfo*:

$$\begin{array}{lll} \mathit{TaskInfo} :: & \mathit{periodic} & : \mathbb{B} \\ & k & : \mathit{Criticality} \\ & \mathit{loadm} & : \mathit{Mode} \xrightarrow{m} \mathit{Load} \end{array}$$

The set of modes always includes a normal mode (Norm $\in$ *Mode*) in which the system is initialised. In this Norm mode there are no Soft jobs; in response to a fault, the system may transition to a mode in which jobs of a certain *Criticality* become Soft, but initially all jobs have deadlines that should be satisfied.[2]

A specific application of the framework (such as that illustrated in §4) will have one or more operational modes to represent different levels of degraded performance. A *fault model* for the system will define those faults that must be tolerated in some way. If the behaviour of the system moves beyond that specified by the fault model, then all that can be achieved at run-time is for the system to switch to a fail-stop strategy where even the most critical tasks are prevented from releasing new jobs. It is a criticism of many 'mixed-criticality' papers that they fail to make it clear what level of faults can be tolerated and what the behaviour of the system should be if this level is exceeded.

In our proposed framework any mode that does not have a further degraded mode to which it can transition is called a *terminal* mode. In the MC literature there is usually only one implied terminal mode and it is often referred to as '*HI*-crit' as it only protects the behaviour of the highest criticality tasks. As the general model can have more than one terminal mode we do not fix a specific name, rather we use a predicate:

$$\mathit{terminal}: \mathit{Mode} \to \mathbb{B}$$

to identify the terminal modes. As there is no further degradation from a 'terminal' mode all deadlines within that mode must be either Soft or Brittle (i.e. not Hard). Despite offering different fault-tolerance modes, a job stream that imposes more than the specified load will not necessarily achieve the stated deadlines; which therefore cannot be Hard.

The two properties outlined above lead to the definition of a type invariant for *TaskInfo*:

$$\mathit{inv\text{-}TaskInfo} : \mathit{TaskInfo} \to \mathbb{B}$$

$$\begin{array}{l} \mathit{inv\text{-}TaskInfo}(\mathit{info}) \quad \triangleq \\ \quad \mathit{info.loadm}(\text{Norm}).\mathit{fness} \neq \text{Soft} \wedge \\ \quad \forall \mathit{mode} \in \mathit{Mode} \cdot \mathit{terminal}(\mathit{mode}) \;\Rightarrow\; \mathit{info.loadm}(\mathit{mode}).\mathit{fness} \neq \text{Hard} \\ \quad \ldots \end{array}$$

As mentioned above, only objects that satisfy this predicate are considered to be instances of *TaskInfo*. In a specific instantiation, there will be further clauses of this invariant that link the load parameters from one mode to those of another; for example a highly critical job may have a larger worst-case execution time estimate in a terminal mode than in the initial Norm mode (this is the case in the instantiation given in §4).

It is important to emphasise that the framework does not impose any specific meaning to the notion of criticality or how it is used to tolerate faults. For example, in the instantiation noted above, there are two fault tolerance modes. For one form of fault, in the corresponding mode low criticality jobs run less often but still have non-soft deadlines; however, in the mode defined for more extreme faults, the lower criticality deadlines become soft. This is a choice made during Planning, not one required by the framework.

[2]The Mixed-Criticality (MC) literature often refers to the Normal mode as '*LO*-crit' as even *LO*-criticality jobs are required to meet their deadlines in this mode; hence no Soft deadlines.

A fault-tolerance strategy not only has to deal with the consequences of a fault, such as a task overrunning its expected worst-case execution time, it also has to resume normal behaviour when it is safe to do so. This is accommodated by allowing transitions from any mode (including terminal modes) back to the NORM mode. For example, it is always safe to move back to the normal mode when there are no active jobs in the system.

3 Specifying a general *Scheduler*

The body of the formal specification of the (general) run-time scheduler defines a state transition system. In §3.1 the objects of type *State* are fixed; the state transitions are specified in §3.3/3.5. Scheduling needs to connect between *Time* in the external world and its approximations stored in internal states; this link is defined in §3.2. The issue of ensuring adequate progress is covered in §3.6.

3.1 Runtime *State*

The *Scheduler* requires access to an internal clock but its values (t: *ClockValue*) can only be approximations to *Time* in the external world (this relation is defined in §3.2 below). In addition to t, the run-time *State* of the system has a number of fields. First, it has an indication of the current mode. Next, it includes the *TaskMap* — but the operations of the *Scheduler* will only have read access. Also in the *State* is a mapping from *JobId*s to their *JobInfo* for the jobs that are *active* – i.e. executable (these objects contain a link to the task type and a record of the clock value at which this job instance was released). The next two fields, *run* and *used* record which job (if any) is actually executing[3] and the currently used execution time of each job. The penultimate field (*PA*) of *State* objects holds the time of the previous activation of the jobs from the designated task; this will be used to determine if a job is being released too early. Finally, *shared* denotes the objects that will be updated by the operations of the jobs. As this is inevitably application specific, no details are given in this general framework. Taken together this leads to the following definitions of *JobInfo*, *State* and the important invariant for this state (*inv-State*):

$$
\begin{array}{lll}
\mathit{JobInfo} :: & \mathit{type} & : \mathit{TaskId} \\
& \mathit{release} & : \mathit{ClockValue}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{State} :: & t & : \mathit{ClockValue} \\
& \mathit{mode} & : \mathit{Mode} \\
& \mathit{tkm} & : \mathit{TaskMap} \\
& \mathit{active} & : \mathit{JobId} \xrightarrow{m} \mathit{JobInfo} \\
& \mathit{run} & : \left[\mathit{JobId}\right] \\
& \mathit{used} & : \mathit{JobId} \xrightarrow{m} \mathit{Duration} \\
& \mathit{PA} & : \mathit{TaskId} \xrightarrow{m} \mathit{ClockValue} \\
& \mathit{shared} & : \mathit{Id} \xrightarrow{m} \mathit{Value}
\end{array}
$$

where

[3]In this framework description we assume, for ease of presentation, that the hardware platform has only a single processor; extending its use to multiprocessor and multicore platforms is straightforward. If tasks are statically allocated to processors then a processor Id can be added to *TaskInfo*. If jobs can migrate following an overrun then a processor Id can be added to *Load* so that a mode change can sanction and control these migrations. Cores can share a local clock or processors have their own local time source – both can be accommodated by the relationship between *Time* and *ClockValues* discussed in §3.2. For multiprocessor platforms, *run* would need to contain a set of *JobId* and additional changes would need to be made to capture the static, and possible dynamic, allocation of jobs to cores. Extensions to cope with more heterogeneous platforms (for example ones that allows a job to migrate between processors of different or variable speeds) will be addressed as part of further work.

$$
\begin{array}{l}
\mathit{inv\text{-}State}(st) \quad \triangleq \\
\quad \textbf{dom}\ st.\mathit{used} = \textbf{dom}\ st.\mathit{active} \wedge \\
\quad (\forall j \in \textbf{dom}\ st.\mathit{active} \cdot \\
\qquad \textbf{let}\ \mathit{mk\text{-}JobInfo}(\mathit{type}, \mathit{rel}) = st.\mathit{active}(j)\ \ \textbf{in} \\
\qquad \textbf{let}\ \mathit{mk\text{-}TaskInfo}(\mathit{per}, k, \mathit{ldm}) = st.\mathit{tkm}(\mathit{type})\ \ \textbf{in} \\
\qquad \mathit{ldm}(\mathit{mode}).\mathit{fness} = \textsc{Soft} \vee st.t \leq \mathit{rel} + \mathit{ldm}(\mathit{mode}).D) \wedge \\
\quad \mathit{select}(st.\mathit{active}, st.\mathit{tkm}, st.\mathit{mode}, st.\mathit{run})
\end{array}
$$

The first conjunct of *inv-State* simply requires that *used* time information is available for all *active* jobs. The second (quantified) conjunct requires that jobs (other than those with Soft deadlines) must not have passed their deadlines (as defined in the current mode). The *select* relation is specific to the chosen planning discipline which defines how the *run* job is chosen from among *active* jobs;[4] if there are no active jobs *run* = **nil**.

$$
\begin{array}{l}
\mathit{select} : (\mathit{JobId} \xrightarrow{m} \mathit{JobInfo}) \times \mathit{TaskMap} \times \mathit{Mode} \times \left[\mathit{JobId}\right] \to \mathbb{B} \\
\mathit{select}(\mathit{active}, \mathit{tkm}, \mathit{mode}, r) \quad \triangleq \\
\quad (\mathit{active} = \{\,\} \wedge r = \textbf{nil} \vee r \in \textbf{dom}\ \mathit{active}) \wedge \cdots
\end{array}
$$

To prevent jobs with Soft deadlines from interfering with jobs with Hard or Brittle deadlines, further rules need to be added to *select*. These conjuncts typically depend on the chosen scheduling approach.[5]

All of the parameters introduced in the last two sections inform the off-line planning process which must be completed, documented and if necessary certified, before the on-line (run-time) phase can begin. Both planning and execution are subject to timing constraints, and hence take place within the context of the external passage of time (i.e. *Time* in the external world). However, as noted in §1.1, *Time* is a granulated phenomenon; planning actions and job executions must both satisfy deadlines — but they are not on the same scale and should be modelled within different time bands.

3.2 Specification grounded in *Time*

As pointed out above, internal *ClockValue*s can only approximate *Time* in the external world; Σ captures the dynamic properties of the system as it progresses through (external) time.

$$\Sigma = \mathit{Time} \to \mathit{State}$$

where

$$\mathit{inv\text{-}}\Sigma : \Sigma \to \mathbb{B}$$

$$\mathit{inv\text{-}}\Sigma(\sigma) \quad \triangleq \quad \mathcal{T}(\sigma) \wedge \mathcal{E}(\sigma)$$

[4] Which *JobId* is in *run* depends on the chosen scheduling discipline but there are some consistency conditions such as the property that any subset of *active* including *run* would choose the same job to run.

$$\mathit{act}' \subseteq \mathit{act} \wedge \mathit{select}(\mathit{act}, \mathit{tm}, \mathit{mode}, \mathit{run}) \ \Rightarrow\ \mathit{select}(\mathit{act}', \mathit{tm}, \mathit{mode}, \mathit{run})$$

[5] The definition of the predicaste *select* is therefore completed in §4.

Notice that values of Σ are mathematical functions and, unlike maps, not required to be finite.

The *t*: *ClockValue* field, in any *State*, is related to *Time* using the time bands [14] notion of precision $=_\rho$ (see discussion in §1.1).

$$\begin{array}{l}\mathcal{T}(\sigma) \quad \triangleq \\ \quad (\forall \alpha \in \mathit{Time} \cdot \sigma(\alpha).t =_\rho \alpha) \wedge \\ \quad (\forall \alpha_1, \alpha_2 \in \mathit{Time} \cdot \alpha_1 < \alpha_2 \;\Rightarrow\; \sigma(\alpha_1).t \leq \sigma(\alpha_2).t)\end{array}$$

The second conjunct of $\mathcal{T}$ ensures that the computer clock cannot go backwards; this is necessary because of the level of imprecision over the definition of equality.

When a job is *run*ning, its *used* field increases in real-time; if it is not the *run* job then *used* does not change:

$$\begin{array}{l}\mathcal{E}(\sigma) \quad \triangleq \\ \quad \forall \alpha_1, \alpha_2 \in \mathit{Time} \cdot \\ \quad\quad \forall j \in (\mathbf{dom}\, \sigma(\alpha_1).\mathit{used} \cap \mathbf{dom}\, \sigma(\alpha_2).\mathit{used}) \cdot \\ \quad\quad\quad ((\forall \alpha \mid \alpha_1 \leq \alpha \leq \alpha_2 \cdot \sigma(\alpha).\mathit{run} = j) \;\Rightarrow\; \sigma(\alpha_2).\mathit{used}(j) - \sigma(\alpha_1).\mathit{used}(j) =_\rho \alpha_2 - \alpha_1) \wedge \\ \quad\quad\quad ((\forall \alpha \mid \alpha_1 \leq \alpha \leq \alpha_2 \cdot \sigma(\alpha).\mathit{run} \neq j) \;\Rightarrow\; \sigma(\alpha_2).\mathit{used}(j) = \sigma(\alpha_1).\mathit{used}(j))\end{array}$$

3.3 *Scheduler* class and methods

A class description of *Scheduler* factors out rely and guarantee conditions that apply to all of its methods (*Release*, *Overrun*, *ModeUp*), it also defines the pre condition for the run-time behaviour of the system which includes setting all *PA* fields far enough back so that initial *Release*s can occur.

The *Scheduler* has access to fields of the *State* with **rd**/**wr** access noted (e.g. it only has read access to *t* whose progress is dictated by $\mathcal{T}$). In order to store the actual application data *a-tkm*, the *Scheduler* class has write access to *tkm* but each of its methods only has read access to *tkm*. The *Scheduler* can only update the membership of *used* when a *Start* method creates a new *JobId* (and sets its *used* entry to zero); *guar-Scheduler* requires that existing *used* entries are not updated by the scheduler since they are governed by $\mathcal{E}$. The key WCET assumption in *rely-Scheduler* is matched by *guar-Job*.

Scheduler (*a-tkm*: *TaskMap*)

$$\begin{array}{ll}\textbf{ext rd } t & : \mathit{ClockValue} \\ \quad \textbf{wr } \mathit{mode} & : \mathit{Mode} \\ \quad \textbf{wr } \mathit{tkm} & : \mathit{TaskMap} \\ \quad \textbf{wr } \mathit{active} & : \mathit{JobId} \xrightarrow{m} \mathit{JobInfo} \\ \quad \textbf{wr } \mathit{run} & : [\mathit{JobId}] \\ \quad \textbf{wr } \mathit{used} & : \mathit{JobId} \xrightarrow{m} \mathit{Duration} \\ \quad \textbf{wr } \mathit{PA} & : \mathit{TaskId} \xrightarrow{m} \mathit{ClockValue}\end{array}$$

$$\begin{array}{l}\textbf{pre } \mathit{tkm} = \mathit{a\text{-}tkm} \wedge \\ \quad \mathit{active} = \{\,\} \wedge \\ \quad \mathit{mode} = \textsc{Norm} \wedge \\ \quad \forall \mathit{tid} \in \mathbf{dom}\, \mathit{tkm} \cdot \mathit{PA}(\mathit{tid}) \leq (t - \mathit{tkm}(\mathit{tid}).\mathit{loadm}(\textsc{Norm}).T)\end{array}$$

rely $(\forall j \in \mathbf{dom}\ used' \cdot used'(j) \leq tkm(active'(j).type).loadm(mode').C) \wedge$
$(\forall j \in (\mathbf{dom}\ active' - \mathbf{dom}\ active) \cdot t - PA(active'(j).type) \geq tkm(active'(j).type).loadm(mode').T)$
guar $(\mathbf{dom}\ used) \lhd used' = used$

Notice that there is no post condition for *Scheduler* because it is not intended that it should terminate. The constraint in *guar-Scheduler* indicates that none of its methods can affect entries that are already in *used*; *Release* can add new entries; values in $used(j)$ advance in accordance with $\mathcal{E}$; and only *Job* operations can remove entries from *used*. The rely condition requires all active jobs to conform to the mode-specific constraints on used execution time and inter-job release times.

The main method of the Scheduler (*Release*) is responsible for creating the new *JobInfo* and, where appropriate, adding it to the set of active jobs. However, if the triggering event occurs too early for the current mode, then a mode change will also be necessary if the associated task is classified as Hard in the current mode. The other possibility is that the event is too early and the firmness of the task is Brittle; in this situation the job is not released for execution and the event is ignored.[6] If the task is periodic then its jobs are, in effect, released by the scheduler at the appropriate time, i.e. never early (or late).

Release (*tid*: *TaskId*)

ext rd t : *ClockValue*
wr *mode* : *Mode*
rd *tkm* : *TaskMap*
wr *active* : $JobId \xrightarrow{m} JobInfo$
wr *run* : $\left[JobId\right]$
wr *used* : $JobId \xrightarrow{m} Duration$
wr *PA* : $TaskId \xrightarrow{m} ClockValue$

pre $tkm(tid).periodic \Rightarrow t =_{\rho} PA(tid) + tkm(tid).loadm(mode).T$
rely $t \leq t'$
guar $(\mathbf{dom}\ active) \lhd active' = active$
post $PA' = PA \dagger \{tid \mapsto t\} \wedge$
$\mathbf{let}\ mk\text{-}TaskInfo(per, k, ldm) = tkm(tid)\ \mathbf{in}$
$(t - PA(tid) \geq ldm(mode).T \Rightarrow$
$\mathbf{let}\ j \in (JobId - \mathbf{dom}\ active)\ \mathbf{in}$
$active' = active \cup \{j \mapsto mk\text{-}JobInfo(tid, t)\} \wedge$
$mode' = mode \wedge used'(j) = 0) \wedge$
$(t - PA(tid) < ldm(mode).T \Rightarrow$
$(ldm(mode).fness = \textsc{Hard} \Rightarrow$
$\mathbf{let}\ j \in (JobId - \mathbf{dom}\ active)\ \mathbf{in}$
$active' = active \cup \{j \mapsto mk\text{-}JobInfo(tid, t)\} \wedge$
$mode' \neq mode \wedge used'(j) = 0) \wedge$
$(ldm(mode).fness = \textsc{Brittle} \Rightarrow$
$mode' = mode \wedge active' = active))$

[6] Several specific map operators are used here and the reader is reminded that they are defined in Appendix A.

The specification also needs to dictate the maximum delay from a triggering interrupt until a *Release* operation executes. In the time band of the application, the *Release* operation should be executed *immediately*. Let τ represent the precision of this band (see discussion in §1.1). This implies that the *Release* operation must execute within τ of the trigger (as measured at an appropriate finer band). In this case, the predicate that defines the maximum delay needs an argument that is not in Σ: any specific execution experiences a sequence of *Time*s at which *Job*s of each task are triggered by external events:[7]

$Events$: $TaskId \xrightarrow{m} Time^*$

Each such sequence is, of course, strictly monotonically increasing. Notice that *Time* relates to time in the external world.

To write this predicate as an assertion (rather than as an operation) the post condition of *Release* is quoted — see Appendix A. Hence, in the following, *inv-Release-timing* requires the relation represented by *post-Release* to be true at the end of an interval of duration δ (with $\delta \leq \tau$):

$$inv\text{-}Release\text{-}timing : Events \times \Sigma \to \mathbb{B}$$

$$\begin{aligned}
&inv\text{-}Release\text{-}timing(evm, \sigma) \quad \triangleq \\
&\quad \forall tid \in \mathbf{dom}\ evm \cdot \\
&\qquad \mathbf{let}\ evs = evm(tid)\ \ \mathbf{in} \\
&\qquad \forall i \in \{1..\mathbf{len}\ evs\} \cdot \\
&\qquad\quad \exists \delta \leq \tau \cdot post\text{-}Release(\sigma(evs(i)), tid, \sigma(evs(i) + \delta))
\end{aligned}$$

Technically, the load assumptions define a set of potential time traces (*Events*). Since the assumptions about load are used as rely conditions for the specific *Scheduler*, an implementation that is shown to satisfy the specification will achieve the deadlines of any job stream that the planning phase was asked to *Confirm*.

3.4 Reestablishing normal mode

The change to fault-tolerant modes is covered in §3.5 (see *Overrun*). Reestablishing normal operation is of course desirable. Reversion to normal mode can be made safely when there are no active jobs.

ModeUp

ext rd $active$: $JobId \xrightarrow{m} JobInfo$
  wr $mode$: $Mode$

pre $active = \{\} \wedge mode \neq$ Norm

post $mode' =$ Norm

Alternative schemes that enable an earlier return to the Norm mode are also available [7, 8, 27, 38, 43].

3.5 Assumptions about *Job*s

To complete the definition of the framework, we need to define the behaviour of each Job including their completions. Notice that *rely-Job* matches *inv-State*, *guar-Job* expresses that the job will not exceed its mode specific WCET and *post-Job* expresses the changes that *work* must make to *shared* variables etc.

[7] This is a slight simplification in that jobs for time triggered tasks are triggered internally and are assumed never to arrive too early.

Job (*j*: *JobId*)

ext rd t : $ClockValue$
rd $mode$: $Mode$
rd tkm : $TaskMap$
wr $active$: $JobId \xrightarrow{m} JobInfo$
wr $used$: $JobId \xrightarrow{m} Duration$
wr $shared$: $Id \xrightarrow{m} Value$

rely $t \leq t' \wedge$
let $mk\text{-}JobInfo(type, rel) = active(j)$ **in**
let $mk\text{-}TaskInfo(per, k, ldm) = tkm(type)$ **in**
$ldm(mode).fness \neq \textsc{Soft} \Rightarrow t' \leq (rel + ldm(mode').D)$

guar $used'(j) \leq tkm(active(j).type).loadm(mode').C \wedge$
$\{j\} \lhd active' = \{j\} \lhd active \wedge$
$\{j\} \lhd used' = \{j\} \lhd used \wedge$

post $work(shared, shared') \wedge j \notin \mathbf{dom}\ active'$

Overrun constrains the required behaviour when a job executes for more than its current C value. In this situation either the job is aborted or there must be a mode change. The subset of modes that can experience the need for a mode change is identified in the pre condition; the modes that are transitioned to are identified in the post condition. The post condition also specifies necessary changes to the set of active tasks and other parameters that define the run-time behaviour of the system (this post condition is completed in the instantiation in §4).

Overrun (*j*: *JobId*)

ext wr $mode$: $Mode$
rd tkm : $TaskMap$
wr $active$: $JobId \xrightarrow{m} JobInfo$
wr $used$: $JobId \xrightarrow{m} Duration$

pre $j \in \mathbf{dom}\ active \wedge$
$mode \in \{\textsc{Norm}, \cdots\} \wedge$
$used(j) > tkm(active(j).type).loadm(mode).C$

guar $active' = \{j\} \lhd active$

post $(tkm(active(j).type).loadm(mode).fness \neq \textsc{Hard} \Rightarrow mode' = mode) \wedge$
$(tkm(active(j).type).loadm(mode).fness = \textsc{Hard} \Rightarrow mode' \neq mode \wedge active' = ...)$

3.6 Forcing progress

In order to argue termination or progress, it is standard to show that some aspect of the state changes monotonically towards a limit; in other words, the operations that can change the state define a well-founded relation over states. This applies in the case of the operations of *Scheduler* but the argument is indirect and involves the links between Σ and *State*. In essence, what decreases is the gap between t: *ClockValue* in *State* and the deadlines of any job whose $JobId \in \mathbf{dom}\ active$. The fact that t increases follows directly from $\mathcal{T}$ which is a conjunct of $inv\text{-}\Sigma$. The effective deadline

for any job is the sum of its release time and the relative deadline in the D field of *Load*; it is true that the *Load* varies by *Mode* but there are a (small) finite number of such D entries. The other crucial part of the argument is to establish that $t - (rel + D)$ cannot evaluate to less than zero: this follows from the fact that jobs are assumed not to consume more than their estimated worst case execution time (C in *Load*) or arrive earlier than determined by T (also in *Load*).[8]

In a sense, the aim of the *Scheduler* is to get rid of jobs. The combined effects of *inv*-Σ and *inv-State* require that the *Scheduler* progresses execution of *Job*s as follows. The *Scheduler* selects which *JobId* is in *run* which causes that *Job* to be allocated resources; the change of time is in accord with $\mathcal{E}$ in the corresponding *used* entry. Only because each *Job* removes itself from *active* once it is finished does the demand on the *Scheduler* decrease and this will only occur when the job has been granted enough resource; this requires that its *JobId* is moved into *run*. It is of course the responsibility of planning to determine that executing the chosen discipline (EDF, FP, etc.) will ensure that deadlines are not breached.

3.7 Instantiating the Framework

Having provided a general model, the steps to instantiate it for a particular scheduling regime can be outlined.

(1) Complete the set of operational *Mode*s of the system and the different *Criticality* levels of the application tasks.
(2) Define the *Firmness* of each deadline in each mode; and extend *inv-TaskInfo* where necessary.
(3) Extend as appropriate the *Load* parameters to deal with further resources and temporal properties that may be utilised in the specific scheduling approach that was chosen during *Planning*.
(4) Complete the definition of the *select* predicate to embody the chosen scheduling discipline; this must include the rules governing the execution of jobs with Soft deadlines.
(5) Extend the definition of *inv-State* to incorporate the rules of the scheduling discipline.
(6) Extend the definition of the scheduler and its methods; for example to complete the definitions of *Release* and *Overrun*.
(7) Extend if necessary the definition of *Job*; for example modify *guar-Job* to articulate any mode-specific constraints on its resource usage.

Note the framework does not limit the number of tasks, the number of criticality levels or the number of modes of operation. Any scheduling policy that can be defined via a select predicate can be accommodated – this includes Earliest Deadline First (EDF), Fixed Priority, Round Robin and FIFO. Although the framework as presented utilises only a uniprocessor, §3.1 discusses how it can easily be extended to manage homogeneous multiprocessor and multicore architectures. Although the specification is straightforward, some necessary parameters such as a task's WCET can be more difficult to determine accuratelyin these hardware platforms. The schedulability test can also be more complex (and hence error prone) if jobs can migrate between cores due to the scheduling approach or as part of the mode change protocol.

The application of this framework to a non-trivial mixed-criticality example is given in §4.

3.8 Overall correctness: linking planning and execution

As indicated in §2, scheduling can be split into a preliminary planning phase and the run-time execution of a scheduler together with the jobs associated with the tasks of the application. Clearly both aspects have to be correct and they

[8] If a job with a Brittle deadline overruns (or arrives early), it will be dropped; an overrun of a job with a Hard deadline causes it to be allocated a more relaxed (larger) C value (or smaller T) in the mode to which the system transitions.

have to correspond. The body of this paper focusses on the *Scheduler*; this short section provides links to material on planning and enlarges on its required correspondence with the *Scheduler* — see also the discussion in §5.

The *Framing* part of the planning process must select a scheduling discipline — although, in some cases, it might be dictated by the customer. Given a selection of discipline and parameters for items such as deadlines and job arrival times, hardware must be selected for the run-time system; once this is done, other parameters such as task WCETs can be obtained. The *Confirming* part of planning then has to verify that the job loads can be tolerated in a way that ensures that deadlines will be met.

The aim of the framework developed in sections §2 and §3 of this paper is to define an initial formal model that can be easily extended to cope with the particular behavioural properties of any proposed Mixed-Criticality scheduling protocol. Ideally such a framework would include the basic elements of a schedulability test that could also be extended to furnish the associated *Confirming* test for the new protocol. Unfortunately this is not currently possible as, even for a simple single processor scheme, the two main scheduling methods (EDF and FP) are typically analysed in fundamentally different ways: for EDF the standard schedulability test [6, 49] checks that all deadlines up to some defined point are satisfied;[9] for FP [4, 5, 37] it is usual to calculate first the worst-case response-time (R) for each task and then check that $R \leq D$ for all tasks.

This fundamental difference as to how the common scheduling paradigms are analysed is exacerbated on multi-processor systems where, for example, jobs or tasks may, or may not, migrate between all, or a subset, of the available cores. These varying circumstances lead to many different forms of analysis.

Although no general analysis approach can currently form part of the framework, Bozhko and Brandenburg [10] have shown that it is possible to define a general form of Response-Time Analysis (RTA) that can be applied to systems scheduled using FP or EDF scheduling. One of the advantages of basing a general method of analysis on RTA is that there is evidence (for a basic, single mode, non Mixed-Critically example) that the validity of the RTA schedulability test can be proven with the proofs being checked by a theorem proving assistant; in the case of [9, 10, 17, 41] Coq was used.

Given this potential for RTA to act as a general purpose form of verified analysis we outline, in §5, how it can be applied to the developed framework.

4 An example instantiation of the framework

As noted in the introduction, two previous papers have developed formal specifications for specific mixed-criticality systems. In [36], EDF (Earliest Deadline First) scheduling was employed and the faults that were tolerated were overruns of execution time (the C parameter in the general model) by the most critical tasks. In the other paper [16], FP (Fixed Priority) scheduling was used and the tolerated faults arose from the most critical, event-triggered, tasks arriving for execution too early (the T parameter being compromised). In this instantiation we again use FP scheduling but cater for both forms of fault. We arrive at the formal model by instantiating the general framework and following the steps outlined in §3.7.

If an event-triggered, critical, job arrives 'too early' then a mode change occurs and as a result the inter-arrival parameter (T) is shortened. But to allow this increase in load to be managed, the time-triggered (lower criticality) tasks have their T parameters extended in the new mode. All deadlines however continue to be satisfied.

[9] This is checked by assuming all tasks release their first job at time 0 and then release subsequent jobs as soon as allowed. For all deadlines (d) to be checked the total work to be done by these jobs before time d must be shown to be no greater than d.

Additionally, if an event-triggered job executes for more than its C parameter allows, then a different mode change occurs with the new mode facilitating a larger C for the highest criticality tasks; but in this mode the lower criticality tasks are no longer guaranteed, their deadlines are Soft.

With FP (Fixed Priority) scheduling, the Planning stage both assigns priorities to each task (in each mode) and checks for schedulability by employing a test based on Response-Time Analysis – see §5. This test relies on the run-time scheduler always executing the current job of the active task with the highest priority.

The actual priorities assigned to each task during planning (and assumed for tests of schedulability) can be derived optimally [3] or be obtained via a heuristic (such as rate-monotonic priority assignment) that is proved to be effective (if not actually optimal) [40]. The disadvantage of an optimal assignment scheme is that it can make the schedulability test significantly more complex to derive and apply. All schedulability tests are sufficient, if the test is passed then all the deadlines are guaranteed to be satisfied. But only the optimal tests are also necessary – fail the test and deadlines will be missed.

In this case study we assume that the deadlines are implicit (i.e. D is equal to T for all tasks in all modes). Moreover priorities are allocated using the rate-monotonic algorithm. Hence, in all modes, the ordering of the priorities of the tasks matches the ordering of their T parameter – a shorter T implies a higher *pri*.

4.1 Instantiation of the General Model

Step 1 – Criticality and Modes

Two levels of criticality suffice to illustrate this instantiation:[10] Jobs of High criticality tasks are Event-Triggered — hence *TaskInfo.periodic* is *false*. With Low criticality tasks, they are Time-Triggered so their *TaskInfo.periodic* is *true*.[11]

$Criticality = \{\text{High}, \text{Low}\}$

There are three modes of operation (Normal, Fault Tolerant and Overrun):

$Mode = \{\text{Norm}, \text{Ft}, \text{Over}\}$

Figure 2 illustrates these modes and the allowable transitions: Transition Norm → Ft occurs when Event-Triggered tasks arrive too early. Transitions Norm → Over and Ft → Over occur when a job of a High criticality task executes for too long. The only other allowable transitions are Over → Norm and Ft → Norm which occur when there is an idle instant. This leaves one other possible transition, Over → Ft; this is not however a reasonable behaviour in this example instantiation – in Over, Low criticality tasks are no longer guaranteed to meet their deadlines, whereas in Ft they are, a switch from Over to Ft would therefore not satisfy any application need. It follows from these behaviours that *terminal*(*mode*) is true for Over but false for Norm and Ft.

Step 2 – Firmness

With the modes and criticality levels defined, *inv-TaskInfo* can now be completed with the necessary properties of the task parameters and the firmness of each deadline in each mode being specified. First we note that deadlines are implicit, so T equals D in all situations. Next the C parameters for each task are the same in modes Norm and Ft.

[10] For applications that require more levels of criticality, it might be appropriate to define a partial order over the values (i.e. they do not have to be linearly ordered).

[11] Although this example has a simple relationship between periodicity and criticality this is not in general the case. Hence the general model introduced above has both a 'periodic' flag and a separate *Criticality* parameter.

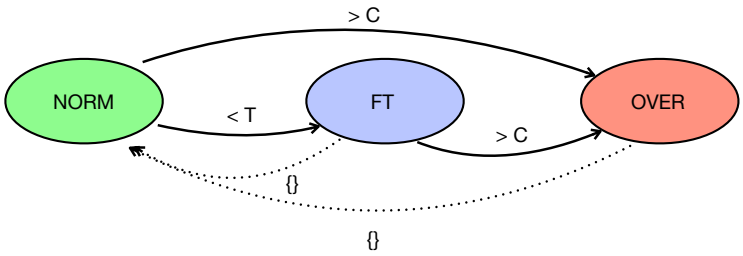

Fig. 2. The modes and their transitions. The green mode is the initial mode in which all deadlines are met. The red mode is *terminal*. Other (blue) modes define degraded behaviour. Transitions marked with $< T$ occur when a job arrives too early; those with $> C$ imply a job has executed for too long. Dashed transitions can only occur when the set of active tasks is empty ({}).

For the HIGH criticality tasks their periods (T) are potentially shorter in mode FT as in mode NORM, but are unchanged in mode OVER. However the C parameters are greater in OVER. As OVER is a terminal mode the firmness of the HIGH criticality tasks is BRITTLE in that mode but HARD in the other two.

The LOW criticality tasks have a related set of properties; periods are potentially longer in FT, computation times are smaller in OVER and deadlines are BRITTLE in modes NORM and FT but SOFT in OVER.

So in OVER only HIGH criticality tasks are guaranteed to execute and complete by their deadlines. LOW criticality periodic tasks are however still released for execution, but have SOFT deadlines. There is no further degradation from this mode (as the OVER mode is terminal).

The above properties are formalised in the specification of *inv-TaskInfo* on the next page – the lines in red being copied from the general model derived in §2.2.

Steps 3, 4 and 5 – Defining the Scheduling Approach

Here we need to add what is required to define FP scheduling. The run-time task parameter, priority, must be added to *TaskInfo*; since tasks may have different priorities in different modes, the *pri* parameter is added to *Load*:

$$
\begin{array}{l}
Load :: \cdots \\
\qquad pri : \mathbb{N}
\end{array}
$$

The *select* predicate from §3.8 is extended to reflect the preemptive FP discipline; furthermore the behaviour of jobs with SOFT deadlines must be defined. Here we require the priority of such jobs to be less than that of any active job with a HARD or BRITTLE deadline:

$$
\begin{array}{l}
select : (JobId \xrightarrow{m} JobInfo) \times TaskMap \times Mode \times \left[JobId\right] \to \mathbb{B} \\
\\
select(active, tkm, mode, r) \quad \triangleq \\
\quad (active = \{\} \wedge r = \textbf{nil} \vee r \in \textbf{dom}\ active) \wedge \\
\quad \forall i, j \in \textbf{dom}\ active \cdot \\
\qquad tkm(active(j).type).loadm(mode).pri \leq tkm(active(r).type).loadm(mode).pri \wedge \\
\qquad (tkm(active(i).type).loadm(mode).fness = \text{SOFT} \wedge tkm(active(j).type).loadm(mode).fness \neq \text{SOFT} \Rightarrow \\
\qquad\qquad tkm(active(i).type).loadm(mode).pri < tkm(active(j).type).loadm(mode).pri)
\end{array}
$$

There is no need for any further extensions to *inv-State* in this example.

$$inv\text{-}TaskInfo : TaskInfo \to \mathbb{B}$$

$$\begin{aligned}
&inv\text{-}TaskInfo(info) \triangleq \\
&\quad info.loadm(\textsc{Norm}).fness \neq \textsc{Soft} \wedge \\
&\quad (\forall mode \in Mode \cdot \\
&\qquad (terminal(mode) \implies info.loadm(mode).fness \neq \textsc{Hard}) \wedge \\
&\quad info.loadm(mode).D = info.loadm(mode).T \ ) \wedge \\
&\quad info.loadm(\textsc{Norm}).C = info.loadm(\textsc{Ft}).C \wedge \\
&\quad (info.Criticality = \textsc{High} \implies \\
&\qquad \neg\, info.periodic \wedge \\
&\qquad info.loadm(\textsc{Ft}).T \leq info.loadm(\textsc{Norm}).T \wedge \\
&\qquad info.loadm(\textsc{Ft}).T = info.loadm(\textsc{Over}).T \wedge \\
&\qquad info.loadm(\textsc{Norm}).C \leq info.loadm(\textsc{Over}).C \wedge \\
&\qquad info.loadm(\textsc{Norm}).fness = \textsc{Hard} \wedge \\
&\qquad info.loadm(\textsc{Ft}).fness = \textsc{Hard} \wedge \\
&\qquad info.loadm(\textsc{Over}).fness = \textsc{Brittle} \ ) \wedge \\
&\quad (info.Criticality = \textsc{Low} \implies \\
&\qquad info.periodic \wedge \\
&\qquad info.loadm(\textsc{Norm}).T \leq info.loadm(\textsc{Ft}).T \wedge \\
&\qquad info.loadm(\textsc{Norm}).C \geq info.loadm(\textsc{Over}).C \wedge \\
&\qquad info.loadm(\textsc{Norm}).fness = \textsc{Brittle} \wedge \\
&\qquad info.loadm(\textsc{Ft}).fness = \textsc{Brittle} \wedge \\
&\qquad info.loadm(\textsc{Over}).fness = \textsc{Soft} \ )
\end{aligned}$$

Step 6 – Updating the Scheduler and its methods

With this example there is no need to modify the definition of the Scheduler (i.e. its pre, rely or guar conditions).

The definition of the *Release* method in the general model covers all the required behaviours. If a job arrives too early and has a HARD deadline then the current mode must be NORM and a mode change to FT is required. Hence rather than just note that the mode must change ($mode' \neq mode$ in *post-Release* in §3.3) we can be explicit as to the mode that must be transitioned to ($mode' = \textsc{Ft}$).

Two minor modifications are also required to *Overrun* to make the modes involved explicit. Here the changed lines are shown in blue.

Overrun (j: *JobId*)

ext wr *mode* : *Mode*
rd *tkm* : *TaskMap*
wr *active* : $JobId \xrightarrow{m} JobInfo$
wr *used* : $JobId \xrightarrow{m} Duration$

pre $j \in \textbf{dom}\ active \wedge$
$\quad mode \in \{\textsc{Norm}, \textsc{Ft}\} \wedge$
$\quad used(j) > tkm(active(j).type).loadm(mode).C$
guar $active' = \{j\} \lhd active$
post $(tkm(active(j).type).loadm(mode).fness \neq \textsc{Hard} \Rightarrow mode' = mode) \wedge$
$\quad (tkm(active(j).type).loadm(mode).fness = \textsc{Hard} \Rightarrow mode' = \textsc{Over} \wedge active' = active)$

There are no changes needed to the simple *ModeUp* method.

Step 7 – Extend the definition of *Job*

The behaviour of each job, apart from when it overruns, is independent of the chosen scheduling approach, it therefore does not need to be modified.

4.2 Summary

The above seven steps illustrate how the general framework can be instantiated to define the required behaviour for a particular non-trivial mixed-criticality model. Most elements of the framework are unchanged (definitions of *TaskInfo*, *State*, *inv-State*, Σ and *inv-*Σ, *ModeUp*, *inv-trigger* and *Job*), others are subject to minor alterations – as the allowable modes of the system are now fixed, (*Scheduler*, *Release* and *Overrun*). The main additions to the framework are the concrete modes and criticality levels (and hence significant extensions to *inv-TaskInfo* to capture the rules for transitioning between the modes). There is also the definition of the chosen scheduling protocol (as embodied in a completed definition of *select* and the addition of the *pri* parameter to *Load*). In summary, the proposed framework significantly reduces the effort needed to derive a full formal specification of a new mixed-criticality scheduler.

5 Response-Time Analysis (RTA) for the Framework

In this section we develop a general form of RTA that can be tailored to apply to the mixed-criticality protocols defined by the developed framework. We also briefly outline how this analysis can be applied to the instantiation of the framework developed in the previous section.

To keep the formulae in this section compact and looking familiar to readers who are familiar with the RTA literature, a number of abbreviations are adopted: for particular *tkm*: *TaskMap* and tid_i: *TaskId*:

$D_i = (tkm(tid_i).loadm)(\textsc{Norm}).D$
$C_i = (tkm(tid_i).loadm)(\textsc{Norm}).C$
$T_i = (tkm(tid_i).loadm)(\textsc{Norm}).T$
$R_i = (tkm(tid_i).loadm)(\textsc{Norm}).R$

5.1 General Response-Time Analysis, RTA

Response-time analysis computes, for each task tid_i: *TaskId*, the longest period of time until a job of tid_i will complete its execution (R_i); this must reflect both the WCET (C_i) of the task and the maximum total interference that the job can experience. RTA works on one task at a time and uses a fixed-point equation to estimate the maximum load that all the other tasks can generate that will interfere with tid_i during its response time (of length R_i, from time of release which is assumed without loss of generality to be time 0).

$$R_i \;=\; C_i \;+\; \sum_{tid_j \in TaskId} I_j(R_i) \tag{1}$$

where $I_j(R_i)$ is the total (i.e. maximum) interference jobs from task tid_j can have on task tid_i in any interval of length R_i.

As noted in §3.8, the actual values of I to be used in eq(1) depend on the scheduling discipline chosen. For EDF scheduling, all jobs with a deadline at or before $0 + R_i$ will count towards I; for fixed priority scheduling it will be all jobs that have been released before R_i and have a higher (or equal) priority than tid_i. In addition, resource sharing protocols and non-preemptive code within the OS may result in other jobs causing a form of interference that is usually referred to as *blocking*. If, as is usually the case, the deadline (D) for a task is no greater that its period (T) then a task cannot interfere with itself ($I_i = 0$); however if $D > T$ it is possible for one job of a task to interfere with the next job of the same task. Analysis for this scenario is available [13] but is not included here.

Equation eq(2) is a recurrence relation because the I term in eq(1) extends R_i. All equations derived from eq(1) are solved using the standard techniques for solving recurrence relations; i.e. fixed point iteration:

$$r_i^{n+1} \;=\; C_i \;+\; \sum_{tid_j \in N} I_j(r_i^n) \tag{2}$$

with the initial value r_i^0 being set to C_i, and $r_i^{n+1} \geq r_i^n$. To compute the least fixed point, the iteration stops when either $r_i^{n+1} = r_i^n$, in which case r_i^n is the task's response time ($R_i = r_i^n$), or $r_i^{n+1} > D_i$ in which case this task, and hence the whole task set, is unschedulable[12].

5.2 Response-Time for the Multi-Modal Framework using Fixed Priority Scheduling

In this section we review the published method of using RTA to test a fixed priority mixed-criticality system with two modes (Norm and Term) and hence one mode switch of importance. This scheme is capable of being applied to EDF (and other scheduling schemes that are amenable to Response-Time Analysis). It is also extendable to more than two modes of operation, although the analysis becomes more complex. This is discussed again in §5.6 where the approach is applied to the instantiation of the general framework given in §4.

For fixed-priority scheduling eq(1) becomes:

$$R_i \;=\; B \;+\; C_i \;+\; \sum_{tid_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3}$$

the summation being over all tasks with a higher (or equal) priority than that of tid_i. The blocking term, B is the maximum interference a job of lower priority can have on tid_i. This arises from the management of shared resources and non-interruptible code within the OS. It is straightforward to include this (constant) term, but for ease of presentation it will be omitted from the following discussions.

5.3 Steady state analysis for Norm and Term

The two modes of operation are Norm and a terminal mode which is given the simple name Term.

The complete analysis consists of first checking that these two modes are schedulable, and, if they are, checking that the worst-case mode change (from Norm to Term) is also schedulable. The approach applied in this paper follows that used to derive RTA for the original analysis [5] of the Mixed-Criticality System model developed by Vestal [46].

[12] More efficient methods of undertaking RTA are available [19].

The first step is to apply eq(3) to the steady state behaviour of each mode. This will generate mode-specific worst-case response times (i.e. R_i(Norm) and R_i(Term)). If these values are less than the associated deadlines for each mode and all tasks then the actual mode change can be addressed.

5.4 Mode change analysis

We again analyse each task in turn, and assume the mode change from Norm to Term occurs at arbitrary time S. Figure 3 represents the worst-case behaviour of task tid_i. Note S must be before R_i(Norm) as tid_i must complete before R_i(Norm) in mode Norm, and the switch must occur before it has completed. The sequence of r_i^n values will start at S and will either iterate to the solution, $R_i(S)$, or obtain a value that is greater than D_i(Term) in which case tid_i cannot be guaranteed to meet its deadline during a mode change taking place at time S. Different values of S are likely to lead to different solutions (different values of $R_i(S)$). The final piece of the analysis will be to identify the longest response time:

$$R_i = \max_{S \in \{0..R_i(N)\}} R_i(S) \tag{4}$$

which by construction will have a value less than T_i(Term). Fortunately, as proven in [5], not all values of S need be evaluated, rather only values of S that correspond to the release of a new job of a lower criticality but higher priority need to be considered.

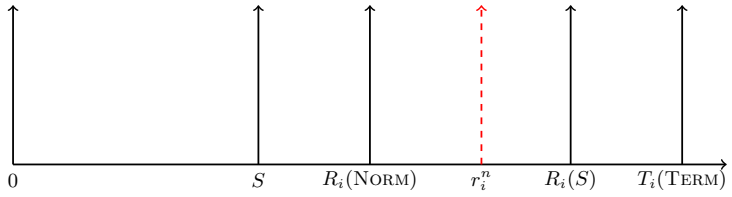

Fig. 3. Schedulable Mode Switch of tid_i at time S

5.5 Interference terms

Given a specific value of S when the mode of the system changes from Norm to Term and a current value of r_i^n for whatever task is being analysed, it is necessary to compute the maximum interference that this task can suffer in the interval from 0 to r_i^n. This allows a new value of r^n (i.e. r_i^{n+1}) to be computed. Interference comes from the execution of higher priority tasks. There are four sources of this interference in the model adopted in this paper, all of which apply to jobs with non-Soft deadlines:

(1) Tasks that have a higher priority in Norm but a lower priority in Term.
(2) Tasks that have a lower priority in Norm but a higher priority in Term.
(3) Tasks of lower criticality that have higher priority in both modes.
(4) Tasks of higher criticality that have a higher priority in both modes.

Note that tasks may change their priority at time S (when the mode changes) and that higher criticality tasks may have their execution time budgets increased or their periods decreased at S while lower criticality tasks (if they still have a Brittle deadline and a higher priority) may have a lower budget and/or a longer period.

5.6 Applying RTA to the example instantiation

As illustrated in Figure 2 the example instantiation has three modes (Norm, FT and Over). Each of these modes must first be assessed and then the mode changes from Norm to FT and Norm to Over. However the mode change from FT to Over need not be analysed as its worst case behaviour is covered by the following test.

With the three modes of this example the worst-case behaviour may well occur when a mode change from Norm to FT is followed almost immediately by a mode change from FT to Over. This would happen if a high criticality job was released early and then executed for longer than allowed in the more constrained modes. To analyse this occurrence, two mode change switch times, S^1 and S^2, are required. The mode change from Norm to FT occurs at time S^1 with $S^1 < R_i(\text{Norm})$. The mode change from FT to Over then occurs at time S^2 with $S^1 < S^2 < R_i(S^1)$. The $R_i(S^1)$ values are obtained from the single mode change analysis that has already been carried out ($S^1 = S$).

The response-time analysis follows the same pattern as before [5]. The interference from higher priority jobs is computed for (and maximised over) the three intervals, 0 to S^1, S^1 to S^2 and S^2 to r_i^n. Overall, as the number of modes, and hence mode changes, increases the number of steps that need to be analysed grows exponentially. Fortunately RTA is pseudo-polynomial and three or more modes can easily be accommodated. In practice most papers on Mixed-Criticality Analysis limit their models to less than six modes.

6 Conclusions

The main contribution of this paper is that a generic model is provided which significantly decreases the effort required to create a formal specification of any specific real-time scheduling scheme. There is extensive published work on Mixed-Criticality scheduling and implementation, but not on their formal specification. We believe formalisation is essential since the notion of mixed criticality as applied within critical Cyber Physical Systems has subtle semantics: often concepts such as correctness, completeness, resilience and robustness are neither straightforward nor intuitive for such systems.

The challenges –and their resolutions– of providing a formal model include:

- separating the assumptions that developers can make from the requirements on the code of the run-time *Scheduler*: use of rely and guarantee conditions, and state invariants;
- delivering managed fault-tolerance: use of layers of rely-guarantee conditions with respect to the *Mode* of execution;
- addressing the distinction between *Time* in the world external to the software and its internal *ClockValue*s: use of notion of precision from the definition of time bands;
- requiring that the methods of the *Scheduler* make progress: use of invariants that relate to deadlines, execution times and *ClockValue*s.

The developed framework has three main elements; (i) *TaskInfo* and *JobInfo*, (ii) *State* and Σ, (iii) the *Scheduler* and the running *Job*s. These relate to (i) the static and dynamic properties of each task and the jobs they create, (ii) the state of the system and its relation to time in the system's environment, and (iii) the run-time scheduler with its methods that control the release of jobs, their termination, and if necessary their fault management, and entities that contain the computations to be undertaken by each job. Key properties of the run-time system are enforced by the datatype invariants: *inv-TaskInfo*, *inv-State* and *inv*-Σ, and by related rely and guarantee conditions.

During *Planning* the framework is expanded to cover the particular properties of the proposed system's run-time behaviour.

The literature on formalising response time analysis includes [9, 10, 17, 18, 21, 41]; of which we consider it easiest to build a bridge to the recent research at MPI Kaiserslautern. (An alternative avenue would be to link to [42] and their use of the Duration Calculus.) Our future work will focus on the schedulability test that is derived and applied during *Planning* to ensure that the application's deadlines will always be satisfied. Ideally such tests will be proven to be correct and their implementation verified using appropriate software tools. We aim to derive a general purpose test that complements the other aspects of the framework. An initial approach will focus on RTA (Response Time Analysis) that is well developed for priority based scheduling, but is currently less usable for EDF (Earliest Deadline First) scheduling. We will also hope to build on the theorem proving assistant work cited in §5.

Acknowledgements

The authors' initial collaboration started under funding from UK EPSRC and was continued under their Platform Grant scheme. This research was funded in part by Innovate UK SCHEME project (10065634) and EPSRC Research Data Management (no new primary data was created during this study). Jones' research was also supported by RPG-2019-020 from the Leverhulme Foundation and polished by discussions at IFIP WG 2.3 and the *Big Specification* meetings at the Isaac Newton Institute in Cambridge.

The authors are grateful to Björn Brandenburg for fruitful discussions on the related work at MPI Kaiserslautern; useful input on readability from the anonymous referees is also gratefully acknowledged.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
[2] J.-R. Abrial. *The Event-B Book.* Cambridge University Press, Cambridge, UK, 2010.
[3] N.C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.
[4] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.
[5] S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.
[6] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptive scheduling of hard real-time sporadic tasks on one processor. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.
[7] I. Bate, A. Burns, and R.I. Davis. A bailout protocol for mixed criticality systems. In *Proc. 27th ECRTS*, pages 259–268, 2015.
[8] I. Bate, A. Burns, and R.I. Davis. An enhanced bailout protocol for mixed criticality embedded software. *IEEE Transactions on Software Engineering*, 43(4):298–320, 2016.
[9] Kimaya Bedarkar, Mariam Vardishvili, Sergey Bozhko, Marco Maida, and Björn B Brandenburg. From intuition to Coq: A case study in verified response-time analysis of FIFO scheduling. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 197–210. IEEE, 2022.
[10] Sergey Bozhko and Björn B Brandenburg. Abstract response-time analysis: A formal foundation for the busy-window principle. In *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, pages 6.1–6.23, 2020.
[11] A. Burns and R.I. Davis. A survey of research into mixed criticality systems. *ACM Computer Surveys*, 50(6):1–37, 2017.
[12] A. Burns and R.I. Davis. Mixed criticality systems: A review (13th edition). Technical Report MCC-1(13), available at https://www-users.cs.york.ac.uk/ab38/review.pdf and the White Rose Repository, Department of Computer Science, University of York, 2022.
[13] A. Burns and A. J. Wellings. *Analysable Real-Time Systems Programmed in Ada.* ISBN: 9781530265503, 2016.
[14] Alan Burns and Ian J. Hayes. A timeband framework for modelling real-time systems. *Real-Time Systems*, 45(1–2):106–142, 6 2010.
[15] Alan Burns, Ian J. Hayes, and Cliff B. Jones. Deriving specifications of control programs for cyber physical systems. *The Computer Journal*, 63(5):774–790, 2020.
[16] Alan Burns and Cliff B. Jones. Specifying fault-tolerant mixed-criticality scheduling. In Simon Foster and Augusto Sampaio, editors, *The Application of Formal Methods: Essays Dedicated to Jim Woodcock on the Occasion of His Retirement*, pages 22–42. Springer Nature Switzerland, 2024.
[17] Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. Prosa: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 273–284. IEEE, 2016.
[18] Albert MK Cheng. *Real-time systems: scheduling, analysis, and verification.* John Wiley & Sons, 2003.
[19] R.I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority pre-emptive systems. *IEEE Transaction on Computers*, 57(9):1261–1276, 2008.

[20] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[21] B. Dutertre and V. Stavridou. Formal analysis for real-time scheduling. In *19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No.00CH37126)*, volume 1, pages 1D4/1–1D4/7 vol.1, 2000.

[22] R. Ernst and M. Di Natale. Mixed criticality systems?a history of misconceptions? *IEEE Design & Test*, 33(5):65–74, 2016.

[23] A. Esper, G. Neilissen, V. Neils, and E. Tovar. How realistic is the mixed-criticality real-time system model. In *Proc. 23rd International Conference on Real-Time Networks and Systems (RTNS 2015)*, pages 139–148, 2015.

[24] P. Graydon and I. Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *Proc. WMC, RTSS*, pages 19–24, 2013.

[25] I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall International, second edition, 1992.

[26] I. J. Hayes and C. B. Jones. A guide to rely/guarantee thinking. In Jonathan Bowen, Zhiming Liu, and Zili Zhan, editors, *Engineering Trustworthy Software Systems – Third International School, SETSS 2017*, volume 11174 of *Lecture Notes in Computer Science*, pages 1–38. Springer-Verlag, 2018.

[27] S. Iacovelli and R. Kirner. A lazy bailout approach for dual-criticality systems on uniprocessor platforms. *Designs*, 3(1), 2019.

[28] D. N. Jackson. *Software Abstractions: logic, language, and analysis*. MIT Press, 2012.

[29] Z. Jiang. How to build a mixed-criticality system in industry - from perspective of system architecture. In J. Li and Z. Guo, editors, *Proc. 7th WMC Workshop, IEEE Real-Time Systems Symposium (RTSS), year = 2019*, pages 9–14, 2019.

[30] Z. Jiang. How to build a mixed-criticality system in industry - from the perspective of system architecture. In *Proc. 8th WMC Workshop, IEEE Real-Time Systems Symposium (RTSS)*, pages 510–517, 2020.

[31] Z. Jiang, S. Zhao, P. Dong, D. Yang, R. Wei, N. Guan, and N. Audsley. Re-thinking mixed-criticality architecture for automotive industry. In *Proc. IEEE 38th International Conference on Computer Design (ICCD)*, pages 510–517, 2020.

[32] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, 6 1981. Printed as: Programming Research Group, Technical Monograph 25.

[33] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[34] C. B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.

[35] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, second edition, 1990.

[36] Cliff B. Jones and Alan Burns. Extending rely-guarantee thinking to handle real-time scheduling. *Formal Methods in System Design*, 62(1):119–140, 2024.

[37] M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Computer Journal*, 29(5):390–395, 1986.

[38] S. Law, I. Bate, and B. Lesage. Justifying the service provided to low criticality tasks in a mixed criticality system. In *Proc 30th International Conference on Real Time Networks and Systems, RTNS*, pages 100–110. ACM, 2020.

[39] J. Lee and M. Kim. Generalized models of mixed-criticality systems for real-time scheduling. *Trans Eng Comput Sci*, 1:1–50, 2020.

[40] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.

[41] Marco Maida, Sergey Bozhko, and Björn B Brandenburg. Foundational response-time analysis as explainable evidence of timeliness. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[42] Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, 2008.

[43] A.V. Papadopoulos, E. Bini, S. Baruah, and A. Burns. AdaptMC: A Control-Theoretic Approach for Achieving Resilience in Mixed-Criticality Systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proc. in Informatics (LIPIcs)*, pages 14:1–14:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[44] M. Paulitsch, O.M. Duarte, H. Karray, K. Mueller, D. Muench, and J. Nowotsch. Mixed-criticality embedded systems–a balance ensuring partitioning and performance. In *Proc. Euromicro Conference on Digital System Design (DSD)*, pages 453–461. IEEE, 2015.

[45] F. Reghenzani and W. Fornaciari. Mixed-criticality with integer multiple WCETs and dropping relations: new scheduling challenges. In *Proc. 28th Asia and South Pacific Design Automation Conference*, pages 320–325, 2023.

[46] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.

[47] R. Wilhelm. Mixed feelings about mixed criticality (invited paper). In Florian Brandner, editor, *Proc. 18th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 1:1–1:9, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[48] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall International, 1996.

[49] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transaction on Computers*, 58(9):1250–1258, 2008.

A Appendix – Notation

As indicated in §1.1, the specific syntax of a formal notation is unimportant and the formulae in the current paper could readily be transcribed into Z [25, 48], B [1], Event-B [2] or Alloy [28]; this paper uses the widely know VDM [35] which was the subject of an international standard in 1996 and last revised in 2008.

The notation for describing the objects manipulated is straightforward. The base types used are Booleans ($\mathbb{B}$) and natural numbers ($\mathbb{N}$). Set types are defined by X-**set** and values can take part in expressions using familiar operators: union $S \cup T$ and test for membership $e \in S$. The empty set is written $\{\ \}$; sets of enumerated constants can be defined as in *Firmness* in §2.1; such constants can only be compared for equality (e.g. HARD $\neq$ SOFT);

Map values ($D \xrightarrow{m} R$) are sets of pairs with the requirement of a many:one association so that they can be applied to arguments in the same way as functions; the domain of a map value is **dom** m thus $m \in (D \xrightarrow{m} R) \wedge d \in \textbf{dom}\ m \Rightarrow\ m(d) \in R$. The other map operators are most easily described in terms of sets of pairs but notice that each operator preserves the many:one property:

The union of maps is only defined where: $\textbf{dom}\ m1 \cap \textbf{dom}\ m2 = \{\ \}$
Domain restriction: $s \lhd m = \{(d, r) \in m \mid d \in s\}$
Domain subtraction: $s \lhd\!\!\!- m = \{(d, r) \in m \mid d \notin s\}$
Map overwrite: $m1 \dagger m2 = (\textbf{dom}\ m2 \lhd\!\!\!- m1) \cup m2$

Finite sequence types can be thought of as maps from natural numbers ($X^* = (\mathbb{N} \xrightarrow{m} X)$). The length of a sequence value is **len** s and elements can be accessed by indexing; the set of indexes is **inds** $s = \{1, \cdots, \textbf{len}\ s\}$ so:

$s \in X^* \wedge i \in \textbf{inds}\ s\ \Rightarrow\ s(i) \in X$

Records are valuable when writing formal specifications because the named (and typed) fields are more readable than anonymous tuples — see for example *Load* in §2.1. Optional values are marked by $\lfloor X \rfloor$ and are equivalent to $X \cup \{\textbf{nil}\}$. Selection of fields from a record value is made by postfixing *.field*. Predicate restriction of record types by invariants is exemplified in *inv-State* in §3.1. Allied with each record description is a constructor function that builds record values from those suitable for its fields but only applies to values which satisfy the invariants. The ranges of distinct *mk-* functions are automatically disjoint. Furthermore, these constructor functions can be used on the left of local **let** definitions to decompose a record value to provide names for its field values (see *rely-Job* in §3.5); it is useful to echo the names of the record fields for the values but they can be abbreviated since these are local bindings. Functions are normally presented with their signatures; predicates are functions whose range is $\mathbb{B}$ (see *inv-TaskInfo* in §2.2). Notice that, unlike maps, mathematical functions can have infinite domains.

In a sequential setting, so-called 'operations' are like functions which take –and yield– an object of type state. Thus *Release* in §3.3 can change *State* objects but its actual read/write access to the fields of these objects are defined after the **ext**ernal keyword. Pre conditions for operations define the set of starting states in which the operation is required to work; post conditions are predicates of the initial and final states and specify the permissible relation between them (values in the final state are distinguished by primes, e.g. *active′*). Notice that the required results might not be unique; specifications can permit non-determinism. There is a 'satisfiability' requirement that there must be at least one possible final state for any initial state. Rely and guarantee conditions as they apply to concurrent systems are described in §1.1.

In addition to the state transitions which are essentially viewed as discrete, some phenomena of real-time systems have to be described over continuous time; this extension is described in §3.2. There is also a need to specify that an operation is executed in a particular time interval; since operations cannot be 'invoked' from logical expressions,

this effect is achieved by quoting the post condition of the operation (see *inv-Release-timing* in §3.3); this approach is described in [35, §9.1].