Formal Analysis of Predictable Data Flow in Fault-Tolerant Multicore Systems

Jalil Boudjadar[1], Boris Madzar[2],
Juergen Dingel[2], Thomas E. Fuhrman[3], Ramesh S[3]

[1] Linköping University, Sweden
[2] Queen's University, Canada
[3] General Motors R&D, Warren MI USA

Abstract. The need to integrate large and complex functions into today's vehicle electronic control systems requires high performance computing platforms, while at the same time the manufacturers try to reduce cost, power consumption and ensure safety. Traditionally, safety isolation and fault containment of software tasks have been achieved by either physically or temporally segregating them. This approach is reliable but inefficient in terms of processor utilization. Dynamic approaches that achieve better utilization without sacrificing safety isolation and fault containment appear to be of increasing interest. One of these approaches relies on predictable data flow introduced in PharOS and Giotto. In this paper, we extend the work on leveraging predictable data flow by addressing the problem of how the predictability of data flow can be proved formally for mixed criticality systems that run on multicore platforms and are subject to failures. We consider dynamic tasks where the timing attributes vary from one period to another. Our setting also allows for sporadic deadline overruns and accounts for criticality during fault handling. A user interface was created to allow automatic generation of the models as well as visualization of the analysis results, whereas predictability is verified using the Spin model checker.

1 Introduction

Automotive electronic control systems demand increasing computing power to accommodate the ever-growing software functionality in modern vehicles. At the same time, the trend in automotive electronic architectures is to allocate this increasing computational load to a reduced number of physical processing cores in an effort to reduce size, weight, and power consumption. These trends lead to new design challenges where an increasing number of software-based features must be grouped into tasks which must in turn be allocated to processing cores. The tasks assigned to a given processor may reflect different levels of safety-criticality (referred to as "mixed-criticality integration"). These types of mixed-criticality systems need to meet the requirements of the software processes sharing processors and resources.

To ensure correct operation of a critical functionality that shares processing resources with a less-critical functionality, the ISO 26262 standard for functional

safety of road vehicles [1] requires mechanisms for "freedom-from-interference". Accordingly, the mapping of tasks to cores and the scheduling must take into account multiple factors such as criticality and balancing the workload while ensuring freedom from interference and fault containment.

A system whose externally-observable behavior changes only when its inputs (in terms of values or timestamps) change is said to be *predictable* [11]. Ensuring the predictability of task sets with dynamic runtime attributes and executing on multicore platforms with static analysis alone is very difficult due to the interference caused by the delays for shared resources allocation. The challenge gets even harder due to dynamic runtime and fault handling mechanisms.

Our paper introduces a formal framework for the predictability analysis of mixed criticality task sets running on a multicore platform. The framework supports window scheduling and dynamic runtime of tasks, where the attributes may vary from one window to another. It also supports fault-tolerance via runtime fault handling mechanisms. In addition to the window-based predictable data flow [6], we add support for the preservation of predictability even in cases where the scheduling constraints are violated (deadline overruns). Using this framework, we identify and implement a strategy for exhaustive verification of predictability and freedom from criticality inversion. We observe that the verification can be reduced to the checking of a specific set of fixed "edge tasks" by showing that these edge tasks never produce "tainted" (i.e., possibly not trustworthy) output. A prototype implementation to facilitate model creation, verification and result visualization is sketched.

The paper is organized as follows: Necessary background is discussed in Section 2. Section 3 describes how predictability can be guaranteed through limited observability. Sections 4 and 5 present, respectively, key parts of a formalization of the systems we analyze and the verification approach. Section 6 sketches the prototype and the case study we analyzed. Section 7 presents related work and Section 8 concludes the paper.

2 Background

Freedom-from-interference in a mixed criticality real-time system can be achieved in several ways. The most common approach segregates the different criticality levels in such a way that they are guaranteed not to interfere (run-time guarantee). Some examples of this kind of segregation include assigning each criticality level to its own processor core, assigning a fixed (though not necessarily equal) time slice to each criticality level, or a combination of the two.

The potential downside of the segregation of criticality levels is poor processor utilization. If one set of segregated tasks finishes early, the spare processor time cannot be given to another set of tasks. A common approach to setting up a task set is to assign each task a period, budget and a priority. The task's execution start, end and duration may vary within the period. Such a variability comes from several sources: inputs from the environment, sharing of the resources and the internal behavior of the task.

A potential problem with this approach, identified by Henzinger [11], is that the times at which a task reads and writes data vary in relation to the start of the period. The values visible to any given task T may change depending on its execution time and the execution order of tasks supplying T with data. Therefore, two correct executions may produce different outputs given identical input (see Figure 1.(a)), which leads to a violation of predictability.

2.1 Criticality

Tasks can be defined as belonging to different criticality levels. Following the work of [7], criticality does not need to be considered during regular (fault-free) execution. It only needs to be taken into account when a fault occurs and load-shedding must take place. Critical tasks must be prioritized over non-critical tasks, as they represent behavior that must occur to preserve important properties of the system.

Commonly, the most efficient algorithms for assigning task priorities, such as Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF), use timing properties rather than criticality levels. Thus, a mixed strategy needs to be set up to handle both timeliness and criticality correctly: one way of doing this is through the use of zero-slack scheduling [7].

Criticality interacts with how faults are handled, as it is possible to be either more lax or more strict with faults. For instance, critical tasks may be allowed to miss their deadlines, whereas non-critical tasks would be terminated.

2.2 Zero-slack Scheduling

The idea behind the zero-slack algorithm [7] is that the worst case execution time (WCET) of a task is often much too pessimistic when compared with the average case execution time. Any algorithm that seeks to ensure freedom from criticality inversion by using the WCET of lower-criticality tasks will under-utilize the processor. It is a much more efficient use of resources if tasks are scheduled regardless of criticality until it becomes absolutely necessary to factor criticality into a scheduling decision. Accordingly, tasks are scheduled in one of two modes: "normal" and "critical". In normal mode, criticality is ignored and an optimal scheduling strategy is used, whereas in critical mode, higher-criticality tasks are given priority. Tasks within a criticality level are scheduled as in normal mode.

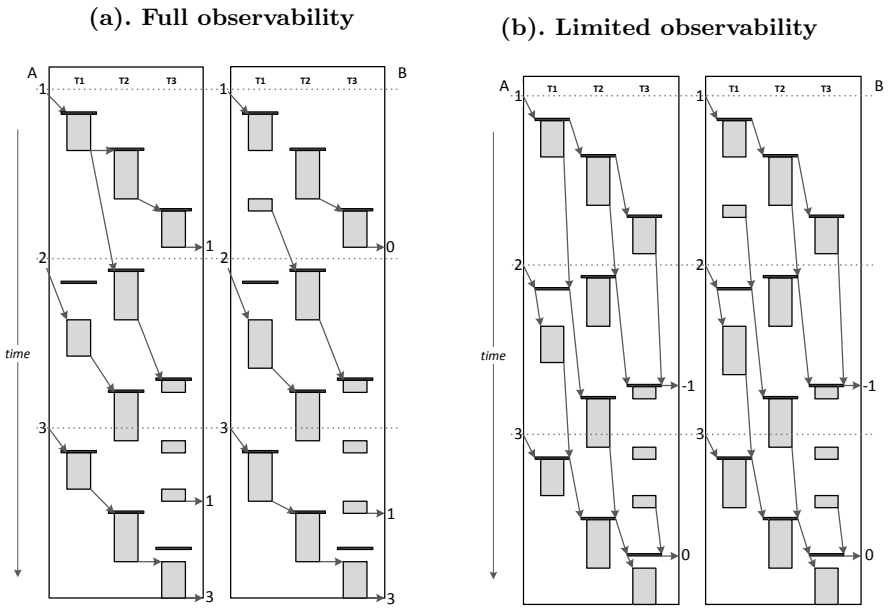
The system usually executes in normal mode. The mode changes to critical when all remaining critical tasks must begin executing if they are to meet their deadlines, assuming they consume their entire execution budgets. This time point is called *zero-slack instant*. A version of this algorithm, known by Simplified Zero-Slack (ZS), uses only two criticality levels. Priorities are assigned using EDF in "normal mode", however in "critical mode" all high-criticality tasks are scheduled before all low-criticality tasks. Within a given criticality level, EDF is still used.

3 Predictability via Limited Observability

An alternative approach to manage tasks while ensuring predictability, by design, has been defined in [12, 6]. This approach focuses on synchronizing data access so that a given task instance will always view the system as being in the same state regardless of when it executes within a given period (*limited observability*). This is achieved by introducing an additional constraint: each task instance must execute in its entirety within a given time window. The start (baseline) and end (deadline) of this window are defined according to the system clock, not the task's CPU time. *A snapshot of the data values is captured at the baseline, so that the task instance reads from this snapshot. Any values written by the task are not communicated to other tasks until the deadline of such a task.*

To illustrate how limited observability can be used to ensure predictability, Figure 1 shows a system with three tasks (left to right): *T1*, *T2* and *T3*. Each task passes its input unchanged to the output. The tasks have periods 14, 10 and 14 and budgets 4, 4 and 3 respectively. Priorities are *T2* > *T1* > *T3*.

Fig. 1: Predictability via limited observability.



We consider 2 different execution scenarios A and B for each observability class (full and limited). Each execution scenario consists of 3 periods for each task. Inputs (1, 2 and 3; on the left hand column of each execution) occur at the exact same points in time for each execution. A displays one possible valid execution with no task violating any constraint (budget or deadline), whereas in scenario B some tasks overrun, e.g *T1* overruns its budget during its first period. The corresponding outputs are shown to the right of each execution. Figure 1.(a)

does not use data access synchronization (i.e., limited observability): although there is no change in the timing or value of inputs, the outputs of A and B differ in both value and timing, as a task can hand over its output once its execution is over without waiting for the period expiry. Figure 1.(b) shows how the example presented in Figure 1.(a) would behave under limited observability achieved through data access synchronization. Regardless of being overrunning or not, a task delivers its output at the end of its current period. This guarantees that the output of a given task is always delivered at the same point in time. As an example, in the execution B of Figure 1.(b) task *T1* overruns but it still delivers the output at the same point in time (end of first period) as in execution A. The priority impact can be seen during the second execution period where *T2* starts first as it is ready due its short period, after which *T1* becomes ready but it cannot preempt *T2*. *T1* waits until the current execution of *T2* terminates before it starts running. Finally, *T3* becomes ready and starts executing, however once *T2* becomes ready again it preempts it due to its higher priority. *T3* resumes early in the third period, but it gets preempted by *T1* this time.

In general, predictability should reduce the need for testing compared to the standard approach since the externally-observed behavior will not change. Limited observability ensures system predictability in case no fault occurs. In case of faulty behaviors, e.g. a deadline overrun, the system adapts its runtime to amortize the faults while maintaining the limited observability, so that it might end up being predictable despite the presence of faults (Section 4.2).

4 Formal Basis of our Framework

This section introduces a formal description of the systems that can be modeled and analyzed in our framework.

4.1 System Specification

The system application we consider consists of a set of components $\{\mathcal{T}_1, .., \mathcal{T}_m\}$, each of which is a set of periodic tasks $\mathcal{T}_j = \{T_1^j, .., T_k^j\}$. Similarly, the system platform is a set $\{C_1, .., C_q\}$ of homogeneous cores, each of which (C_j) is assigned to one component (task set) $\mathcal{T}_j$.

The tasks of a given component $\mathcal{T}_j$ will be scheduled by a real-time operating system according to a scheduling function *Sched* given by:

$$Sched^j : \mathcal{T}_j \times \mathcal{T}_j \times \mathbb{R}_{\geq 0} \rightarrow \mathcal{T}_j$$

where $\mathbb{R}_{\geq 0}$ is the time domain. The function compares 2 tasks at a given time instant and returns the task having priority at that time point. It is described abstractly in order to be able to model both static and dynamic priority scheduling algorithms. The scheduling policies we have modeled in this framework are: Earliest-Deadline First (EDF), Dynamic Deadline-Monotonic (DMS) and Simplified Zero-Slack (ZS). Throughout this paper we will focus mainly on the ZS

policy as it enables to deal with faults and mixed-criticality, but DMS and EDF are manipulated in the same way.

Since each component $\mathcal{T}_j$ can use two execution modes (*normal* and *critical*), we distinguish two scheduling functions, $Sched_n^j$ and $Sched_c^j$, to be applied in normal and critical modes respectively. For the sake of simplicity, since components behave in the same way and cores are identical (modular design) we will only focus on one component $\mathcal{T} = \{T_1, .., T_n\}$ running on one core C. Accordingly, all the exponent notations ($-^j$) related to the choice of a component will be omitted. Among the tasks of a component, we identify $\mathcal{T}_c \subseteq \mathcal{T}$ to be the set of critical tasks.

Formally, each task is given by a period p, a budget b, a deadline d and a criticality level c. Since it is not needed to distinguish between the period and deadline of tasks in our context (assumed to have the same values), we omit deadlines and just keep the period length which must be longer than the required budget. The criticality level does not have an effect during regular operation, but can be used in an overload or fault condition to prioritize tasks for load-shedding. Following the window notion (Section 3) allowing for dynamic runtime, the task's attributes can vary from one period to another. Basically, a window represents one release of the task execution. Formally, the notation $w_i^j = (p, b, c)$ states the period, budget and criticality level of task T_i for the j^{th} window. Accordingly, we represent each task T_i by a sequence of windows $W_i = (w_i^1, w_i^2, ...)$ describing its runtime. We denote the set of all potential windows by $\mathcal{W}$, and assume that there is no gap between windows i.e., the deadline of a window will be the baseline of the next window. A task execution can be scheduled anywhere within a window. The operating system stores a static lookup table containing all the possible configurations (windows) for each task. To simplify notation, we use $w.x$ to refer to the attribute x within the window structure w.

Communication between tasks is aligned to the baselines and deadlines by the operating system in a way that it is entirely transparent to the task. No matter where the task is executing within its window, it will see an identical "snapshot" taken at the baseline of the values in shared memory written by other tasks. Any changes made after the task's baseline will not be visible within the current window. Similarly, any values written to shared memory by the task will not become visible to other tasks until the deadline of the writing task. The writes to the same memory location are applied in the order of the corresponding task deadlines rather than when the data was actually written from the task point of view. If a deadline and a baseline are coincident (i.e. a write and a read of the same data in shared memory), the write is to happen before the read.

4.2 System Semantics

To simplify the semantics, we only focus on one component $\mathcal{T} = \{T_1, .., T_n\}$ running on one core since the rest of the system behaves in the same way using the same execution rules, assuming there is no inter-component dependency. First we introduce the following variables:

- clk is a clock variable to track the global time.
- $Mode = \{Normal, Critical\}$ is the set of execution modes, and $mode$ is a variable to store the current execution mode of the system.
- $Status = \{Waiting, Running, Overrun, Done\}$ is the set of status values, whereas $status = [1..n]$ is an array used to store the current status of each task. We assume that all the status values are assigned by the operating system (assigned to the tasks), except status $Done$ which is triggered by the task itself once its execution is over.
- $curW$ is an array variable storing the current window of each task.
- $Rbudget$ is an array variable used to track the remaining budget of each task during runtime.
- $curTime$ is an array clock variable to store the start time of each period for each task. It will be used as a baseline to track when a period expires.
- $curET$ is an array variable used to measure the CPU time acquired by each task during each execution. Each variable will be set to the current time when the corresponding task is scheduled.
- $ExcessT$ is an array variable used to store the time point when a task starts overrunning its deadline.

The semantics is given in terms of a timed transition system (TTS) $\langle S, s_0, \rightarrow \rangle$ [13] where S is a set of states, s_0 is the initial state and "$\rightarrow$" is the transition relation. Formally, $S = \mathbb{R}_{\geq 0} \times Mode \times Status^n \times \mathcal{W}^n \times \mathbb{R}^n_{\geq 0} \times \mathbb{R}^n_{\geq 0} \times \mathbb{R}^n_{\geq 0}$, $s_0 = (0, Normal, \forall i\ status(T_i) = Waiting, \forall i\ curW(T_i) = w_i^1, \forall i\ curTime(T_i) = 0, \forall i\ Rbudget(T_i) = w_i^1.b, \forall i\ curET(T_i) = 0, \forall i\ ExcessT(T_i) = 0)$ whereas transitions are given by rules **Release1**, **Release2**, **Normal**, **Critical**, **Nrml2Crit**, **Crit2Nrml** and **Overrun** . We use notation [ ] to access to the internal structure and values of each state. Updating a field, e.g a task status, within a state s leads to a new state s' having the same values as s except for the modified field. Initially, the system is in normal mode and tasks are waiting to be scheduled. All clock variables are set to 0.

Scheduling. According to the current execution mode, the operating system schedules one of the ready tasks (having status $Waiting$) using the appropriate scheduling function, $Sched_n$ or $Sched_c$.

$$\textbf{Normal}: \frac{\forall\ s \in S, T_i \in \mathcal{T} \mid s.mode = Normal, s.status(T_i) = Waiting \wedge \forall T_j \in \mathcal{T} \quad Sched_n(T_i, T_j, s.clk) = T_i}{s \rightarrow s[status(T_i) := Running, curET(T_i) := clk]}$$

$$\textbf{Critical}: \frac{\forall\ s \in S, T_i \in \mathcal{T}_c \mid s.mode = Critical, s.status(T_i) = Waiting \wedge \forall T_j \in \mathcal{T}_c \quad Sched_c(T_i, T_j, s.clk) = T_i}{s \rightarrow s[status(T_i) := Running, curET(T_i) := clk]}$$

In both rules, the task to be scheduled (T_i) potentially preempts another task T_j (if T_j is already running). If so, the status of T_j needs to be updated to $Waiting$, and its actual remaining budget needs to be recalculated using the

previous value ($Rbudget(T_j)$) and the CPU time used from its last scheduling time point until the preemption, i.e. $Rbudget(T_j) = Rbudget(T_j) - (s.clk - curET(T_j))$. We don't embed these statements in the scheduling rules just to avoid duplicating the scheduling rules for two cases: 1) CPU is free; 2) there is a lower priority task T_j currently running. The status of T_i is updated to *Running* and the current time is recorded ($curET(T_i) := clk$) to keep track of how long T_i has been running.

Mode Switches. When the zero-slack instant is reached, the execution mode switches to *Critical* and only critical tasks are allowed to execute. Once all critical tasks are satisfied in their current windows, the mode switches back to *Normal*.

$$\textbf{Nrml2Crit}: \frac{\forall\, s \in S, T_i \in \mathcal{T}_c \mid s.mode = Normal, T_i s.status(T_i) = Waiting \\ \wedge\, s.curW(T_i).p + s.curTime(T_i) - s.clk \leq s.Rbudget(T_i)}{s \rightarrow s[mode := Critical]}$$

$$\textbf{Crit2Nrml}: \frac{\forall\, s \in S, T_i \in \mathcal{T}_c \mid s.mode = Critical, s.status(T_i) \neq Waiting}{s \rightarrow s[mode := Normal]}$$

When the remaining time of the current window, calculated from the baseline $curTime$, is less than or equal to the remaining budget of a critical task the mode switches to *Critical*. One can remark that we can predict the mode change only for the current execution windows. When all critical tasks ($\mathcal{T}_c$) are either terminated or running, the execution mode switches back to *Normal* where tasks will be scheduled accordingly using rules **Normal** and **Critical**.

Fault Handling. There are three possible failure modes that the system can experience: task failure, budget violation and deadline violation.

Task Failure. Task failure is the most obvious failure mode: a task either incorrectly calculates its outputs or suffers other catastrophic failures e.g., unexpected termination. Handling this type of faults is beyond the scope of this work.

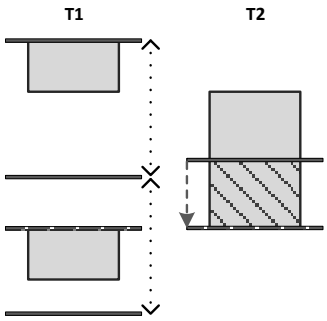
Budget Violation. The execution budget defined in the configuration of the execution window represents the maximum amount of time that a task consumes in correct operation. Exceeding this budget is considered a fault of the task itself (incorrect operation) or of the system integrators (incorrect execution budget).

When a violation is detected, the operating system can take one of two possible actions: either the budget violation is ignored or the offending task is terminated. Ignoring the violation is possible as the task set should not be designed to use 100% of the available processor time. Ideally, the overrun will eventually be absorbed by the available slack (idle ticks) and the system will return to normal. Terminating the task ensures that the assumptions made by the operating system continue to hold, however the behavior of the system as a whole could become wildly incorrect.

Deadline Overrun. A task *must* finish its execution before its deadline. Violating the deadline is considered a fault, but not necessarily of the task in question: it is possible that the task missed its deadline due to other tasks misbehaving, or due to an incorrect configuration.

Deadline violation is more serious than budget violation and some compensatory actions must be taken. The simplest option is to terminate the offending task. The other option is to delay the deadline until the task has finished executing. In this case, the current deadline and the next baseline are delayed until the task enters a completed state. The next deadline of the task experiencing an overrun is not delayed, rather, the next window is shortened. This guarantees that the overrunning task still sees the same snapshot of shared data at its baseline. The operating system achieves this by delaying any task's baseline and deadline occurring during the overrun, except for the overrunning task, and applies them in the same order in which they would normally occur after the overrun has completed. The next windows of these tasks are also shortened in the same way as for the overrunning task. In Figure 3, task *T2* violates its deadline and causes a delay (dashed arrow). The baseline of task *T1* occuring within this overrun period is delayed as well. The deadline of *T1* remains at the same position relative to the original deadline (dotted arrow), with the effective size of the window reduced appropriately.

Fig. 3: Overrun delays



By shortening subsequent windows, it is possible that a cascade of deadline violations will be created. However, as illustrated in Figure 4, the slack present in the system should ideally allow the overruns to be absorbed and allow the system to return to normal operation. For this mechanism to work, the system cannot operate under full load. A safety margin must be included in the system design, with more processor capacity available than is required by the tasks under normal operation. The size of this safety margin would depend on the allowable overrun.

Having all tasks meet their deadlines is a sufficient condition for predictability, as shown by PharOS [6]. However, it is not a necessary one: if the overrun can be absorbed such that the final outputs still occur when they are supposed to, predictability can be preserved despite schedulability being violated due to the overrun.

$$\textbf{Overrun}: \frac{\begin{array}{c}\forall\ s \in S, T_i \in \mathcal{T} \mid s.status(T_i) \in \{Waiting, Running\} \\ \wedge\ s.curTime(T_i) \geq s.curW(T_i).p\end{array}}{s \rightarrow s[status(T_i) := Overrun, \forall T_j \in \mathcal{T}\ \ ExcessT(T_j) := clk]}$$

Rule **Overrun** describes when an overrun occurs. Basically, when a task reaches the end of its current window $curW(T_i).p$ (which coincides with the

deadline) before completion an (effective) overrun case is declared. The current time clk is stored in variable $ExcessT()$ of task T_i in order to calculate the overrun duration once the task execution is done. Moreover, in order to postpone the deadlines and baselines occurring during the overrun, we also communicate the overrun start to the other running tasks ($\forall T_j \in \mathcal{T}\ ExcessT(T_j) := clk$) so that the current window of each task will be delayed as well (fake overrun) with the same duration as for the overrunning task.

Through the release of a new window, the configuration of a task will be updated according to rule **Release1**.

$$\textbf{Release1}: \frac{\begin{array}{c}\forall\ s \in S, T_i \in \mathcal{T} \mid s.status(T_i) = Done, s.ExcessT(T_i) = 0 \\ \wedge\ s.curW(T_i) = w_i^x, s.curTime(T_i) = s.curW(T_i).p \\ \wedge\ \forall T_j \in \mathcal{T}\ s.status(T_j) \neq Overrun\end{array}}{\begin{array}{l}s \rightarrow s[status(T_i) := Waiting, curTime(T_i) := clk, curET(T_i) := 0, \\ curW(T_i) := w_i^{x+1}, Rbudget(T_i) := w_i^{x+1}.b]\end{array}}$$

$$\textbf{Release2}: \frac{\begin{array}{c}\forall\ s \in S, T_i \in \mathcal{T} \mid s.status(T_i) = Done, s.ExcessT(T_i) > 0 \\ \wedge\ s.curW(T_i) = w_i^x\end{array}}{\begin{array}{l}s \rightarrow s[curW(T_i) := w_i^{x+1}, curW(T_i).p := w_i^{x+1}.p - (clk - ExcessT(T_i)), \\ curTime(T_i) := clk, curET(T_i) := 0, Rbudget(T_i) := w_i^{x+1}.b, \\ status(T_i) := Waiting]\end{array}}$$

Rule **Release1** describes the expiry of a window w_i^x and the release of a new window w_i^{x+1} of a task T_i successfully executed during the previous window w_i^x, i.e. without missing its deadline ($s.curTime(T_i) = s.curW(T_i).p$) and none of the other tasks is currently overrunning its own deadline. The status as well as the variables we introduced to monitor the task execution are reinitialized accordingly. When the effective overrun of a task is over, all the other postponed tasks will be released with their new windows. Rule **Release2** describes how the new window length of each task will be reduced with any overrun delay from the previous window.

In both rules **Release1** and **Release2**, one can remark that a task cannot release a new window if any other task is overrunning. This ensures that all deadlines and baselines occurring during an overrun are postponed until the overrun terminates.

5 Verification

As long as the tasks complete before their deadlines, the system described in Section 4 is predictable. This includes cases where execution budget violations occur. Thus, a combination of task set and fault model which is free from deadline violations is said to be schedulable and, by design, predictable [6].

The more interesting cases are those where the task set is not always schedulable. However, the task set could still be predictable: deadline violations could

be absorbed by utilizing idle processor time or output values of certain tasks could be ignored in the given execution mode.

In order to be able to inject faults intentionally and verify the system's predictability accordingly, we extend the system model described in Section 4 with the following:

- For each task window $w_i^j = (p, b, c)$, by how much (overrun $\mathbf{o} \in \mathbb{R}_{\geq 0}$) the task will exceed its budget should it enter a fault state, i.e. $w_i^j = (p, b, c, o)$.
- We also allow for a description to be added to each task to specify how the input values are used to compute the outputs.

The final number of faults that can be absorbed depends on the length of overruns and the free time slot (λ) of the processor, i.e., $\sum_i \sum_j \ w_i^j.o \leq \lambda$. The analysis of predictability is performed by exhaustively checking that for each set of valid timestamped inputs, the system always produces the same set of timestamped outputs — with all possible combinations of faults allowed by the system configuration. The verification process either concludes that the system is predictable, or provides a counterexample in which the predictability is not preserved. The predictability can be analyzed using 2 approaches: direct and indirect.

5.1 Direct Approach to Analyze Predictability

The most direct approach for checking the predictability considers all possible sets of timestamped inputs, combines them with all possible failure modes, and checks that the same set of timestamped outputs is always produced for each respective set of inputs. This is not as daunting a task as it may first seem. The tasks are periodic and the number of configurations is finite. The system will eventually start to repeat previously-explored states at which point the search can stop. Additionally, as the task behavior is assumed to be correct, the range of possible inputs does not need to be fully examined. However, this approach still consumes much time and resources as well as being not cheaply implementable using Spin.

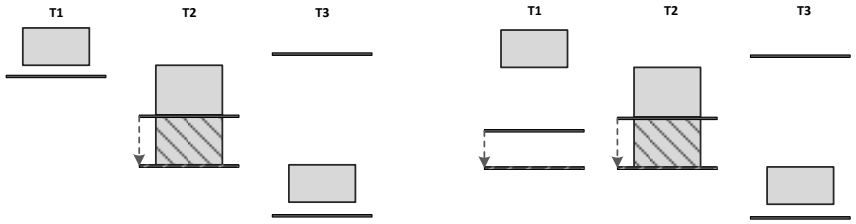
5.2 Indirect Approach to Analyze Predictability

As mentioned earlier, the system is predictable by design in no-fault cases. Therefore, to verify predictability it is necessary to analyze which behaviors the system exhibits when a fault occurs, and which of those behaviors will result in a violation of the predictability. Our approach is based on the following three observations and steps:

Inputs only happen at read times. Each input/output has two parameters: a data value and time. The data value is beyond the influence of the system and is assumed to always be correct and valid. The timing of input arrival is technically beyond the control of the system as well; however, the system controls

Fig. 4: Absorbing a deadline overrun

(a). Predictability preserved **(b). Predictability not preserved**



when it reads in the value of the input parameter. This is a reduction of many possible cases to one: the input can arrive anywhere between reads, but all these scenarios are effectively equivalent to the input arriving at the read time. If the system is predictable with inputs that are timestamped at the read time, it is also predictable if the actual arrival times are used.

Only monitor "edge" task timing. Every task set is assumed to have a task for reading input and another for writing output; these two tasks are called *edge* tasks. The edge tasks perform the input and output at their baselines and deadlines, respectively. If either the input baseline or the output baseline has to be moved to satisfy the data flow requirements, predictability is lost. Consider, e.g., Figure 4 where *T1* is an input task, *T3* is an output task and the processing task *T2* is overrunning. The overrun can be absorbed, thus neither the baseline of the input task nor the deadline of the output task will be delayed. In Figure 4.(a), even though the baseline of *T2* gets delayed the input and output times remain unchanged, so the predictability is not violated. However, in Figure 4.(b) the predictability is violated as *T1* reads its inputs at a later time, i.e., the new baseline of *T1* occurring during the overrun of *T2* is delayed by the dashed arrow until *T2* completes.

To capture edge tasks, we introduce the following:

- We define each data $x = (v, a)$ by a value v and an arrival time (stamp) a.
- The inputs and outputs of each task T, consisting each of a set of data, are given by $T.input$ and $T.input$ respectively.
- We identify the tasks being as edges with T through function $Edge(T) = \{T_i \mid T.output \cap T_i.input \neq \emptyset\}$.

In order to maintain the predictability guaranteed by design, one needs to maintain the same relative order between the baseline/deadline of the edge tasks. In some cases, e.g. non critical settings, this can be achieved by just delaying the deadline of the faulty task as well as the baseline of its edge tasks.

Flag outputs of forcibly terminated tasks. In the most permissive operating mode, where execution budget violations are ignored and deadline violations result in delays, checking data values is not necessary as the system guarantees

correct data flow by eschewing timing enforcement. However, in a more restrictive mode that involves forced task termination, e.g hard critical systems or due to the non availability of processor free slots, the data values do need to be checked. This is done by verifying whether a task that was terminated by the operating system is involved in calculating a data value. If a terminated task was involved anywhere in the chain, the output is no longer reliable because it is not guaranteed whether the task completed the calculation successfully before being terminated. To this end, we introduce a new status *Forced* which will be assigned to any task once its misses its deadline. The termination rule is similar to rule **Overrun**, which is applicable when $s.curTime(T) \geq s.curW(T).p$ (deadline missed) and updates the status of T in the following: $status(T) := Forced$.

To check the involvement of the output of a terminated task in the calculation of other tasks data, we add a "tainted" flag to data values as they are passed between tasks. The flag is set when a task that is supposed to write a given value is forcefully terminated, and it is propagated by all other data-dependent tasks. If a tainted value is observed among the output, it results in a violation of the predictability. To flag the output of tasks, we introduce the following:

- The involvement of data x in the calculation of the output of a task T is given by predicate $Involved(x, T)$. Such a predicate can easily be derived from the functional description of tasks.
- To taint the output of a task T once it is forcibly terminated, we introduce $Taint(T.output)$ as a predicate. $Taint()$ is initially initialized to false, i.e. $\forall T,\ Taint(T.output) = false$.
- The flag function $Flag(T)$, used to propagate the taint from a forcibly terminated task T to the output of its (descendent) edge tasks, is given by:

$$Flag(T) = \begin{cases} Taint(T.output) := true & \text{If } Edge(T) = \emptyset \\ & \text{Or } (\forall T_i \in Edge(T), \forall x \in T.output\ \neg Involved(x, T_i)) \\ (Taint(T.output) := true) \wedge (Flag(T_i \mid \forall T_i \in Edge(T))) & \text{Otherwise} \end{cases}$$

The predictability violation can then simply be checked through the existence of tainted outputs. Using model checking, we quantify over all system states S by exploring all the tasks $\mathcal{T}$. Formally, the predictability is preserved if for any non forcibly terminated task the outputs are not tainted:

$$\forall s \in S\ \forall T \in \mathcal{T},\ s.status(T) \neq Forced \Rightarrow Taint(T.output) = false$$

6 Implementation and Application

Our system description has been modeled using the Promela language, allowing for verification of predictability of arbitrary task sets. The usefulness of the verifier lies in its ability to perform an exhaustive search. The verification exhaustively explores all possible executions and checks for predictability violations. If a system is not predictable, a counterexample demonstrating the violation is provided. A graphical front end was created to simplify task set definition and result parsing. This utility generates the required Promela model based on the

parameters supplied by the user and pre-written skeleton code. It also parses the counterexample trail files produced by the verifier into a graphical representation of the task-to-processor assignment over time.

6.1 Spin Modifications

The counterexample trail files produced normally consist of the states and transitions visited by the verifier leading up to a state violating the predictability. This allows a trail file to be "replayed" by Spin. Within the trail file, the states are identified by their state numbers—a property internal to Spin that cannot be relied upon or (easily) determined given the source Promela model. The counterexample file becomes meaningless, as the state numbers cannot be mapped to actual behavior of the system. Even if the trail file is replayed and the source Promela statements displayed, they are meaningless to the user since the Promela code is automatically generated.

To overcome this, we have extended Spin to support state annotations using the existing Promela label syntax. Promela supports C-style labels (`label:`), used as targets for `goto` statements as well as identifying special states (`end`, `progress`, etc.). An additional special label type, `annotation`, is added. Any text included in the label is attached to the state and propagated to the generated verifier. The verifier, in turn, includes the text of these states in the trail files it produces. Annotation labels are merged and lifted when state merging happens, or within constructs that only produce one state (e.g., `dstep`) from a block of Promela code. A list of all annotations that fall within a state is included in the trail file in these cases. Meaningful data can now be extracted from the trail file.

6.2 Promela Model

The Promela model is divided into three parts: a set of tasks, the scheduler, and the environment. The task code is entirely generated by the interface, based on the user parameters. The scheduler and environment are mostly constant between different task sets, with different sections enabled and disabled via preprocessor macros depending on user inputs.

Each task is comprised of five blocks of Promela code, which are invoked at the appropriate times: initialization, baseline, deadline, execution tick and forced termination. Initialization code allows the task to push values to the global descriptor table. Baseline and deadline code are called at each baseline and deadline respectively. Execution tick ("run") code is called once per every scheduling tick that the task is assigned processor time. Finally, forced termination code is called when the task needs to be terminated unexpectedly. Regular termination, if needed, is supposed to be handled by the run code.

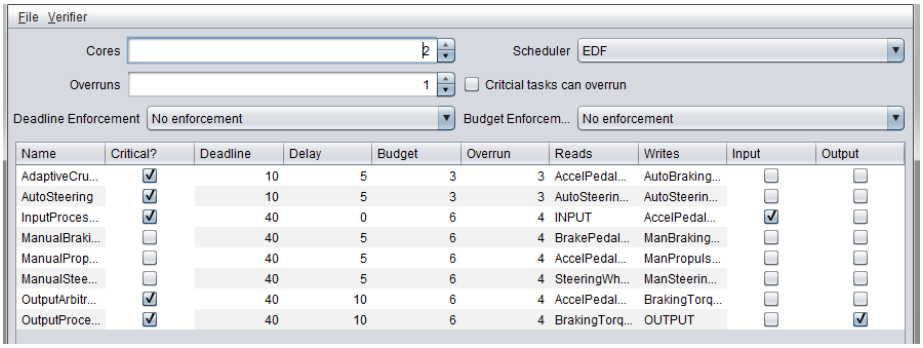
The scheduler code is entirely deterministic—all non-deterministic choices happen within the tasks, as it is assumed that the scheduler is fault-free. This code is executed once per tick, and is responsible for updating task timing information and choosing which tasks get assigned processor time.

The environment consists mainly of the scheduler loop. It calls the task-specific code and the scheduler code at the appropriate times. The environment and scheduler together are meant to represent the operating system's role.

6.3 Interface

A cross-platform Java GUI (Figure 6) was created to simplify creating task sets and modifying parameters. The interface allows users to enter task-specific code, as well as general parameters such as number of cores and enforcement modes. It then generates the final Promela model, automating certain repetitive tasks (e.g., inserting a task identifier in the proper places) that are too complex for the C pre-processor normally used by Spin for such purposes. Finally, if a trail file is produced, the GUI generates a visualization showing how the defined tasks were scheduled and when the predictability violation occurred. For further details regarding the implementation, we refer readers to [14].

Fig. 6: The Java graphical interface.



6.4 Case Study: Active Safety Demo

To show the applicability of our framework, we have analyzed a realistic example from the automotive domain. The most relevant parts of the task set description are given in Table 1. Various system configurations have been analyzed—different numbers of cores and allowable faults, along with different scheduling algorithms and enforcement modes. The size of the state space and therefore both the execution time and memory requirements increase as the number of permitted faults increases, because each fault represents a non-deterministic choice that needs to be made once per scheduling tick per task, causing a branch in the search tree. The search tree becomes very broad with many faults, but the depth remains tractable.

All optimizations performed by Spin when generating the verifier (e.g., state merging) were enabled. The performance of the verifier on some representative configurations can be seen in Figure 7. The results shown are with critical tasks permitted to fail. Each scheduling algorithm produced broadly similar results, so only the results for EDF are shown. The performance of both the verifier

Table 1: Parameters and data flow of the active safety demo task set, (C = Criticality, W = Window Size, B = Budget, O = Overrun amount, on fault). See [14] for descriptions of the values.

Task	C	W	B	O	Values Read	Values Written
AdaptiveCruise	Hi	10	3	3	APP BPP CO CSS TD VS	ABT APT
AutoSteering	Hi	10	3	3	ASO LMP SWA	AST
InputProcessing	Hi	40	6	4	*(Environment)*	APP ASO BPP CO CSS LMP SWA SWT TD VS
ManualBraking	Lo	40	6	4	BPP	MBT
ManualPropulsion	Lo	40	6	4	APP	MPT
ManualSteering	Lo	40	6	4	SWA	MST
OutputArbitration	Hi	40	6	4	APP ABT APT ASO AST BPP CO MBT MPT MST SWT	BT COI PT SOI ST
OutputProcessing	Hi	40	6	4	BT COI PT SOI ST	*(Environment)*

itself and the visualization generation is very good on a modern system (Core i7, 16GB RAM), the entire process generally taking less than a second for a realistic number of faults. Further analysis results are available in [14].

Fig. 7: Analysis results of the case study.

Cores	Faults	States	Time	Predictable?
1	0	2178	0.010s	No
2	0	2231	0.002s	Yes
2	1	19468	0.025s	Yes
2	4	44369	0.052s	No
4	4	632906	0.682s	Yes
4	32	20055317	22.9s	Yes

7 Related Work

In the literature, several frameworks for the predictability analysis of real-time systems have been proposed [10, 9, 8, 17, 5]. However, only few proposals consider multicore platforms and dynamic attributes of tasks.

The authors of [8] presented a model-based architectural approach for improving the predictability of real-time systems. This approach is component-based and utilizes automated analysis of task and communication architectures. The authors generate a runtime executive that can be analyzed using the MetaH language and the underlying toolset. Such a work does not deal with multicore or criticality.

Garousi *et al* introduced a predictability analysis approach [10], for real-time systems, relying on the control flow analysis of the `UML 2.0` sequence diagrams as well as the consideration of the timing and distribution information. The analysis includes resource usage, load forecasting/balancing and dynamic dependencies. Our work differs because it supports dynamic runtime and fault handling.

The authors of [3] introduced a compositional analysis enabling predictable deployment of component-based systems running on heterogeneous multi pro-

cessors. The system is a composition of software and hardware models according to a specific operational semantics. Such a framework is a simulation-based analysis, thus it cannot be used as a rigorous analysis means for critical systems.

The authors of [16] defined a predictable execution model for COTS (commercial -off-the-shelf) based embedded systems. The goal is to control the use of each resource in such a way that it does not exceed its saturation limit. However, such a claim cannot always be maintained because of the non-determinism in the behavior of tasks and their environment.

The authors of [4] introduced a predictability analysis framework for real time systems given by a set of independent components running on a single core platform. Data flow is abstracted using dependability whereas predictability is compositionally analyzed through schedulability as a sufficient condition. However, simplifying the architecture to obtain a compositional analysis might not be practical for modern COTS-based embedded systems.

In [15], the authors introduce data flow graphs as a scheduling means for data flow within single core systems so that liveness and boundness are guaranteed. The schedulability analysis of data flow is then performed by translating data flow graphs to graph-based real-time tasks. A study of the applicability of such a framework for multicore systems having dynamic runtime is very interesting. In a similar way, the authors of [2] introduce a model of data flow computation to overcome the restrictions of classical data flow graphs by allowing dynamic changes during runtime. The dynamism of data flow graph is expressed by 2 parameters: the number of data required (rate) for each flow and the activation/deactivation of communications between the functional units. Compared to that, our framework considers a static topology of the data flow graph encapsulated within the dynamic runtime of tasks however the data flow timeliness can vary in accordance with faults (overruns).

8 Conclusion

In this paper, we have introduced a formal framework and model checking based approach for the predictability analysis of mixed criticality task sets running on multicore platforms. The framework supports window scheduling and dynamic tasks bahavior, and allows for failures to be handled at runtime. We formulated a system description and modeled it in Promela. A GUI was implemented to increase ease of use and Spin was extended to support the generation of visualizations. The analysis results for a realistic example are encouraging and suggest that the approach might scale to industrial settings.

We greatly simplify the analysis by observing that only monitoring "edge" tasks for delays and checking outputs for values tainted by terminated tasks is needed. Interesting future work would be to model the data flow separately from tasks behavior, in similar way to [2], to make our framework more flexible.

Acknowledgment

This work is supported by the Natural Sciences and Engineering Research Council of Canada, as part of the NECSIS Automotive Research Partnership with General Motors, IBM and Malina Software Corp.

References

1. ISO 26262-1:2011 Road vehicles–Functional safety. Technical report, ISO, 2011.
2. V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *EMSOFT '13*, pages 3:1–3:10. IEEE Press, 2013.
3. E. Bondarev, M. Chaudron, and P. de With. Compositional performance analysis of component-based systems on heterogeneous multiprocessor platforms. In *SEAA '06.*, pages 81–91, Aug 2006.
4. A. Boudjadar, J. Dingel, B. Madzar, and J. H. Kim. Compositional predictability analysis of mixed critical real time systems. In *FTSCS'15*, volume 596 of *CCIS*, pages 69–84. Springer, 2015.
5. A. Boudjadar, J. H. Kim, K. G. Larsen, and U. Nyman. Compositional schedulability analysis of an avionics system using UPPAAL. In *Proceedings of ICAASE'14*, pages 140–147, 2014.
6. D. Chabrol, C. Aussagues, and V. David. A spatial and temporal partitioning approach for dependable automotive systems. In *IEEE Conference on Emerging Technologies Factory Automation*, pages 1–8, 2009.
7. D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *RTSS'09*, pages 291–300, 2009.
8. P. Feiler, B. Lewis, and S. Vestal. Improving predictability in embedded real-time systems. Technical Report CMU/SEI-2000-SR-011, Dec 2000.
9. J. Fredriksson. *Improving Predictability and Resource Utilization in Component-Based Embedded Real-Time Systems.* PhD thesis, Mälardalen University, 2008.
10. V. Garousi, L. C. Briand, and Y. Labiche. a unified approach for predictability analysis of real-time systems using UML-based control flow information. In *MoDELS*, volume LNCS 3844, 2005.
11. T. A. Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366:3727–3736, 2008.
12. T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Embedded Software*, pages 166–184. Springer, 2001.
13. T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 226–251. Springer-Verlag, 1992.
14. B. Madzar. Modelling and verification of predictable data flow in real-time systems, M. Sc thesis. Queen's University Canada, 2015.
15. M. Mohaqeqi, J. Abdullah, and W. Yi. Modeling and analysis of data flow graphs using the digraph real-time task model. In *21st International Conference on Reliable Software Technologies*, volume 9695 of *LNCS*, pages 15–29. Springer, 2016.
16. R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *RTAS'11.*
17. S. Yau and X. Zhou. schedulability in model-based software development for distributed real-time systems. In *WORDS'02*, pages 45–52, 2002.