Schedulability in Mixed-criticality Systems

Rany Kahil

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Rany Kahil

Thèse dirigée par **Saddek Bensalem**

préparée au sein du **Laboratoire Verimag**
dans **l'École Doctorale Mathématique, Sciences et Technologies de l'information (MSTII)**

Schedulability in Mixed-criticality Systems

Thèse soutenue publiquement le **26/06/2019**,
devant le jury composé de :

Prof. Thao Dang
Directeur de Recherche of the CNRS, Présidente
Prof. Sanjoy Baruah
Washington University, Rapporteur
Prof. Yamine Ait Ameur
Universités INPT-ENSEEIHT/IRIT, Rapporteur
Prof. Radu Grosu
Vienna University of Technology, Examinateur
Prof. Samarjit Chakraborty
Technical University of Munich, Examinateur
Prof. Saddek Bensalem
Universite Grenoble Alpes, Directeur de thèse

Acknowledgements

Apart from all my efforts over my years as a PhD student, the completion of this thesis would not have been possible without the guidance and encouragement of many people. Most of all, I would like to express my greatest gratitude to my thesis advisors, Saddek Bensalem and Petro Poplavko.

Saddek has always been there to provide advice and direction. In his special way, he was able to keep me motivated without making me feel pressured. He was very understanding and supportive regarding my work, and also on a personal level. He really is an exceptional advisor.

I was lucky to have Petro as my co-advisor, he always had new ideas and his feedback was invaluable. I thank him for all the help he gave me, all our late time meetings, and all our long discussions. My thesis would not have been the same without him.

I am grateful to Marius and Jacques for their help when I needed it, and to all people from Verimag, who have always been friendly and created a comfortable work environment.

I would also like to thank my family. My mother, father and sister who encouraged me to make the decision of pursuing the Doctorate's degree. And a special thanks to my wonderful wife who is always beside me despite the distance. Her endless support and love have kept me harmonious throughout the period of my studies.

Abstract

Real-time safety-critical systems must complete their tasks within a given time limit. Failure to successfully perform their operations, or missing a deadline, can have severe consequences such as destruction of property and/or loss of life. Examples of such systems include automotive systems, drones and avionics among others. Safety guarantees must be provided before these systems can be deemed usable. This is usually done through certification performed by a third party, a certification authority. Safety evaluation and certification are complicated and costly even for smaller systems.

One answer to these difficulties is the isolation of the critical functionality. Executing tasks of different criticalities on separate platforms prevents non-critical tasks from interfering with critical ones, provides a higher guaranty of safety and simplifies the certification process limiting it to only the critical functions. But this separation, in turn, introduces undesirable results portrayed by an inefficient resource utilization, an increase in the cost, weight, size and energy consumption which can put a system in a competitive disadvantage.

To overcome the drawbacks of isolation, Mixed Criticality (MC) systems can be used. These systems allow functionalities with different criticalities to execute on the same platform. In 2007, Vestal proposed a model to represent MC-systems where tasks have multiple Worst Case Execution Times (WCETs), one for each criticality level. In addition, correctness conditions for scheduling policies were formally defined, allowing lower criticality jobs to miss deadlines or be even dropped in cases of failure or emergency situations. The introduction of multiple WCETs and different conditions for correctness increased the difficulty of the scheduling problem for MC-systems. Conventional scheduling policies and schedulability tests proved inadequate and the need for new algorithms arose. Since then, a lot of work has been done in this field.

In this thesis, we contribute to the study of schedulability in MC-systems. The workload of a system is represented as a set of jobs that can describe the execution over the hyperperiod of tasks or over a duration in time. This model allows us to study the viability of simulation-based correctness tests in MC-systems. We show that simulation tests can still be used in mixed-criticality systems, but in this case, the schedulability of the worst case scenario is no longer sufficient to guarantee the schedulability of the system even for the fixed priority scheduling case. We show that scheduling policies are not predictable in general, and define the concept of weak-predictability for MC-systems. We prove that a specific class of fixed priority policies are weakly predictable and propose two

simulation-based correctness tests that work for weakly-predictable policies. We also demonstrate that contrary to what was believed, testing for correctness can not be done only through a linear number of preemptions.

The majority of the related work focuses on systems of two criticality levels due to the difficulty of the problem. But for automotive and airborne systems, industrial standards define four or five criticality levels, which motivated us to propose a scheduling algorithm that schedules mixed-criticality systems with theoretically any number of criticality levels. We show experimentally that it has higher success rates compared to the state of the art.

We illustrate how our scheduling algorithm, or any algorithm that generates a single time-triggered table for each criticality mode, can be used as a recovery strategy to ensure the safety of the system in case of certain failures. To do so, we representing the system as a set of synchronized timed-automata components, where the scheduling algorithm is modeled as a timed-automaton that acts as a part of the Fault Detection Isolation and Recovery (FDIR) component in the system.

Finally, we propose a high level concurrency language and a model for designing an MC-system with coarse grained multi-core interference.

Rèsumè

Les systèmes temps-réel critiques doivent exécuter leurs tâches dans les délais impartis. En cas de défaillance, des événements peuvent avoir des catastrophes économiques. Dans certain cas une atteinte à des vies humaines. Des classifications des défaillances par rapport aux niveaux des risques encourus ont été établies, en particulier dans les domaines des transports aéronautique et automobile. Des niveaux de criticité sont attribués aux différentes fonctions des systèmes suivant les risques encourus lors d'une défaillance et des probabilités d'apparition de celles-ci. Ces différents niveaux de criticité influencent les choix d'architecture logicielle et matérielle ainsi que le type de composants utilisés pour sa réalisation. Les systèmes temps-réels modernes ont tendance à intégrer sur une même plateforme de calcul plusieurs applications avec différents niveaux de criticité. Cette intégration est nécessaire pour des systèmes modernes comme par exemple les drones (UAV) afin de réduire le coût, le poids et la consommation d'énergie. Malheureusement, elle conduit à des difficultés importantes lors de leurs conceptions. En plus, ces systèmes doivent être certifiés en prenant en compte ces différents niveaux de criticités. Il est bien connu que le problème d'ordonnancement des systèmes avec différents niveaux de criticités représente un des plus grand défi dans le domaine de systèmes temps-réel. Les techniques traditionnelles proposent comme solution l'isolation complète entre les niveaux de criticité ou bien une certification globale au plus haut niveau. Malheureusement, une telle solution conduit à une mauvaise des ressources et à la perte de l'avantage de cette intégration. Ce problème a suscité une nouvelle direction de recherche dans la communauté temps-réel, et de nombreuses solutions ont été proposées. En 2007, Vestal a proposé un modèle pour représenter les systèmes avec différents niveaux de criticité dont les tâches ont plusieurs temps d'exécution, un pour chaque niveau de criticité. En outre, les conditions de validité des stratégies d'ordonnancement ont été définies de manière formelle, permettant ainsi aux tâches les moins critiques d'échapper aux délais, voire d'être abandonnées en cas de défaillance ou de situation d'urgence. L'introduction de plusieurs WCET et différentes conditions de validité ont accru la difficulté du problème de planification pour les systèmes avec differents niveaux de criticité. Les politiques de planification conventionnelles et les tests d'ordonnoncement se sont révélés inadéquats et le besoin de nouveaux algorithmes est apparu. Depuis, beaucoup de travaux ont été réalisés dans ce domaine. Dans cette thèse, nous contribuons à l'étude de l'ordonnancement dans les systèmes avec différents niveaux de criticité. La surcharge d'un système est représentée sous la forme d'un ensemble de tâches pouvant décrire l'exécution sur l'hyper-période de tâches ou sur une durée donnée. Ce modèle nous permet d'étudier la viabilité des tests de correction basés

sur la simulation pour les systèmes avec différents niveaux de criticité. Nous montrons que les tests de simulation peuvent toujours être utilisés pour ces systèmes, et la possibilité de l'ordonnancement du pire des scénarios ne suffit plus, même pour le cas de l'ordonnancement avec priorité fixe. Nous montrons que les politiques d'ordonnancement ne sont généralement pas prévisibles. Nous définissons le concept de faible prévisibilité pour les systèmes avec différents niveaux de criticité et nous montrons ensuite qu'une classe spécifique de stratégies à priorité fixe sont faiblement prévisibles. Nous proposons deux tests de correction basés sur la simulation qui fonctionnent pour des stratégies faiblement prévisibles. Nous montrons également que, contrairement à ce que l'on croyait, le contrôle de l'exactitude ne peut se faire que par l'intermédiaire d'un nombre linéaire de préemptions. La majorité des travaux reliés à notre domaine portent sur des systèmes à deux niveaux de criticité en raison de la difficulté du problème. Mais pour les systèmes automobiles et aériens, les normes industrielles définissent quatre ou cinq niveaux de criticité, ce qui nous a motivés à proposer un algorithme de planification qui planifie les systèmes à criticité mixte avec théoriquement un nombre quelconque de niveaux de criticité. Nous montrons expérimentalement que le taux de réussite est supérieur à celui de l'état de la technique.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In real-time systems, the execution of tasks is constrained by rigid timing restrictions. Temporal correctness of the system is as important as its logical correctness. Timing constraints are usually represented by enforcing deadlines on the executions of tasks. In hard real-time systems, these deadlines are absolute and all tasks must meet them for the functionality to be considered correct. Soft real-time system can be more flexible and it may be acceptable for a task to finish after a short period from its deadline.

Systems where failure of one or more components can lead to catastrophic ramifications on safety are characterized as safety-critical systems. Effects of a failure could include destruction or heavy damage to property, injury and loss of life. Some examples of safety-critical systems include automotive systems, nuclear reactors, certain medical devices, drones and avionics. Safety-critical systems can be real-time as well, in this case a failure can be caused by a delay in the execution of a task, causing it to violate the system's temporal constraints.

In general, not all functionalities in a safety-critical system are of critical nature. For example, a surveillance drone would constitute of a flight control system and a camera control/image processing system. The flight control is considered of high criticality, a failure within this functionality can lead to the destruction of the drone and possibly cause injury. The camera control functionality is of a lower criticality, effects of its failure can be a disruption of the video feed or a reset of the camera's controls. If the two functionalities of the drone are separated, each executing on its own physical hardware, the high criticality functionality would be isolated, preventing any interference from the low criticality component. In an alternative approach, the two different components can be integrated on the same hardware. Such systems, where functionalities of different criticalities share the same computational platform are known as Mixed Criticality (MC)

systems. Our work in this thesis focuses on the scheduling and schedulability tests of hard real-time MC systems.

1.1 Motivation

The integration of functionalities on the same physical platform has many desirable benefits. It reduces the cost, weight, size and energy consumption. These advantages can be crucial for the success of a system and can be the determinant that gives the advantage over competing devices in the market.

Due to the dangers posed in the case of malfunction, safety-critical systems need to be certified before they are implemented and deployed. The certification process is performed by a Certification Authority (CA), and it is the duty of this third party to verify that the system is safe. Certification is usually guided by documents containing technical specifications known as certification standards. The ISO 26262 standard [1] used in the automotive area, distinguishes between four different criticality levels. The DO-178 B/C standard [2] is used in avionic systems, it defines five Design Assurance Levels (DAL), from DAL-A to DAL-E, where DAL-A is the most critical level and failures can have tragic effects while a failure in a DAL-E does not affects the safety of the system.

In view of the different criticality levels considered in certification standards, conventional scheduling models proved very difficult to certify, as these models are unaware of the different criticalities of tasks. Consequently, all applications including non-critical ones, may need to be developed and certified by the same standards used for the highest criticality tasks in the system. This makes development and certification more expensive and more time consuming.

Another complication comes from the pessimistic assumptions taken by CAs about the execution conditions for higher criticality functionalities. Usually, the higher the criticality of a task the more pessimistic its Worst Case Execution Time (WCET) estimate is. This makes sense from a safety point of view since the task will have more time to finish successfully in-case of unforeseen complications. Nevertheless, this will result in an inefficient resource usage as the difference between the WCET and the average case becomes bigger. In worst cases, this could possibly render a system unschedulable, by making its estimated workload larger than the capabilities of the platform.

Conventional models and scheduling policies are not adequate to solve these difficulties that arise in mixed-criticality systems. New models, aware of the different criticalities of tasks, alongside criticality-aware scheduling policies have the potential to simplify the certification process and properly answer to the scheduling needs of these systems.

1.2 Mixed Criticality Systems

To bridge the gap between the execution time anticipated by the system designer and the more pessimistic WCET needed for certification, mixed criticality models allow tasks to have more than one WCET. In 2007, Vestal proposed a model [3] where each task has its own criticality, in addition to one WCET for each criticality level in the system. Correctness conditions for schedulability were formally defined, allowing non-critical tasks to violate their timing constraints in emergency cases, giving higher criticality tasks more time to finish their execution.

Different modes of execution are identified based on the on-line performance of jobs. During run-time, the system is assumed to run in nominal mode where all tasks must meet their deadlines. In the event where a critical task exceeds one of its worst case estimates without signaling termination, the system is allowed to change its execution mode from the nominal to a more critical mode. After this mode change, non-critical tasks, or tasks whose criticality is lower than the one that caused the mode change are allowed to miss their deadline and can even be dropped.

1.2.1 Challenges

To acquire correct behavior, with the necessary degree of assurance in a mixed-criticality system, a formal model that represents the system and clearly identifies the correctness of its scheduling policy must be chosen. After adopting an appropriate model, a suitable scheduling policy is needed to manage the use of shared resources and ensure that the timing constraints of the system are met.

The introduction of multiple WCETs, and a different correctness criteria in MC models, increased the complexity of the scheduling problem. In 2010, it was proven that the mixed-criticality scheduling problem is NP-hard even for two levels of criticality on a uniprocessor platform [4]. In recent work, the authors of [5] try to study the reason for the intractability of the scheduling problem in MC-systems. Two causes are identified, the first, as is described in their work, is the 'on-line' nature of the problem. This description comes from the fact that some information are only known during execution. The second reason found, is that we attempt to find efficient polynomial time algorithms to a computationally intractable problem.

The last necessary step to guarantee correct behavior is to verify the correctness of the scheduling policy. This is usually done by a schedulability test/correctness test. Since the model and the correctness criteria differ, conventional schedulability tests are not applicable for MC-scheduling policies and new tests are needed. Finding an MC-schedulability test is not an easy task even in simple cases. The challenge comes from

the correctness conditions, which in nominal cases, requires that all tasks adhere to the timing constraints imposed on the system, but in other cases, tasks are permitted to miss their deadlines while keeping the system in a correct state.

This difficulty becomes easily visible if we look at the case of fixed priority scheduling of jobs. In the conventional case, the worst case scenario is easily identified as the scenario where all jobs execute for their WCET and it is enough to test the correctness of this scenario to verify the correctness of all others. In MC-scheduling, it becomes more difficult to identify the worst case scenario, as jobs have multiple WCETs and if one job executes for more than its WCET others can be dropped. In fact, in our study we show that there is no one worst case scenario, in the sense that no scenario can be tested that can cover all the rest.

As a result of these challenges, a big portion of the work done in mixed-criticality scheduling is for dual-criticality system, containing only two levels of criticality. This assumption simplifies the problem at hand, still bears useful theoretical results, and can be directly used for some cases in avionics where system functionalities are divided between mission-critical and safety-critical. However, recent efforts encourage the consideration of more criticality levels as most of the standards in industry identify four or five criticality levels.

1.3 Contributions and Structure

A big part of this thesis is dedicated to the study of correctness testing and its complexity in MC-systems. We focus on simulation-based correctness tests and try to identify the conditions needed for such tests to be usable in the mixed criticality context. We represent the workload as a set of independent jobs, whereby a correctness test must give a verdict whether a scheduling policy can correctly schedule all instances for the given job set. In the mixed criticality case, it is not enough to test only the worst case scenario and we provide examples why this is not sufficient.

Chapters 2 and 3 give an overview of the prior work done in the field and essential formal definition for the model used in our work. In Chapter 4, by means of an example, we show that, contrary to what was believed in the literature, until a short time ago, linear number of preemptions are not necessarily enough, in general, to correctly schedule a system. This has important consequences on the complexity of correctness testing.

In Chapter 5, we continue our work oriented towards testing the correctness of MC-scheduling policies by investigating their predictability property. We find that these policies tend not to be predictable in general, and we define weak-predictability, which is a less restrictive characteristic, more suitable for MC-systems. We prove that an

important class of priority based policies is weakly predictable for single processor and some multi-processor cases. After that, we propose two correctness tests that can be used for weakly-predictable scheduling policies.

Motivated by the need for scheduling algorithms that can handle systems having more than two criticality levels, in Chapter 6 we present a scheduling policy that generates a set of time-triggered tables, one for each criticality mode. The presented scheduling policy is applicable to systems having any number of criticality levels and allows jobs to have dynamic priorities. Experimental results indicate that it outperforms two state of the art algorithms.

Using our proposed algorithm, in Chapter 7, we demonstrate how an MC-scheduling policy can be used as a fault-recovery strategy. In this chapter, we represent the system, the jobs and the scheduler, as a set of synchronized Timed-Automata (TA) components. We use the work of Dragomir et al. [6] were the authors define how to systematically and automatically generate a Fault Detection Isolation and Recovery (FDIR) component composed of a diagnoser/controller to detect certain types of failures in a system. We describe how the component generated from the scheduling policy can be used as a part of the FDIR component to guide the recovery process of the system in the case of failures.

In Chapter 8, we introduce a model for designing an MC-system with coarse grained multi-core interference. In our design flow we employ a concurrency language, also based on synchronized timed-automata, that can be used for designing, at high abstraction level, custom resource management policies that can handle interference and mixed-criticality. We compile the application into a representation in this language and combine the result with a resource manager into a joint software design used to deploy the given system on the target platform.

Finally in Chapter 9, we conclude our work and discuss problems that we find interesting as future work.

Chapter 2

Prior Work

The first step in solving any research problem is finding a correct problem formulation. In the mixed-criticality field, the basic problem formulation was introduced by Vestal, and became known as Vestal's model. As research in this topic advanced, newer problem formulations that extend the original one were proposed. We review some these in section 2.1. The rest of the sections of this chapter are dedicated to review the prior work in scheduling and schedulability tests in MC-systems.

2.1 Problem Formulations

2.1.1 The Vestal Model

In Vestal's model [3], a system's workload is represented by a set of periodic tasks. These tasks are considered to have different criticalities, which in general, have to be designed with different assurance levels. To represent this in the model, Vestal defines an ordered set of four 'design assurance levels' $L = \{A, B, C, D\}$, with A being the highest level. A task τ_i is defined by its period T_i, deadline D_i, its appropriate design assurance level L_i, which can be identified during safety analysis, and four WCET estimates $C_{iA} \geq C_{iB} \geq C_{iC} \geq C_{iD}$ each providing a different degree of assurance.

Vestal's paper [3] is considered as the fundamental work that launched the wave of research in the mixed criticality area. Since then, hundreds of results have been developed in this domain that use Vestal's model or a variation of it. In the coming subsections we describe some of these models.

2.1.2 The Burns and Baruah Model

In this model [7], tasks are partitioned over a finite set of components. Each component is allocated a criticality level. Similar to Vestal's model, a task is defined by a period,

a deadline, a set of worst case execution estimates and a criticality level. Instead of predefining only four assurance levels, Burns and Baruah define the execution estimates as a vector $\vec{C_i}$ where $C_i(l)$ is the worst case execution estimate for criticality level l. This removes the limit of only four levels of assurance from Vestal's model.

In addition to the relation between WCET estimates and the criticality level, this model assumes a relation between the criticality level and each of the deadline and period parameters. It is presumed that if a task τ_i is to be moved from a criticality level L_i^1 to a higher one L_i^2, then $D_i^2 \leq D_i^1$ and $T_i^2 \leq T_i^1$ which makes sense since higher levels of assurance are expected to be more rigorous. During runtime, if a task of criticality level L_i exceeds its $C_i(L_i)$, tasks of the same criticality level or lower are prevented to execute until the next time the processor is idled [8].

2.1.3 The Elastic Mixed-criticality Task Model

In favor of providing high criticality tasks more time to execute, it is acceptable in an MC-system to prevent lower criticality tasks from execution, run them in degraded mode or even drop them [8, 9]. In attempts to guarantee some service for low criticality tasks even under critical conditions, *Su et al.* proposed an Elastic Mixed-Criticality ($E-MC$) task model [10]. The ($E - MC$) model is designed for dual-criticality systems. High criticality tasks have two WCETs and are described as in the previous models. The difference is that low criticality tasks have two periods. The first one, referred to as the 'desired period' and is equivalent to the usual period of tasks in other models. The other is called the 'maximum period' which is larger than the 'desired period' and is used to represent the minimum service requirement of low tasks.

The system is considered correctly schedulable if the execution requirement of high criticality tasks and the minimum service requirement of low criticality tasks are ensured. Additionally, low criticality tasks have a set of possible early-release points, which allows them to release early, exploiting the slack time produced from the execution of high tasks.

2.1.4 The Ekberg and Yi Model

The aim of the model proposed in [11] is to generalize the mixed-criticality task model as much as possible. The authors believe that the general assumption in MC-systems, that different criticality levels provide different levels of assurances related to temporal constraints, should not be enforced. Rather, it should be the system designer that decides what different criticality modes represent. Transitions between different modes of the system are defined using a Directed Acyclic Graph (DAG) where the nodes are the criticality modes and the edges represent legal mode changes. Furthermore, task

parameters such as deadlines and periods can be changed between different criticality modes, and new tasks can be added as well in higher criticality modes.This was motivated by scenarios where a failure or a malfunction occurs and additional tasks may need to execute to handle failures or compensate for the missing functionality.

2.2 Job Scheduling

2.2.1 Fixed Priority Policies

A fixed-priority scheduling policy is a *work-conserving* policy that assigns a priority for each task/job in the system. The solution provided is usually represented in the form of a priority table which defines a total ordering relationship between jobs. Priorities of jobs are fixed and do not change throughout the execution of the system. During run-time, a fixed-priority scheduling policy always schedules the highest-priority job that has arrived and has not completed yet.

In [12], it was shown that Earliest Deadline First (EDF) is not optimal for the scheduling of MC-systems The addition of criticality levels, even only two, introduces feasible systems that can not be scheduled by EDF.

A noteworthy result in fixed priority scheduling for mixed-criticality systems is the work by Baruah *et al.* in [13]. Motivated by increasing the utilization of resources for MC-systems that adhere to demanding certification requirements, the authors propose a new fixed-priority scheduling algorithm, Own Criticality Based Priority (OCBP), designed for certifiable mixed criticality systems. The algorithm tries to recursively find the job with the lowest priority. First, from the set of all jobs who were not allocated priorities, a candidate job is chosen to be the lowest priority one. Then a simulation is performed where all jobs execute for the WCET estimated for the criticality level of the job. If the candidate job is able to terminate before its deadline then it is assigned the lowest priority and the algorithm start searching for the second lowest priority and so on. If the candidate job does not meet its deadline, another job is chosen from the set of jobs.

A system is deemed schedulable if, by following the process described above, all jobs are successfully assigned priorities. The algorithm fails to schedule the system, if it reaches an iteration where a lowest priority job can not be found. In [4], OCBP was proven to be optimal among fixed-priority policies. That is if a system can not be scheduled by OCBP then no other fixed-priority algorithm can correctly schedule it. In addition, load based schedulability analysis for the algorithm was presented, defining two load value, one for each criticality level. OCBP was extended to sporadic task systems by Li *et al.* in [14].

2.2.2 Extended Fixed Priority Policies

Although all jobs must meet their timing constraints under normal circumstances, most mixed criticality problem formulations allow non-critical jobs and jobs of lower criticality to miss their deadlines in case a high criticality job exceeds its lower worst case estimate without completion. This lead to the distinction between different modes of execution in an MC-system where correctness conditions can differ from one execution mode to another. Being mode-unaware, the effectiveness of fixed-priority policies was bounded, and extension models were proposed to overcome this limitation. The work in [15] allows tasks of lower criticalities to be abandoned in case a task of higher criticality needs more time to finish. The priorities of the high criticality tasks are allowed to be changed as well.

Another extension for uniprocessor platforms was later proposed by Chen et al [16]. In their work, the authors try to generalize fixed-priority scheduling and introduce the Generalized Fixed-Priority (GFP) scheme. They distinguish between three different execution phases in a dual-criticality system. A steady low criticality mode, a transition period and a steady high criticality mode. Tasks are assigned three priority parameters one for each mode. During normal execution jobs use priorities for the low mode. After a criticality change occurs, the system moves from low mode to the transition period. In this period, high criticality tasks, that have been dispatched but did not complete before a mode change use the priority assigned for this mode, while other high criticality tasks that are released after the mode change use the priority assignment for the high mode. Experimental results show that better schedulability could be achieved by this generalized scheme.

In [17], the authors proposed a Fixed Priority per Mode (FPM) algorithm for dual-criticality uniprocessor systems, and provided a theoretical proof of its dominance over OCBP. The proposed algorithm, Mixed Criticality Earliest Deadline First (MCEDF), supports precedence-constraints for jobs and generates two priority tables, one for each mode. For the high criticality mode, it uses the EDF policy, since it is optimal for single criticality scenarios. To generate the priority table for low criticality mode, the workload is divided into busy intervals, which are maximal time intervals such that processor is not idle. The algorithm tries to find the lowest priority job in each busy interval favoring low criticality jobs if possible. Then, priority constraints are generated for different busy intervals. These are represented by a directed graph, where nodes refer to jobs and the edges describe the priorities between them. Once a low priority job is found it is removed from the set of jobs in the system and a new iteration begins to find the next job. Once done, a total ordering of jobs is achieved by a topological search.

To check the correctness of their results, a simulation over a set of scenarios referred to as 'basic HI scenarios' is done. This was assumed enough to guarantee the schedulability of all scenarios as the chosen set covers the most conservative high criticality scenarios and fixed priority policies are know to be predictable [18]. Later in Chapter 5, we show that FPM policies are not in general predictable, we also provide theoretical prove that the method used to test the correctness of MCEDF is indeed valid under the assumptions presumed.

For multi-core platforms, an FPM algorithm was presented by Dario et al. [9]. Making use of a base algorithm, which gives a global fixed priority ordering of jobs, the Mixed Criticality Priority Improvement (MCPI) algorithm attempts to improve the schedulability for MC-systems by trying to increase the priority of high criticality jobs. An increase of schedulability up to 30% was gained in comparison to traditional solutions.

2.2.3 Time-triggered Policies

Time-triggered policies define a static, pre-computed table which determines at every instant of time which job must be scheduled at each processor provided that it did not terminate yet and assuming that the job may require up to its worst case execution time. Similarly to fixed priority policies, one time-triggered table is not enough to efficiently schedule MC-systems.

The *Single Time Table per Mode (STTM)* scheme was introduced in [19] as an extension to time-triggered policies, assigning one time-triggered table for each criticality mode. By the use of a criticality level indicator during run-time, the system can keep track of the current execution mode. The system is assumed to start in low criticality mode and switches to high criticality mode as soon as a job exceeds its WCET without signaling completion. The appropriate time-triggered table is used to schedule each execution mode. The authors of [19], also present an algorithm for dual-criticality systems that uses the priority table generated from OCBP to generate two time-triggered tables, one for each mode. The time-triggered table for the low mode is generated by simulating all jobs under the assumption that each will need its low WCET to terminate. The second time-triggered table is generated in the same manner but high criticality jobs are assumed to execute for their high worst case estimate.

In [20], the authors show that the STTM scheduling paradigm dominates FPM and they proposed an algorithm that transforms any FPM scheduling policy to an equivalent STTM policy. The process of generating the time-triggered tables is again done by simulation using the fixed priorities obtained from the FPM algorithm. However, to guarantee correctness after a mode change, in the simulation used to generate the time-table for high criticality level, some jobs are disabled until the time they are scheduled to

execute in the lower criticality table. Other approaches were also proposed for buliding time-triggered tables. Theis *et al.* [21] showed how to build time triggered tables using search tree. Another method using linear programming was demonstrated by Jan *et al.* [22].

2.3 Task scheduling

2.3.1 Uniprocessor Scheduling

In [23], a sufficient response time analysis is provided for the scheduling of sporadic task systems that have monitoring capabilities. The authors introduce the Adaptive Mixed Criticality (AMC) scheme and show that it dominates the earlier Static Mixed Criticality (SMC) scheme. The boost of schedulability in the AMC scheme is a result of stopping the execution of all low criticality tasks in the event that a job executes for more than its low WCET. Although AMC is dominant in term of schedulability, SMC still has the benefit of not dropping all low criticality tasks, but instead only drops the low criticality task that execute for more than its WCET allowing the rest of the low criticality tasks to complete after their deadline. The work in [23] focused on systems of two criticality levels, Flamings et. al. developed an extension for the AMC scheme for criticality systems with more than two levels [24]. Other extensions for AMC where proposed by Huang et. al. [25], Zhao et. al. [26] and Burns et. al. [27].

An algorithm called EDF-VD for the scheduling of MC-sporadic-task systems was proposed in [28, 29] focusing on dual-criticality systems. For each high criticality task the algorithm tries to find a modified period that is smaller than the original one. Virtual deadlines for jobs are then computed using the modified periods of tasks. If a job is of low criticality then its virtual deadline is set to be the release time of the job incremented by the tasks period. If the job is of high criticality, its release time is incremented by the new modified period. During run-time, jobs are scheduled using EDF policy, but in case a job passes its low WCET without termination then all low criticality jobs are discarded and high criticality jobs are scheduled using the original deadlines. EDF-VD was shown to be able to schedule any system schedulable by a clairvoyant algorithm with a processor that is 4/3 times faster. Later on, an implementation and a schedulability test for EDF-VD targeting systems of more than two criticality levels was presented in [30].

In [31], scheduling strategies were proposed and evaluated for preemptive and non-preemptive systems with varying-speed processors. A processor is defined by two execution speeds, normal and degraded. Relating this problem to MC-systems, the correct schedulability of the system demands that all tasks verify their temporal constraints in

normal speed and only critical ones must provide correctness guarantees when the processor speed degrades. The authors make use of linear programming to construct tables for scheduling on processors with varying speeds. Scheduling algorithms and sufficient schedulability tests were proposed in [32] for the case of systems where varying-speed processors do not have monitoring capabilities.

Zero-slack scheduling was presented in [33] to protect from the criticality inversion problem. Criticality inversion occurs when a low criticality task interrupts a high criticality tasks that has already overrun its low WCET estimate. An algorithm is suggested that reduces the overall needed utilization by lowering the number of preemptions. In addition a zero-slack rate monotonic scheduler is presented.

2.3.2 Multiprocessor Scheduling

Two common scheduling approaches for multiprocessor platforms are the partitioned and global scheduling schemes. The partitioned approach does not allow tasks migration among processors, instead it generally has two phases, task allocation and priority assignment, whereas global scheduling allows the migration of tasks. For MC-systems, a number of partitioned scheduling algorithms were proposed [34, 34–38].

In [34], the zero-slack rate monotonic algorithm [33] was extended to the multiprocessor case. To protect the temporal correctness of high criticality tasks in overload scenarios having execution spikes, a criticality-aware packing algorithm called Compress-on-Overload Packing (COP) was proposed. The algorithm consists of two phases, the first allocates tasks to processors using three variants of bin packing algorithms. This process takes into account that the high criticality tasks can take up till their highest execution estimate thus making sure that they still meet their deadlines in cases of overload. Tasks that do not fit during this step are packed using a modified version of worst-fit decreasing packing. In the second phase, the zero-slack algorithm [33] is used for each set of allocated jobs.

Kelly *et al.* proposed another task allocation heuristic, again inspired from variants of bin-packing schemes. They show that ordering tasks by either decreasing utilization or decreasing criticality before applying the packing algorithms increases the number of schedulable tasks using a rate monotonic algorithm or Audsley's algorithm [39]. Best experimental results were obtained using criticality decreasing ordering of tasks with Audsley's priorities.

In [40], a different approach for partitioned scheduling is considered. Dual-Partitioned Mixed-Criticality (DPM) algorithm is presented that allows some migration of tasks while trying to maintain the advantages of partitioned systems. The authors introduce the dual-partitioning approach, in which high criticality tasks, following the partitioned

scheme, are statically allocated to processors, but low criticality tasks are allowed limited migration. During execution within a single criticality mode tasks are not allowed to migrate, but during a mode change, instead of dropping low criticality tasks, they are allowed to migrate to different cores. Experimental results show that dual-partitioning can enhance the schedulability of fully partitioned algorithms.

Li and Baruah investigated the global scheduling approach for implicit-deadline sporadic MC systems. In [41], they extended EDF-VD [29] to multi-core systems and provided sufficient schedulability conditions. The algorithm uses the same procedure to generate modified periods as in [29]. After which, *fpEDF* [42], a criticality-unaware global scheduling algorithm is used to schedule the tasks with modified periods. In the case a job executes for more than its low WCET without signaling termination, all low jobs are dropped and high jobs continue their execution using *fpEDF*.

Finally, before ending this chapter, we mention some related work for managing access to shared resources. Sharing resources among functions of different criticality levels is an important factor for reducing cost and improving system efficiency. The priority ceiling protocol [43] was extended in [44] for mixed criticality systems. Criticality specific blocking terms were added allowing low criticality applications to turn over resources' budgets to higher criticality applications. Likewise, the Stack Resource Protocol (SRP) was extended to MC-systems by Zhao *et al.* [26].

Chapter 3

Model Formulation

3.1 The Workload Model

The system's workload is modeled by a set of independent mixed-criticality jobs. Working with a set of jobs, as opposed to working with tasks, allows us to explore the usability of exact correctness tests in MC-systems by simulating the execution of jobs over an interval of time. This allows the schedulability of the system to be evaluated up until a point in time. If the time interval is chosen to coincide with the hyper-period of the tasks then the selected set of jobs can represent the entire system. The hyper-period is the least common multiple of all task periods, thus hyper-period intervals contain the same set of jobs.

Although hyper-periods can get very large in practice there have been efforts towards minimizing the hyper-period of a task set. In [45, 46], the authors try to reduce the length of a hyper-period by making use of period variations. In their approach, tasks' periods are first given in a range of valid periods. Then they propose an algorithm that employs the ranges of periods for different tasks to find a task model where each task has one period from its set of valid periods such that the hyper-period of the new system is as small as possible.

We adopt a model similar to the one proposed by Burns and Baruah [7], but with one difference, for MC- systems with more than two criticality levels, a job has only two worst case estimates instead of one for each level.

An MC-job. A job J_i in an MC-system is defined by the five parameters $J_i = (A_i, D_i, X_i, C_i^N, C_i^E)$ where:

- $A_i \in \mathbb{N}^+$, the arrival time
- $D_i \in \mathbb{N}^+$ & $D_i > A_i$, the deadline relative to the arrival time

- $X_i \in \mathbb{N}^+$, the criticality of the job
- $C_i^N \in \mathbb{N}^+$, the WCET estimated for nominal cases
- $C_i^E \in \mathbb{N}^+$, the more pessimistic WCET estimate

C^N is used to represent the worst case estimate in **nominal** cases. We assume that this is provided by the system designer for all the tasks in their system. The second execution estimate, C^E is derived for critical tasks only to ensure their correct behavior under harsher conditions or **emergency** cases. We assume that for a non-critical task $C_i^N = C_i^E$, and for critical tasks $C_i^N < C_i^E$. We also define C_i^U to be the uncertain execution time estimated for a job as $C_i^U = C_i^E - C_i^N$.

An MC-instance. An MC-instance $\mathcal{I}$ is a set of n MC-jobs.

A scenario. A scenario c of an instance $\mathcal{I}$ is a vector of size n of execution times for all jobs $(c_1, c_2, ..c_n)$, where c_i is the execution time needed for job J_i to finish in the given scenario.

We find it convenient to define a special set that holds jobs which execute for more than their normal worst case execution time estimate but less than the emergency one.

Definition 3.1. The **emergency set** J^E of a scenario c is defined by:
$J^E = \{J_i \in \mathcal{I} : C_i^N < c_i \leq C_i^E\}$

If the emergency set of a scenario c is empty i.e. $J^E = \{\}$, then we say that c is of **low criticality**. Otherwise, the scenario c is considered of **criticality level** ℓ, where $\ell = max_{J_i \in J^E}(X_i)$.

A basic scenario. It is a scenario in which for each job J_i we have , $c_i = C_i^N$ or $c_i = C_i^E$.

A LO-scenario. It is the basic scenario where all jobs execute for exactly C_i^N.

A schedule. A schedule $\mathcal{S}$ of a given scenario is the mapping:

$$Time \rightarrow J_\epsilon \times J_\epsilon \times ... \times J_\epsilon = J_\epsilon^m$$

Every job should start at time A_i or later and run for no more than c_i time units.

A scheduling policy. A scheduling policy for an instance $\mathcal{I}$ specifies deterministically which job to execute at each time instant.

A schedule for a scenario of criticality level ℓ is **feasible** if all jobs J_i with $X_i \geq \ell$ are given c_i execution time between their arrival and deadline.

A scheduling policy is **correct** if for every valid scenario c it generates a feasible schedule. As a consequence, a correct scheduling policy will ensure that:

- For a low criticality scenario, all jobs must complete before their deadlines.
- For a scenario of criticality level ℓ, all jobs of criticality level ℓ or more must complete before their deadline.

A job is considered **ready** at time t if that job has arrived but not completed at time t. The state of the scheduler at every time instance t during run-time is composed of the set of terminated jobs, the set of ready jobs at t, the progress of ready jobs in case of preemption, the current executing job and the current **criticality mode**, referred to by χ_{mode}, initialized as 1 and changed to higher value in case a mode switch occurs.

Mode switch . In mixed criticality scheduling a **mode switch** or a **mode change** occurs when a job, whose criticality is higher than the current criticality mode χ_{mode}, executes for more than its normal worst case estimate C^N. As a result the criticality mode is increased to match the critiality level of the job that caused the mode change.

Dual-criticality systems. An MC-system that only considers two levels of criticality is known as a dual-criticality system. The two criticality level are labeled LO and HI, representing the low and high criticality levels respectively. Similarly, jobs are labeled as LO jobs and HI jobs, representing the low and high criticality level jobs. A Fixed Priority per Mode (FPM) scheduling policy in a dual-criticality system, is a mode-switched policy with two tables: PT_{LO} and PT_{HI}. The former includes all jobs, and the latter only HI jobs. As long as the current criticality mode χ_{mode} is LO, this policy performs the fixed priority scheduling according to PT_{LO}. After a switch to the HI mode, this policy drops all pending LO jobs and applies priority table PT_{HI}. For Single Time Table per Mode (STTM) policies, the two time-triggered tables are T_{LO} and T_{HI} the tables for the LO and HI mode, respectively. The two STTM tables are correct iff:

1. They schedule all jobs after their arrival and before their deadline, allocating each job C^N time units in LO table and each HI job C^E time units in HI table.
2. If at any time we switch from LO to HI, then all not-yet-terminated HI jobs will have enough time to continue their execution so as to reach C^E time units.

In Chapter 6 we will propose an STTM policy for systems with more than two criticality levels. For such systems, an STTM policy defines one time-triggered table T_l per criticality mode l for $1 \leq l \leq L$ with L being the highest level of criticality in the system.

Definition 3.2 (Reasonable Policies)**.** A single-processor dual-criticality scheduling policy is 'reasonable' if after the mode switch it applies the EDF policy to schedule the HI jobs and either drops the LO jobs altogether or gives them less priority than that of any HI job. In particular, in the case of FPM policies, 'reasonable' means that $PT_{HI} = EDF$ table.

Definition 3.3 (FPM equivalent tables)**.** A dual-criticality FPM policy is said to have *FPM equivalent tables* if the relative priority order of HI jobs is the same in both PT_{LO} and PT_{HI}.

Chapter 4

On the Complexity of Testing a Scenario

It was claimed in [4] that an optimal scheduling policy executes any basic scenario with only a linear ($O(n)$) number of preemptions for a fixed number of criticality levels L. More precisely, they claim what is reproduced below as Lemma 4.1.

Lemma 4.1 (Refuted Lemma [4])**.** *If an instance is MC-schedulable, then there exists an optimal online scheduling policy that preempts each job j only at time points t such that at time t either some other job is released, or j has executed for exactly $C_j(i)$ units of time for some $1 \leq i \leq L$.*

In the lemma $C_j(i)$ is the WCET estimate for job j at criticality level i and L is the number of criticality levels. In our notations, for dual-criticality systems, level 1 is LO, level 2 is HI, $C_j(1)$ is C_j^N, $C_j(2)$ is C_j^E.

Lemma 4.1 states that if an instance is MC-schedulable then there exists a correct scheduling policy that only preempts a job J_j either when some other job in the instance is released or when J_j executes for exactly one of its WCET estimates.

In this work, we present a refutation to Lemma 4.1 in the form of a counter example. In Example 4.1 we give a dual-criticality instance and show that no correct schedule exists for that instance where jobs can only be preempted as specified by Lemma 4.1. Thus according to the lemma this instance should not be MC-schedulable. However, we show that a correct schedule exists for our example and the instance is indeed schedulable but one of the executing jobs needs to be preempted at a time different from the ones allowed by the revisited lemma.

Example 4.1. *Consider the following problem instance:*

Job	A	D	χ	C^N	C^E
1	0	14	*HI*	6	7
2	0	11	*LO*	5	5
3	5	10	*HI*	2	3

First, we check if it is MC-schedulable by following the preemption rules in Lemma 4.1. At $t = 0$ a scheduling policy can execute either job J_1 or job J_2 since job J_3 did not arrive yet. According to the lemma, whichever job is chosen should not be preempted before $t = 5$, since before that time the chosen job will not have executed for its C^N and no other job will have arrived. Thus we have two cases:

- J_1 is chosen. It can be the case that it does not signal termination before $t = 5$. At that instant J_1 can be interrupted because J_3 arrives. But now both jobs J_2 and J_3 have to finish in the interval $[5, 11]$ in order not to miss their deadline. The interval is 6 time units, but the jobs can have a combined execution of $7{=}(5 + 2)$ units in the LO scenario. Thus no schedule exists that executes J_1 in $[0, 5]$ and abides by the rules of Lemma 4.1.

- J_2 is chosen. Then it must also execute uninterrupted until it terminates at $t = 5$. What is then left to execute are the two high criticality jobs. We keep in mind that if an instance is MC-schedulable then a scheduling policy should be able to schedule all correct scenarios. One possible scenario is that both of the jobs execute for their C^E. In that case a total of $10{=}(7 + 3)$ units of execution are needed. For both of the jobs not to miss their deadline they have to terminate before $t = 14$ and in the execution window $[5, 14]$, we only have time to execute 9 units which is not enough. Thus we conclude that no schedule exists that executes J_2 in $[0, 5]$ and abides by the rules of Lemma 4.1.

Hence, according to Lemma 4.1 this instance is not MC-schedulable, but that is not correct. Figure 4.1 shows a Gantt chart representing an STTM scheduling policy that correctly schedules the instance, contradicting Lemma 4.1. Recall that this policy starts execution in static table 'T_{LO}' and keeps using this table as long as there is no switch to the HI criticality mode $\chi = HI$, in which case it switches to static table 'T_{HI}'. This example shows that an instance can be MC-schedulable but no optimal online scheduling policy exists that preempts a job j only at time points where another job is released or j has executed for exactly $C_j(i)$ units.

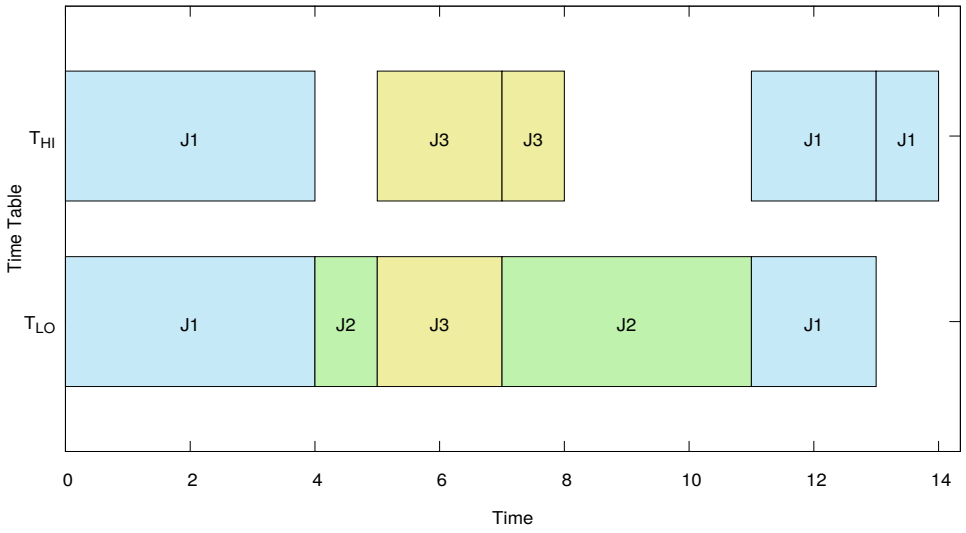


FIGURE 4.1: Valid time-triggered tables for the instance in Example 4.1

4.1 Consequences for Complexity

Taking into account our counter example, the authors of [4] published [47], an erratum to [4], where they replace Lemma 4.1 by a proof, from which follows that the upper bound on the number of preemptions is not $O(n)$ but instead $O(n^2)$. Thus by the refutation presented in this section and the erratum [47], the best known upper bound on the number of preemption is increased by one order number 'n' of jobs.

When testing for correctness, taking into account only the complexity of testing one scenario, is not enough. In [47], it was established that deciding schedulability of an MC-system with a constant number of criticality levels L, can be done in $O(n^L)$ time. In addition, the authors show that testing the correctness of a given solution for the LO scenario can be done in $O(n^2)$ for the general case.

In the next chapter, for dual-criticality system, we propose a correctness test for policies derived from fixed priority scheduling, where Lemma 4.1 remains correct, and only $O(n)$ preemption are required per scenario. Nevertheless, we show that $O(n)$ basic scenarios need to be tested to assure correctness thus bringing the lower bound back to $O(n^2)$. We show that, the lower bound can be brought further down with a more efficient test than enumerating $O(n)$ basic scenarios.

Chapter 5

Correctness in MC-scheduilng

Evaluating the correctness of scheduling policies is a non-trivial task. In this chapter, we start by showing that testing for correctness in the case of mixed-criticality proves to be more complicated than for conventional scheduling. Example 5.1 shows one case where a fixed-priority scheduling policy is deemed schedulable for a given scenario but fails to schedule an 'easier' scenario that differs from the original by decreasing the execution of one job. Policies possessing such unintuitive behavior are said to be not predictable.

In the next section, we discuss predictability and sustainability of scheduling policies/schedulability tests and their adaptation to MC-scheduling. These two characteristics are important in the study of correctness and can be sometimes essential for the applicability of a correctness test. Section 5.1.1 describes sustainability as defined by *Baruah and Burns* in [48], its extension to mixed-criticality by *Guo et al.* in [49]. After that, we discuss predictability, and in Section 5.2, we extend predictability to weak-predictability for mixed-criticality systems. In Sections 5.3 and 5.4, two correctness tests are proposed for weakly-predictable scheduling policies.

5.1 Fundamental Correctness Concepts

The goal of a correctness test is to give verdict whether a given scheduling policy can correctly schedule the workload of a system. This can be done by evaluating whether jobs, over all possible scenarios, will have enough execution time to terminate before their deadlines. In general, for real-time systems, correctness testing is achieved either by using analytical upper bounds, or by simulation. The former is necessary for more general task system, such as sporadic tasks, while the latter can give better results in systems that can be represented as a set of fixed jobs. In our study of correctness, we focus on simulation based correctness tests that exploit the possibility to *evaluate the*

tight termination time bounds by direct simulation. For such a test to be meaningful, the scheduling policy has to be predictable. In addition, we will demonstrate that for our goal of finding tight upper bounds by direct simulation it is essential to look not only at WCETs, but also at *actual* execution times that can occur in the system. Otherwise one will not be able to correctly schedule certain non worst-case scenarios, even in the case where the policy in question was proved to be sustainable.

In the conventional scheduling theory for fixed job systems, distinction between sustainability and predictability was not needed. In mixed-criticality scheduling, the distinction between predictability and sustainability becomes more pronounced as sustainability does not always ensure predictability. To clearly distinguish the difference between the two concepts, we start by including a formal definition of both, followed by an example (Example 5.1), where a fixed priority scheduling policy is sustainable but not predictable.

5.1.1 Sustainability

A good amount of research has been devoted to the study and analysis of sustainability in the scheduling of real time systems. In [48] the authors formalize the sustainability characteristic in real time systems (non mixed-criticality) as follows.

*"**Sustainability [48].** A schedulability test for a scheduling policy is sustainable if any system deemed schedulable by the schedulability test remains schedulable when the parameters of one or more individual jobs are changed in any, some, or all of the following ways: decreased execution requirements; later arrival times; smaller jitter; and larger relative deadlines".*

In this thesis, and from the definition above, we consider 'sustainability' to be a property of not only a policy itself, but rather of a pair 'a policy plus a correctness test'. We find it convenient to join the two into the notion of a '*scheduling algorithm*'. One example is the well-known 'single-processor EDF scheduling algorithm', which, next to implying the EDF policy, also implies the correctness test checking whether the total task utilization does not exceed 100 %.

The correctness criteria for a scheduling policy in an MC-system is different than the conventional single criticality case, since not all jobs always have to meet their deadlines. As a consequence, determining the schedulability of a scheduling policy is different and the sustainability definition needs to be revised. Guo et al. [49] extended the definition of sustainability to mixed criticality systems. In their work, they represent the workload of an MC system as a finite collection of sporadic tasks with each task having a criticality and possibly generating an unbounded number of mixed criticality jobs. MC-sustainability was defined as follows.

*"**MC sustainability [49].** An MC scheduling algorithm is said to be sustainable if any MC instance that is MC-schedulable by the algorithm remains so if one or more of the parameters characterizing the instance is improved. Improvements to be characterizing parameters:*

1. *Decreasing WCET parameters*
2. *Increasing periods for sporadic task systems*
3. *Postponing relative deadlines*
4. *Decreasing the criticality level assignment of a task/job."*

The authors of [49] focus their work on the dual-criticality problem. Six mixed-criticality scheduling policies were evaluated in [49] to demonstrate that they are MC-sustainable for the different parameters. It was found that the polices are all sustainable with respect to WCET, periods and deadlines, but some are not sustainable with respect to the criticality level parameter.

5.1.2 Predictability

The similarity between predictability and sustainability is that both of these properties preserve schedulability under *"decreased execution requirements"*. In sustainability analysis, as we saw in the previous section, execution requirements can be in the form of WCETs. These are upper bounds that may overestimate the worst case of the execution due to difficulties in modeling processors with caches, out of order executions, pipelines etc. Even in the case of a tight upper bound, a job can take numerous execution paths that are different from its worst-case path resulting in various execution times that are considerably less that the WCET.

It should be noted that predictability is not exactly the same property as sustainability. Sustainability is a property of a given correctness test [48], while predictability is a property of a given policy. Sustainability states that if the correctness test passes when assuming certain worst case system parameters then the system remains to be correct when the actual parameters are better than the worst case assumed. For sustainability, the correctness test can be anything from a simulation of the policy to an analytical formula that calculates upper bound on response times. By contrast, predictability can be seen as a special case which assumes that the correctness testing necessarily consists in simulation. Saying that the policy is 'predictable' is the same as saying simulation based test is 'sustainable' for it.

Definition 5.1 (Predictability)**.** A scheduling policy is said to be **predictable**, if for any scenario that is MC-schedulable by the policy, any other scenario that is better is

also MC-schedulable by the policy.
For predictability, "MC-schedulable" means that *simulating* the given scenario shows that the policy correctly schedules all jobs, Scenario c_1 is considered to be "better" than scenario c_2, if any job in c_1 executes for the same amount or less than it does in c_2.

In the next subsection, we show that in MC-systems, sustainability under a certain correctness test, does not result in predictability and we clarify why the two notions are fundamentally different in this case.

5.1.3 A Sustainable yet non-predictable Example

The Criticality Monotonic (CM) scheduling algorithm is one of the six algorithms that were studied in [49]. The policy schedules, at each time instant, a ready job of the highest criticality. It was proven in [49] that the CM scheduling algorithm using deadline monotonic scheduler within a criticality level is MC-sustainable. As a consequence, it is sustainable with regards to the *"decreased execution requirement"* represented, in this case, by the WCET parameter.

Although the CM algorithm is MC-sustainable, the example below shows that the underlying policy is not predictable. Indeed, a case can be found where the policy will correctly schedule a scenario of given instance but will fail to schedule a better one.
Hereby there is no contradiction with the results of [49], as the CM algorithm would give a verdict that the given system is not schedulable.

Example 5.1. *Consider a periodic system that has two tasks with periods $T = 20$. Since the tasks have the same period, each task will generate one job in the hyper-period. We also assume that at the start of the system each task will release a job. The jobs to be scheduled are shown in the table below:*

Job	*A*	*D*	*Criticality*	C^N	C^E	*Priority*
1	0	10	*HI*	6	8	1
2	0	10	*LO*	5	5	2

We assume the system is being scheduled by the CM policy with deadline monotonic scheduler within each criticality level. Testing the correctness of this policy for the worst case scenario, a simulation will be performed where J_1 is assumed to execute for 8 units of time and J_2 for 5.

The simulation will execute J_1 first, as it is the highest criticality job, until it terminates at $t = 8$. Then it will schedule J_2 which will terminate at $t = 13$, missing its deadline. Although J_2 missed its deadline, this scenario is deemed MC-schedulable. This is the case because J_1 executed for more than its C^N thus only HI criticality tasks are required to meet their deadlines.

It could be the case that at runtime job J_1 executes for only 6 units of time instead of 8. In this case J_2 still misses its deadline, but this scenario is not MC-schedulable since no job executed for more than its C^N, and in this case all jobs are required to meet their deadlines. This simple example shows the complications that arise in testing MC-systems. It gives one case of a policy of an MC-sustainable algorithm being able to correctly schedule a scenario but failing to schedule a better one.

5.2 Weak Predictability

Example 5.1 shows that unlike in single-criticality case, in an MC system, not all policies of an MC-sustainable algorithm are predictable. Another observation is that correctly testing the schedulability of a policy by simulating the worst case scenario does not anymore imply the schedulability of other scenarios. We expect that the CM policy will not be the only one that is not predictable. This is primarily due to the fact that decreasing the execution of a high criticality job, might stop the system from switching to HI-criticality mode, thus all jobs will be required to meet their deadlines, whereas before the decrease only HI jobs had to meet their deadlines. For this reason, we provide a weaker definition of predictability that takes the mode change into consideration and remains sufficient to perform the simulation based correctness tests proposed in the next sections.

Definition 5.2 (Weak Predictability)**.** An MC-scheduling policy is weakly-predictable if for any scenario that is MC-schedulable, decreasing the execution time of a job A – while keeping all other execution times the same – should not delay the termination time of any other job B under the following two conditions:

- If job A caused a mode switch, then the decrease in the execution of job A does not cancel the mode switch that was caused by A
- Job B terminates in the same criticality mode, before and after the decrease of execution of A

In a weakly-predictable policy, if at least one of the two conditions above is not met a decrease in the execution of one job is allowed to delay the termination of another job. Thus a weakly-predictable policy does not always have to be predictable. But a predictable scheduling policy is always weakly-predictable and all the results that follow from the weakly predictable property can be applicable.

The main intuition behind this weak definition of predictability is that it requires the system to "behave in a predictable way" only when a mode switch is not involved.

Thus MC-policies are more likely to be weakly-predictable and hence eligible to use the correctness test proposed for such policies.

In the work of *Socci et al.* [9, 50], a definition of predictability for the mixed criticality case was proposed and referred to as *predictable per mode*. In their definition, the predictability property had to be maintained only for jobs terminating in the same criticality level in both scenarios, *i.e.,* their definition is similar to the definition of weak-predictability but without the first condition. Socci's definition remains too restrictive for FPM policies, in the sense that not all of these policies are predictable per mode. Example 5.2 gives one FPM policy that is not predictable per mode.

Example 5.2. *Consider a dual-criticality FPM policy $\mathcal{P}$, that uses EDF to schedule the execution of jobs in the LO mode. Upon a mode switch, all LO jobs are dropped, HI jobs are scheduled using EDF as well.*

Let $\mathcal{I}$ be the instance described in the table below:

Job	A	D	χ	C^N	C^E
1	0	2	*HI*	1	2
2	0	3	*LO*	2	2
3	0	4	*HI*	1	2

For example 5.2, consider two scenarios of $\mathcal{I}$, c=(2, 2, 2) and c'=(1, 2, 2). The only difference between the two scenarios is that the execution of J_1 has been decreased by 1 for c'. Simulating the execution of c using $\mathcal{P}$, J_1 will execute for 1 unit of time, and since it does not signal termination a mode change will occur, dropping J_2. In HI-mode, J_1 will execute one extra unit and then J_3 will execute and terminate at $t = 4$. Knowing that all HI jobs met their deadlines, c is deemed MC-schedulable by $\mathcal{P}$.

As for the simulation of c', J_1 will execute for one time unit and will terminate. At $t = 1$, J_2 having the earliest deadline among the ready jobs, will be scheduled and will execute until $t = 3$, after which J_3 will execute for one time unit, cause a mode switch, and terminate at $t = 5$, in the HI mode and missing its deadline.

This gives an example of an FPM policy where a scenario is MC-schedulable, but decreasing the execution of a job (J_1 in this case), while keeping all other executions the same, delayed the termination of another job (J_3). Moreover, since J_3 terminates in the same criticality mode in both scenarios, this example provides an evidence of an FPM policy that is not predictable per mode.

We will show in the next section, that all dual-criticality FPM policies are weakly-predictable. Yet, before doing so, we need to formulate an equivalent definition to weak-predictability, where instead of comparing a scenario with another one that has *decreased* execution times, we will swap the two scenarios and give the definition in terms of *increased* execution times. We include the second definition here for clarity, since it will be used when we prove that FPM policies are weakly-predictable.

Weak Predictability (Second Definition). A scheduling policy is weakly-predictable if for any scenario that is MC schedulable, increasing the execution time of any job A – while keeping all other execution times the same – should not make any other job B terminate earlier only when the following two conditions hold:

- If job A did not cause a mode switch then the increase of its execution also does not lead it to cause the mode switch.
- Job B terminates in the same criticality mode before and after the increase of execution of A

The first condition can only be violated, if before the increase, A executed for at most C^N, and after the increase it is the first job to exceed the C^N thus causing a mode switch.

5.2.1 Weak Predictability in FPM Policies

The following theorem from [18] states a very useful property, for which we formulate two corollaries:

Theorem 5.3 (from [18])**.** *The fixed-priority (FP) policy is predictable for single- and multi-processor scheduling.*

Corollary 5.4. *For* single-processor *dual-criticality instances FPM is weakly predictable.*

Proof. Consider a dual-criticality FPM policy with given priority tables PT_{LO} and PT_{HI}. Consider any scenario c. For a given job A, let scenario c' differ from c only by an increase in execution time of job A by Δc_A, such that this increase does not lead A to be the job that causes a mode switch in c'. Let job B be an arbitrary job. We have to prove that B can only terminate at the same time or later in c' compared to c, but never earlier, provided that B terminates in both scenarios in the same criticality mode.

If there is no mode switch in c then predictability in this case follows from the predictability of FP scheduling. So let us first consider the case where A terminates after the mode switch in c. This means that the schedules of c and c' are the same up until the

switch time. At switch time the same LO jobs are dropped, if any, and both scenarios are left to execute the same jobs using same priority table PT_{HI} with only one difference, the increase in the execution of A. Thus it follows directly from the predictability of FP that in this case B will never terminate earlier in c'.

Secondly, we consider the case where both A and B terminate before the mode switch in c. If the increase in execution, leads B to terminate after the mode switch then, the condition that B terminates in the same criticality mode does not hold and we have nothing to prove. If after the increase in the execution of A, job B terminates before the mode switch then by predictability of FP it can not terminate earlier than in c.

Thus, the only non-trivial case that we have to consider is when in scenario c, job A terminates before the switch, executing entirely in the LO mode, and job B terminates in both scenarios after the mode switch.

Let t_c and $t_{c'}$ be the switch times of c and c'. Due to the predictability of FP scheduling, up until the switch time in c, no job other than A can exceed its C^N in c' before it does in c. And since A does not cause the mode switch, then we have $t_c \leq t_{c'}$.

Let t_A be the termination time of job A in c. Let t_B and t'_B be the termination time of job B in c and c' respectively. Since both terminate after the switch then we have $t_B > t_c$ and $t_{c'} > t_{c'}$. For $t_B \leq t_{c'}$ we have $t_B < t'_B$ and this completes the proof for this case. Thus we still have to prove for $t_B > t_{c'}$.

The execution of jobs in the interval $[0, t_A]$ is the same in both scenarios c and c'. Let us consider only the HI jobs that do not terminate by time t_c in c. In the time interval $[t_A, t_c]$, both scenarios are using PT_{LO} as the priority table. A executed for the same amount or more and FP is predictable thus all other HI jobs executed for the same amount or less in c' compared to c. Then in c' compared to c, every job has to execute for at least the same amount or more between t_c and its termination.

Let us make the following assumption for scenario c. Assume that between t_c and t_B there are no idle intervals and the (HI) jobs which execute there have equal or higher priority than B (w.r.t. PT_{HI}). We refer to this set of jobs as S_B. In addition let E_B be the execution time between t_c and t_B. This is the execution time needed for all jobs in S_B to terminate, as they all have higher priority than B and hence terminate before t_B.

Now let us consider scenario c'. Recall that at time t_c at least the same set of jobs have to execute the same amount of work as in c. Let us consider what happens after time t_c. Since $t'_B > t'_c$ and all jobs in S_B have equal or higher priority than B according to PT_{HI} then all other jobs in S_B terminate before B does. We are left with two cases:

- Either only jobs from set S_B execute between t_c and t'_c and in this case $t_B \leq t'_B$ follows from the fact that FPM is a work conserving policy and these jobs have at least the same amount of work to execute in c'.
- Jobs outside of S_B execute as well (having higher priority according to PT_{LO}) between t_c and t'_c but in this case we also have $t_B \leq t'_B$ from the argument on the amount of work to be executed.

Now we remove our earlier assumption for scenario c, instead we suppose that in the time interval $[t_c, t_B]$, there are one or more sub-intervals where the processor is idle or it executes jobs with lower priority than B according to PT_{HI}. We note that this can be the case only if during these sub-intervals there are no "ready" jobs in S_B to execute since B has the lowest priority in S_B and FPM is work conserving.

Let $[t_i, t_j]$ be the last subinterval where the processor was either idle or executing a job with lower priority than B. Thus in $[t_j, t_B]$, only jobs from S_B execute and we will refer to this set of jobs as S'_B. It is easy to see that none of the jobs in S'_B arrive before t_j.

Noting that in both scenarios, c and c', jobs in S'_B can not execute before t_j as they do not arrive before that time. Also,in both scenarios, all jobs in S'_B have to execute for the same amount, and terminate before B does since B terminates in HI mode in both scenarios and is scheduled by PT_{HI} when it terminates. In addition in c only jobs in S'_B are being scheduled in $[t_j, t_B]$ with no idle intervals included while in c' the same set of jobs are executed where B terminates last but also some other jobs might execute, thus B cannot finish earlier in c'.

□

Corollary 5.5. *For* single- and multi-processor *dual-criticality instances, an FPM policy that generates only FPM-equivalent tables is weakly-predictable.*

Proof. Consider a dual-criticality FPM policy with given priority tables PT_{LO} and PT_{HI}. Let scenarios c, c' and jobs A, B be defined as in the proof of Corollary 5.4. We have to prove that B can only terminate at the same time or later in c' compared to c, but never earlier, provided that B terminates in both scenarios in the same criticality mode. The same reasoning as before can be used to show that the only non-trivial case is that in scenario c job A terminates before the mode switch and job B terminates in both scenarios after the mode switch.

Let $\mathcal{S}$ be the schedule generated for c and $\mathcal{S}'$ the schedule generated for c'. Let t and t' be the mode switch time in $\mathcal{S}$ and $\mathcal{S}'$ respectively. By the predictability of FP, and the fact that job A does not cause the mode switch, we have $t \leq t'$. Since LO jobs are

dropped at switch time, only HI jobs that did not finish before the switch will execute after time t in $\mathcal{S}$.
Since FP scheduling is sustainable, no job other than A can execute in $\mathcal{S}'$ more than it executed in $\mathcal{S}$ up until t. Thus, the same set of HI jobs that have to execute after t in $\mathcal{S}$ will execute in $\mathcal{S}'$ and possibly jobs may require more execution time in $\mathcal{S}'$. In addition, some LO jobs may also execute after t in $\mathcal{S}'$ as they are dropped at t' with $t \leq t'$.

Note that as a consequence of having FPM-equivalent tables, using PT_{LO} instead of PT_{HI} after dropping the LO jobs at a mode switch, will not change the generated schedule, because the priority order of the HI jobs is the same in both tables. Thus for FPM policies having PT_{LO} and PT_{HI} FPM-equivalent, using any table after a mode switch results in the same schedule. Hence an FP scheduling policy that uses PT_{LO} to schedule the workload remaining after time t in c will generate the same schedule as $\mathcal{S}$. Also, using the same FP policy to schedule the workload remaining after time t in c' will generate the same schedule as $\mathcal{S}'$.

Since after time t the workload in $\mathcal{S}'$ is more than that in $\mathcal{S}$ and by the predictability of FP scheduling, then there is no such job B that terminates in $\mathcal{S}$ before $\mathcal{S}'$ after time t. □

Lemma 5.6. *An FPM policy that doesn't restrict its tables to be FPM-equivalent is not weakly predictable for multiple processors in general.*

Proof. Example 5.3 provides an FPM scheduling policy where PT_{LO} and PT_{HI} are not FPM-equivalent. We show that it is not weakly predictable. □

Example 5.3. *Consider the following 3-processor problem with instance $\mathcal{I}$described in the table below:*

Job	A	D	χ	C^N	C^E
1	0	6	*LO*	6	6
2	0	14	*HI*	4	5
3	6	15	*HI*	7	8
4	6	8	*HI*	1	2
5	6	9	*HI*	1	2
6	6	11	*HI*	3	4
7	6	13	*HI*	3	4
8	0	6	*LO*	6	6
9	0	7	*LO*	6	6

The Gantt chart in Figure 5.1 shows the execution in two scenarios: c and c' for an FPM scheduling policy with the priority tables specified in the figure where the property $PT_{LO} \sim PT_{HI}$ is not satisfied. In the time-slots, indexes 1,2,... 9 identify J_1, J_2, ... J_9.

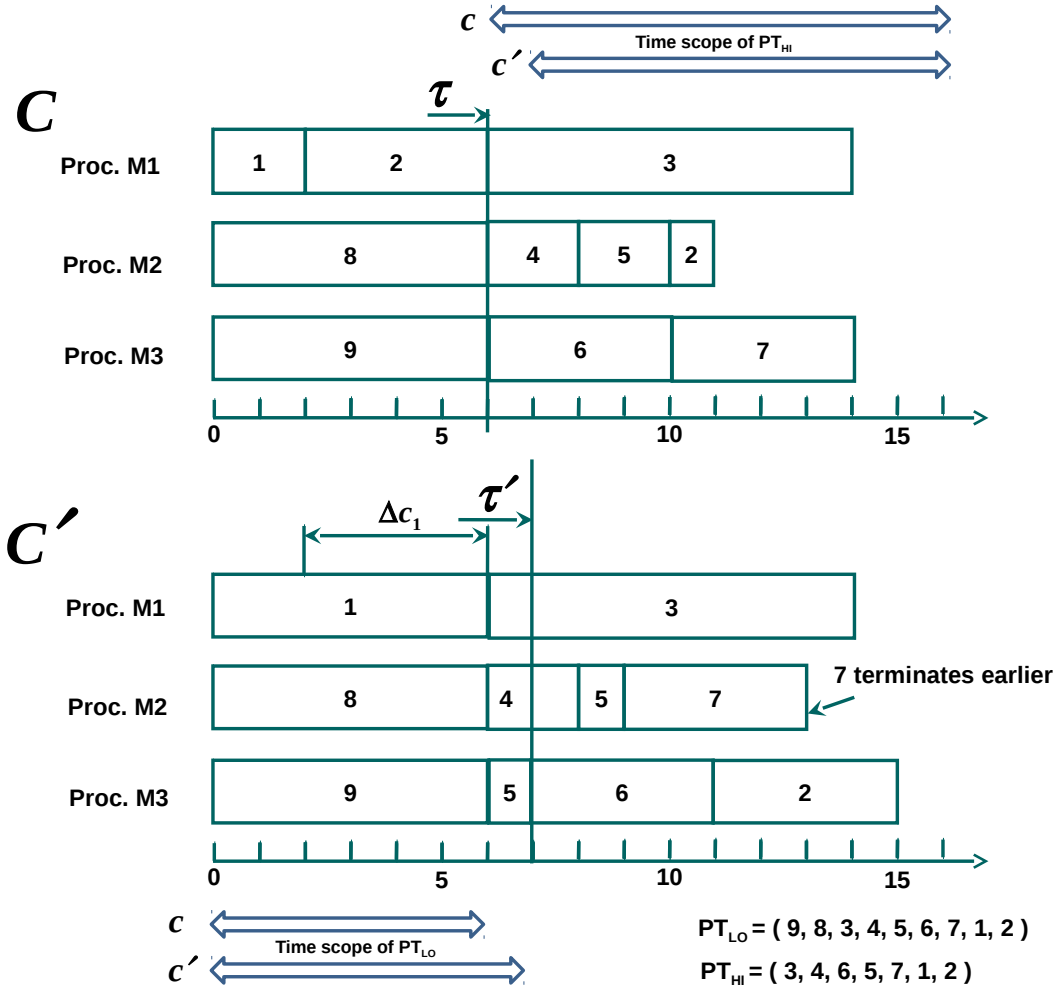


FIGURE 5.1: FPM non-predictable demonstration on multiprocessor case.

τ and τ', are the mode switch times in c and c' respectively. Scenario c is defined by $(c_1 = 2, c_2 = 5, c_3 = 8, c_4 = 2, c_5 = 2, c_6 = 4, c_7 = 4, c_8 = c_9 = 6)$. Scenario c' differs from c by $c'_1 = c_1 + \Delta c_1 = 2 + 4$.

The priority tables of the two modes in this example differ only by the relative priority of J_5 and J_6 and the window between τ and τ' is just one time unit. Nevertheless we see that job J_7 (as well as J_5) terminates in scenario c' earlier than in scenario c. This behavior contradicts the requirements of weak-predictability.

The previous example illustrates not just an exceptional case but well-known common properties of multiprocessor scheduling, differentiating them from single-processor case. Changing the order of job execution leads to a change of load distribution of different jobs between processors, which leads to different interference *w.r.t.* lower priority jobs. In our case, in window $[\tau, \tau']$ swapping the priority order between J_5 and J_6 has perturbed the load balance between the processors, such that a smaller priority job J_7 terminates

earlier. Note that in both priority tables the set of jobs that have higher priority than J_7 is the same and all of them arrive no later than J_7. Under the same conditions on single processor these jobs would inevitably have the same total interference on J_7 in the two scenarios, but not on multiple processors.

5.3 The Canonical Correctness Test

In order to adapt with the requirements of mixed criticality systems, fixed-prioirty scheduling policies have been extended to FPM policies. Unfortunately, because these policies support a mode switch, they become non-predictable in the usual sense, and a simple test of simulation in one scenario does not apply. One possible schedulability test for a fixed set of jobs is the examination of basic scenarios that should represent all corner cases of execution times. This test can be applied for fixed set of jobs offline, and, potentially, for task systems online at the moment when the job arrival times are known until a point when the processors become idle. In this section, we propose an adapted simulation-based schedulability test, that verifies the correctness of a scheduling policy and asserts a correct predictable behavior during runtime in case it is successful.

5.3.1 Basic Scenarios for Correctness Testing

Definition 5.7 (Basically Correct Policy [4])**.** A scheduling policy is **basically correct** for instance $\mathcal{I}$, if for any basic scenario of $\mathcal{I}$ the policy generates a correct schedule.

Lemma 5.8 (Correctness Test by Checking all Basic Scenarios)**.** *If a scheduling policy is weakly-predictable then the policy correctness follows immediately from its basic correctness. In other words, if the policy gives a correct schedule in all basic scenarios then this is also the case for the non-basic scenarios as well.*

Proof. For a given scheduling policy, let us call basic scenario $\lceil c \rceil$ the ceiling scenario of scenario c if in $\lceil c \rceil$ each J_i executes for time $C_i\left(\chi_{\text{TERM}}(c,i)\right)$, where $\chi_{\text{TERM}}(c,i)$ is the mode in which job J_i terminates in scenario c.

The plan of the proof is as follows. Let $\mathcal{I}_{\text{TERM}}(c,\chi)$ be the set of jobs that terminate in c in mode χ. We split the set of all jobs into $\mathcal{I}_{\text{TERM}}(c,\text{LO})$ and $\mathcal{I}_{\text{TERM}}(c,\text{HI})$. It is easy to show that by weak predictability all the jobs in the first subset will terminate in the LO basic scenario no earlier than in scenario c. In the rest of the proof we show that the other subset will terminate in the ceiling scenario, $\lceil c \rceil$, no earlier than in scenario c. Thus, the correct termination can be checked in the basic scenarios.

To prove for the second subset of jobs, suppose we could build a sequence of scenarios $c_1, \ldots c_m \ldots c_M$ such that $c_1 = c$, $c_M = \lceil c \rceil$ and we would obtain c_{m+1} from c_m by

increasing the execution time of some job "A" in such a way that this increase does not let "A" cause a mode switch. Suppose also that the jobs from $\mathcal{I}_{\mathrm{TERM}}(c, \mathrm{HI})$ would terminate in all scenarios c_m in the HI mode. By weak predictability this would lead to the required conclusion.

The first subsequence of the required sequence is obtained by iteratively taking a job from $\mathcal{I}_{\mathrm{TERM}}(c_m, \mathrm{HI})$ and increasing its execution time to C_j^E. In the second subsequence, we take the jobs from $\mathcal{I}_{\mathrm{TERM}}(c_m, \mathrm{LO})$ and increase their execution times to C_j^N. It is easy to show by induction that the resulting sequence satisfies the requirements.

All jobs in $\lceil c \rceil$ will also terminate in the same or higher-criticality mode. This statement is obviously true for jobs that terminate in the LO mode. At the switch to HI mode, all LO jobs are dropped thus all jobs that terminate in the HI mode are HI jobs. Since for HI jobs $C^N < C^E$, in the ceiling scenario such jobs will exceed their LO execution time. Therefore they will terminate in the HI mode. We show in example 5.4 that if we allow a HI criticality job to have $C^N = C^E$, a job can terminate in $\lceil c \rceil$ in a lower-criticality level than it did in c. For dual-criticality instances this implies that the jobs with $\chi_{\mathrm{TERM}}(c, i)$=HI terminate in the HI mode also in $\lceil c \rceil$. By the definition of weak predictability, these jobs cannot terminate in scenario $\lceil c \rceil$ earlier than in c. It remains to consider the jobs that terminate in LO mode in c. Obviously in the LO basic scenario these jobs cannot terminate earlier. Therefore the jobs of scenario c are "covered" by one of the two basic scenarios: $\lceil c \rceil$ and LO, in the sense that meeting the deadlines in those scenarios implies meeting deadlines in scenario c.

□

Example 5.4. *Fig. 5.2 shows two scenarios for a job instance that allows WCETs of HI jobs to be equal:*

Job	A	D	χ	C^N	C^E
1	0	2	*LO*	2	2
2	0	5	*HI*	1	2
3	2	3	*HI*	1	1

Keeping in mind that FPM policy, which we apply here, is weakly-predictable, we have two important points to observe in this example. First, observe that J_3 terminates in the HI mode in scenario c while it terminates in the LO mode in its ceiling scenario $\lceil c \rceil$. Thus, if the two WCET estimates of a HI job are allowed to be equal, weak-predictability is not always sufficient to ensure that the basic scenarios cover all other scenarios. Second observation is that this instance has only two basic scenarios, one is $\lceil c \rceil$ and the other one is exactly the same but with job J_2 terminating at t = 4 instead of switching to execute up until t=5. In both basic scenarios the instance is schedulable

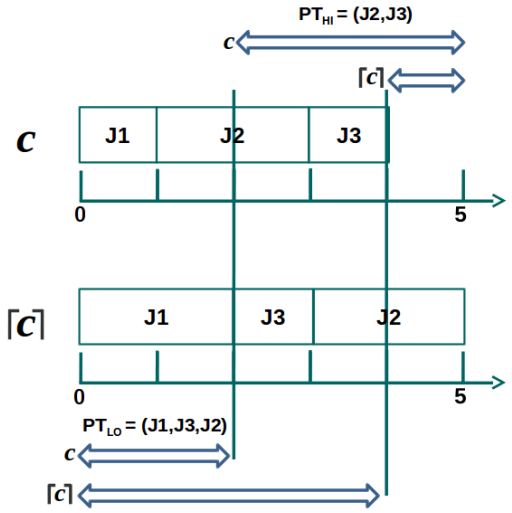


FIGURE 5.2: A caption

by the given FPM policy. Yet in c job J_3 misses its deadline. This shows that in the case where a high criticality job is allowed to have its $C^N = C^E$, even if all possible basic scenarios are simulated and successfully scheduled, this might not be enough to ensure the correctness of the scheduling policy with respect to all possible runtime behaviors. Due to this complication and the fact that we believe that disallowing HI jobs to have equal WCET does not limit the model as it can be ensured by an arbitrarily small increase of C^E, we decided to include this restriction in our problem formulation.

Lemma 5.9. *An instance $\mathcal{I}$ is MC-schedulable if it admits a basically correct scheduling policy.*

The above lemma is Lemma 1 from [4]. At first glance, it seems to be contradicting to the observations we just made in Example 5.4, where the basic scenario coverage is not sufficient for FPM schedulability, but it should be noted that the lemma only claims that a correct policy exists, not that this policy is FPM. In the proof given in [4] they show a simple procedure to transform any basically correct policy into a similar policy that is, in addition, also predictable, thus, by Lemma 5.8, yielding correct schedules in non-basic scenarios as well. The above lemma implies that a complete correctness test can be reduced to testing all basic scenarios. However, this would be inefficient, as there is an exponential number of basic scenarios. Fortunately, testing in all basic scenarios is redundant. We show in the next subsection that to test a weakly-predictable policy for a dual-critical instance it suffices to simulate $H + 1$ basic scenarios, where H is the total count of HI jobs in the problem instance.

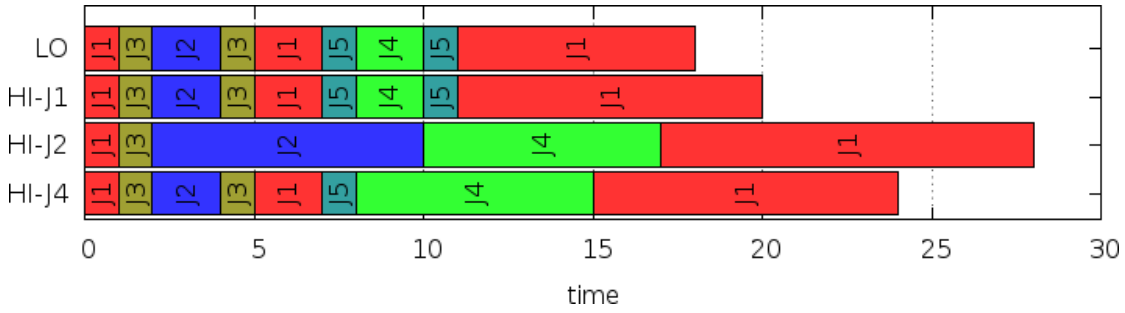


FIGURE 5.3: The job-specific scenario schedules for Example 5.5 obtained with priority table $PT = (J2, J4, J3, J5, J1)$

5.3.2 The Canonical Correctness Test (CCT)

Definition 5.10 (Job-specific Basic Scenario)**.** For a given problem instance, a scheduling policy and a HI job J_h, the job-specific basic scenario for job J_h is denoted by $HI\text{-}J_h$ and defined as follows. Job J_h executes for its C^E. For any other HI job, if it terminates in the LO basic scenario schedule $\mathcal{S}^{LO}$ before J_h terminates, then it executes for its C^N else it executes for its C^E. The schedule for $HI\text{-}J_h$ is denoted by $\mathcal{S}^{HI\text{-}J_h}$.

In multiprocessor scheduling, for a given job J_h multiple jobs may also terminate *exactly* at the same time in $\mathcal{S}^{LO}$ as J_h, and they are conservatively assumed to also execute for their C^E in $HI\text{-}J_h$.

Definition 5.11 (Canonical Basic Set)**.** It is the set that contains the LO basic scenario and the job-specific basic scenarios for all HI jobs of the given instance.

Note that $\mathcal{S}^{HI\text{-}J_h}$ coincides with $\mathcal{S}^{LO}$ up to the time when job J_h switches, and after the switching time it starts using HI execution times for the jobs that did not terminate before the switch.

Example 5.5. *Figure 5.3 shows Gantt charts for the job-specific scenarios of the single-processor problem instance given in the table below:*

Job	A	D	χ	C^N	C^E
1	0	30	*HI*	10	12
2	2	10	*HI*	2	8
3	1	8	*LO*	2	2
4	8	17	*HI*	2	7
5	7	11	*LO*	2	2

If we look at the termination times of HI jobs in $\mathcal{S}^{LO}$ (the schedule for the LO basic scenario) we see that J_2 finishes first followed by J_4 then J_1. Thus in scenario $HI\text{-}J_4$, J_2 will execute for its C^N since it terminates before J_4 in $\mathcal{S}^{LO}$ and the rest of the jobs will execute for their C^E resulting in the schedule shown for $HI\text{-}J_4$ in Figure 5.3. The rest of the job-specific schedules are generated in the same manner.

Theorem 5.12 (Canonical Correctness Test)**.** *To ensure correctness of a scheduling policy that is weakly-predictable it is enough to test it for the canonical basic set.*

Proof. Consider any basic scenario c and simulate the policy until either the mode switch, if any, or the end of the schedule. Let J_h be the job that switches. After the switch, increasing the job execution times can lead only to non-decreasing termination times, therefore we can conservatively replace c by $HI\text{-}J_h$. Hence, the policy is basically correct, and, by Lemma 5.8, also (completely) correct. □

Unfortunately, since by Lemma 5.6, FPM is not weakly-predictable in multiprocessor case and the canonical correctness test might not be sufficient under such general conditions, in this case by Corollary 5.5, we may need the FPM policy to have FPM-equivalent tables to obtain weak-predictability and use the correctness test.

5.3.3 Building the Case for Class NP for FPM

The *canonical correctness test* algorithm was directly derived from the correctness test procedure described in [4], used in an attempt to prove that MC-scheduling is in class NP. Note that their algorithm is more complex and more general, as it applies to a number criticality levels more than two. Though that procedure, for efficiency reasons, would organize the schedules of basic scenarios in a tree structure and use backtracking, our less efficient formulation has only polynomially higher complexity, which does not impact on the reasoning on NP complexity. We now reapply the line of reasoning of [4] to prove that FPM is in class NP. Although in Chapter 4, Lemma 4.1 [4] was refuted as it does not hold for all MC polices, it is true by construction in the case of FPM policies. We use this lemma in the proof of Theorem 5.13.

Theorem 5.13. *Dual-criticality single processor FPM policies are in class NP.*

Proof. It follows from the weak-predictability of single processor FPM polices and Theorem 5.12 that we can check for correctness by simulating a polynomial number of scenarios (the canonical basic set). By Lemma 4.1 the cost of simulating each scenario is also polynomial. Therefore the canonical correctness test has polynomial complexity when testing FPM solutions. □

Theorem 5.14. *Under the restriction of having FPM-equivalent tables, dual-criticality single- and multi-processor FPM policies are in class NP.*

Proof. Follows directly from Corollary 5.5 and Theorem 5.12 □

Note that unlike the case of Theorem 5.13, the case described in Theorem 5.14 is not known to be NP hard. We have established the upper bound as NP, but the lower bound in this case is an open problem, it may be either NP-hard or P, and it may differ for single- and multi-processor cases.

5.4 The Economical Correctness Test

The Canonical Correctness Test (CCT) simulates a linear number of scenarios. In this chapter we propose the Economical Correctness Test (ECT) that needs to simulate just two scenarios under certain restrictions. ECT transforms the policy into an 'equivalent' STTM policy which requires only two tables to be tested instead of $H+1$ as is the case in CCT.

Although ECT is only applicable for reasonable policies, this is not a serious limitation, because it does not remove optimality in dual criticality scheduling. This is due to the fact that EDF is an optimal scheduling policy for ordinary (non mixed-criticality) single-processor problems, and after a mode switch, it is this problem that remains to be solved online. It is worth mentioning that any non-reasonable policy can be transformed into a reasonable policy by simply changing its behavior to use EDF after a mode switch. Due to the optimality of EDF the transformed policy is either 'equivalent' or it dominates the original. Thus we expect that most scheduling policies should be reasonable or proven equivalent. The aim of this chapter is to prove that ECT test is equivalent to CCT and to study its algorithmic complexity. We start by a simple example to show that finding an equivalent STTM policy is not a trivial task. After that, we describe an approach, taken from the work in [51], to generate the STTM tables used in the proposed correctness test, prove its correctness in the general case and study its complexity in the case of FPM policies. All the theorems and proofs in this chapter are for single processor dual-criticality MC-Systems.

5.4.1 A Non-Trivial Problem

The transformation algorithm, denoted $\mathcal{T}(\mathcal{P})$ transforms a reasonable scheduling policy $\mathcal{P}$ to an STTM policy by using simulation augmented with additional rules for enabling/disabling jobs. Before we describe the algorithm in details, an example is given [51], to show that transforming a policy into an STTM policy having only two tables is not trivial. As seen in section 5.3, as many tables as there are HI jobs, might have to be tested, to ensure the schedulability of all scenarios.

Example 5.6. *Let us consider the following instance as an example:*

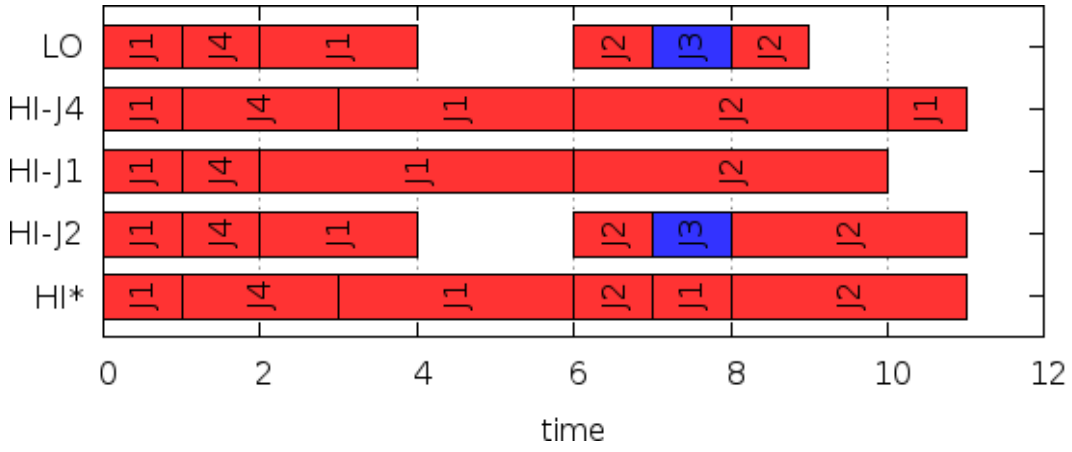


FIGURE 5.4: Canonical basic set and the HI* table

Job	*A*	*D*	χ	C^N	C^E
1	*0*	*12*	*HI*	*3*	*5*
2	*6*	*11*	*HI*	*2*	*4*
3	*7*	*8*	*LO*	*1*	*1*
4	*1*	*4*	*HI*	*1*	*2*

and the following FPM priority assignment for it (which can be computed using MCEDF[50]):

$$PT_{LO} = J_3 \succ J_2 \succ J_4 \succ J_1$$
$$PT_{HI} = J_2 \succ J_4 \succ J_1$$

Fig. 5.4 shows the schedule for the LO scenario and the HI job-specific basic scenarios for the instance in Example 5.6. The top most schedule represents the LO scenario, and it can be used as T_{LO} for the generated STTM policy.

However, none of the schedules for the HI basic scenarios shown in Figure 5.4 can be used to schedule the HI-mode for the generated policy. If either of the schedules for HI-$J1$ and HI-$J4$ is used, then J_2 will not have enough time to complete if a mode switch occurs at $t = 9$, since in both schedules it is given one time unit to execute between 9 and 10, but it needs two to complete. If the schedule for HI-$J2$ is used, then J_1 will miss its deadline if a mode switch occurs at $t = 4$. The correct **HI*** table thus can be different from all the ones generated for HI basic scenarios. The one for this example is shown in the bottom of the figure.

5.4.2 Generating the LO table

The LO table is generated by simulating the scheduling policy for the LO basic scenario. Although a scheduling decision can be taken at every time instant, for most policies

it can be restricted to just when certain events occur, for example a job arrival or termination.

To generate the LO table for an FPM policy one can simulate FP policy for criticality level 'LO'. The algorithm presented in Fig. 1 generates a time triggered schedule S by simulating the FP policy for the job instance $\mathcal{I}$ under priority table PT for criticality level χ'. In the case of FP, only two types of events are needed to efficiently simulate the execution of jobs, the arrival of a job and its termination. In Fig. 1, an event is represented by a 3-tuple containing the time of the event, a label to identify the type of the event, and a job. An example of an event can be $(t, `LBL\text{-}ARR', J)$ which states that job J arrives at time t.

In line 2 of the algorithm, jobs whose criticality level is lower than the criticality mode of the simulation are dropped. After that all arrival events are added to the priority queue of events Q_E, where events are sorted in ascending order with respect to the time of the event.

In addition to Q_E, the algorithm uses three other local variables. Q_P a priority queue that contains ready jobs sorted with respect to priorities in PT. J_{exe} used to store the job id that is scheduled to execute. In case the processor is idle J_{exe} is set to be $\emptyset$. $prgs$ - an array of type 'time', used to store the progress of each job. For example, if we have $prgs[i] = b$ this means that job J_i has executed for b time units. Throughout the algorithm a call to *SchedStart*(J, t_1, S) marks the beginning of a scheduling interval for job J at time t_1 in S, and the call to *SchedStop*(J, t_2, S) marks the end of the scheduling interval for the same job at t_2.

The main loop iterates over the events stored in Q_E. If there was a scheduled job, at line 12, the progress of job J_{exe} is updated to add the time passed between the last event and the current one. In the case of a termination event, the schedule S is signaled to stop the executing job at time instant '$time'$ and it is removed from Q_P. In the case of an arrival event, the arriving job is added to Q_P. Between lines 22 and 30 the job that has the highest priority in Q_P is scheduled, if another job was scheduled by a previous event then it is stopped by a call to *SchedStop*(). In the last part of the algorithm, it is checked if no event will preempt the scheduled job, a termination event is added at the appropriate time.

Observation 1 (Ordering of Schedule Events). We implicitly assume that events with same time-stamp and different type are popped with the preference to the event types that are first mentioned in the switch case of the algorithm, in particular that events labeled as 'LBL-TERM' are always popped first if there are any for the current time stamp.

```
1  Input:  job instance 𝓘 ,  priority table PT ,  criticality χ
2  Output: schedule S
3  Local:  array [1..n] of "time type" prgs ,  executing job J_exe ,
4  priority queue Q_E, Q_P
5  J_EFF ← { J ∈ 𝓘 | J.X ≥ χ }
6  PQueuePushSet( Q_E, [ J_EFF, 'LBL-ARR' ], ArrivalTimes )
7  lastTime ← 0
8  J_exe ← ∅
9  while Q_E ≠ ∅ do
10 |  ( [ time, LBL], J ) ← PQueuePop(Q_E)
11 |  if J_exe ≠ ∅ then
12 |  |  prgs[J_exe] += (time − lastTime)
13 |  end
14 |  switch LBL do
15 |  |  case 'LBL-TERM' do
16 |  |  |  SchedStop( J, time, S)
17 |  |  |  PQueuePop(Q_P)
18 |  |  |  J_exe ← ∅
19 |  |  case 'LBL-ARR' do
20 |  |  |  PQueuePush( Q_P, J, PT)
21 |  end
22 |  if Q_P ≠ ∅ then
23 |  |  J_i ← PQueueHead(Q_P).Job
24 |  |  if J_exe ≠ ∅ ∧ J_exe ≠ J_i then
25 |  |  |  SchedStop( J_exe, time, S)
26 |  |  |  J_exe ← ∅
27 |  |  else if J_exe = ∅ then
28 |  |  |  J_exe ← J_i
29 |  |  |  SchedStart( J_i, time, S)
30 |  |  end
31 |  |  termTime = C_i^N − prgs[J_i]
32 |  |  if X_i = χ then
33 |  |  |  termTime = C_i^E − prgs[J_i]
34 |  |  end
35 |  |  if termTime ≤ PQueueHead(Q_E).EventTime then
36 |  |  |  PQueuePush( Q_E, J_exe, 'LBL-TERM', termTime )
37 |  |  end
38 |  end
39 |  lastTime ← time
40 end
```

Algorithm 1: Simulation Algorithm for FP at Given Criticality Level χ

Observation 1 is needed to prevent the case where a job may terminate and get preempted at the same time.

Lemma 5.15. *The complexity of the off-line FP simulation for each mode is* $O(n(\log n))$*, with* n *being the number of jobs.*

Proof. The initialization of the priority queue is done in $O(n \log n)$ time. The main loop has $O(n)$ iterations, since there are $O(n)$ events. In every iteration, operations either have a complexity of $O(1)$, such as schedule operation, or $O(\log n)$ such as the priority queue operations. This results in a total complexity of $O(n(\log n))$. □

5.4.3 Generating the HI* Table

Whereas Fig. 1 can be used for generating the LO table for an FPM policy, we use a similar algorithm to generate HI^* for any reasonable policy, by simulating for $PT = EDF$ and $\chi = HI$ with some important modifications

- **LO** table is given as additional input
- we can disable some jobs, based on some 'rules'

A disabled job is intended to be hidden from the scheduling policy until it is enabled again. To that purpose we create a new queue Q_D to store disabled jobs. To disable a job, it is sufficient to remove it from the queue of ready jobs Q_P in Fig. 1, and store it in Q_D. By doing so, the scheduling policy will not schedule any disabled jobs. To enable a job again, it is simply moved back from Q_D to Q_P.

5.4.3.1 Transformation Rules

The intuition behind these transformation rules is to disallow a job from progressing in the HI* table for more than it has progressed in the LO table at any time t, unless it has been allocated its C^N on the LO table by that time. We define $T_j^{LO}(t)$ to be the cumulative execution progress of job J_j at time t in the LO table. Similarly, $T_j^{HI*}(t)$ is defined for the HI* table. HI jobs that execute for more than their C^N are referred to as *switched jobs*. A ready job J_j is considered *enabled*, and is placed in the ready queue, at time t, if at least one of the rules below is true:

$$T_j^{LO}(t) = C_j^N \tag{5.4.1a}$$

$$T_j^{HI*}(t) < T_j^{LO}(t) \tag{5.4.1b}$$

$$T_j^{HI*}(t) = T_j^{LO}(t) \wedge \ S^{LO}(t) = J_j \tag{5.4.1c}$$

The first rule enables all jobs that have switched. Rule 5.4.1b allows a job to execute at time t, if it has executed for more in the LO table up until t. Rule 5.4.1c enables a job

to be scheduled at HI* at time t, in case it has been allocated the same time on both tables, and is scheduled to execute at t in the LO table.

5.4.3.2 An Example

Looking back at Example 5.6, we will describe how HI* was generated. First at $t = 0$ no job has executed and thus the first two transformation rules can not be satisfied. However rule 5.4.1c is true for J_1 at $t = 0$, since it is being scheduled on LO at that time, and its total progress in both tables is zero. Thus J_1 is scheduled at $t = 0$. Following the same reasoning, J_4 is scheduled at $t = 1$. At $t = 2$, J_4 executes up until its C^N in the LO table and is enabled by the first transformation rule and J_1 is enabled by the third rule. Since J_4 has higher priority it is scheduled at $t = 2$. At $t = 3$ only J_1 is enabled and is scheduled until $t = 6$ where J_2 becomes enabled by rule 5.4.1c. Having higher priority J_2 is scheduled at $t = 6$ but only until $t = 7$ as it is no longer enabled at that time. J_1 is scheduled until $t = 8$ where J_2 becomes enabled first by rule 5.4.1c, and later at $t = 9$ by rule 5.4.1a.

5.4.3.3 The FPM HI* Table

The simulation in Fig. 1 can be modified to generate the **HI*** table while keeping its algorithmic complexity the same. To be able to apply the 'transformation rules' in the simulation the algorithm needs the already constructed **LO** table and keeps track of the progress of jobs not only in the HI mode, but also in the **LO** table.

Three events need to be added to the simulation in Fig. 1. The first two events,'LO-START' and 'LO-STOP', mark the start and the termination of execution times in S_{LO} for HI jobs. For Example 5.6, looking at the *LO* schedule in Fig. 5.4, the events for J_1 are $(0,\ \mathit{LO\text{-}START}, J_1)$, $(1,\ \mathit{LO\text{-}STOP}, J_1)$, $(2,\ \mathit{LO\text{-}START}, J_1)$ and $(4,\ \mathit{LO\text{-}STOP}, J_1)$. These events allow the algorithm to keep track of how much each HI job has executed in S_{LO}, in order to decide if a job is enabled or not. For generating the HI* table we no longer need to store arrival events. Instead, all jobs are assumed to be disabled at the start. Each disabled job is enabled and moved to Q_p whenever a $\mathit{LO\text{-}START}$ event occurs. The third event to be added is 'DISABLE'. This event is generated the same way as 'LBL-TERM' is generated *i.e.,* when a job is executing, but allowing the job to execute until it reaches either its execution progress in S_{LO} if it is not already executing in S_{LO} at that time, or until it reaches the next event. If a job's 'DISABLE' event is triggered, that job is stopped and it is removed from the ready queue Q_P. The job will be enabled again when its '$\mathit{LO\text{-}START}$' event is triggered. At every event we check to see if a job needs to be disabled before the next event. As we show in the lemma below, this modification does not increase the algorithmic complexity of the simulation.

Lemma 5.16 (Complexity of Transformed FP Scheduling)**.** *Given a fixed priority from the original policy and a time-triggered table for the LO scenario generated by Algorithm 1, the algorithmic complexity of the transformed simulation used to generate HI* is* $O(n(\log n))$

Proof. The schedule generated for the LO mode in Algorithm 1 has a polynomial number of arrivals, preemptions and terminations of jobs. As a result there are at most $O(n)$ '*LO-START*', '*LO-STOP*' and 'DISABLE' events. Thus the number of main-loop iterations remains unchanged compared to the non-modified simulation, with a maximum operation cost of $O(log\ n)$. □

5.4.4 Proof of Correctness

In this section, we provide the two theorems from [51, 52], which together show that, a reasonable weakly-predictable policy can correctly schedule an instance *iff* the instance can be scheduled by the transformed policy.

Theorem 5.17 (Transformation Correctness)**.** *For a given problem instance, if the original policy* $\mathcal{P}$ *is correct and reasonable then the transformed policy* $\mathcal{T}(\mathcal{P})$ *is also correct.*

Theorem 5.18 (Reverse Correctness)**.** *For a given problem instance on single-processor, under the assumption that the original policy is reasonable and weakly predictable, we have that if the policy* $\mathcal{T}(\mathcal{P})$ *is correct then the original policy is correct as well.*

For completeness, the proof of correctness of the theorems is provided in Appendix A.

5.4.5 ECT - Correctness and Complexity

Corollary 5.19 (Correctness testing with Economical Correctness Test (ECT))**.** *For dual-criticality single-processor problem instances, given a reasonable weakly predictable policy, testing the correctness of the transformed policy constitutes a necessary and sufficient correctness test.*

This result follows directly form the theorems in section 5.4.4. Testing the correctness of the transformed policy can be easily achieved, by testing if jobs in the generated **LO** and **HI*** tables meet their deadlines. This test is more efficient than the canonical correctness test proposed in the previous section since it needs to test only two tables instead of a number that is proportional to the HI jobs in the instance.

Corollary 5.20 (ECT Complexity)**.** *For dual-criticality single-processor problem instances, testing the correctness of FPM policies using ECT has an algorithmic complexity of* $O(n(\log n))$*, where* n *is the number of jobs in the given instance.*

The correctness of the corollary follows from Lemmas 5.15 and 5.16.

5.5 Chapter Summary and Contributions

In this chapter, we discussed subjects related to correctness testing for dual-criticality applications. The work presented, is an extension to the work of Socci *et al.* [51]. In the original work, 'predictability per mode' was defined for MC-systems. We showed, by means of an example, that the definition of predictability in [51], similarly to the definition of predictability in traditional scheduling theory, remains too restrictive for the case of mixed criticality. Indeed there are FPM policies that are not predictable following these definitions, which raises the question of the applicability of testing by simulation for these policies. Our contribution is defining the concept of weak-predictability as a less conservative property which we find more adequate for MC-systems, and proving that all members of the FPM class of policies are weakly-predictable.

Two correctness tests were presented in this chapter, the Canonical Correctness Test (CCT) and the Economical Correctness Test (ECT). These correctness tests were the results of a collaborative work between myself, Socci, Poplavko and Bensalem [52, 53]. A correctness test similar to CCT was used to test correctness in the MCEDF algorithm [17]. This test was assumed correct without giving any formal proof. In this thesis, for dual-criticality weakly-predictable policies, a correctness proof for CCT is provided, guarantying that it is applicable in the case of MCEDF, as well as all other FPM policies. As for ECT, the transformation algorithm was introduced in [51] and is not a contribution of this thesis. In this thesis, we contribute by showing that weak-predictability is necessary for the correctness proof of the test, more specifically to the proof of Theorem A.10. In addition, we provide simplified algorithms for generating the LO and HI* tables, since we only use them for the single processor case, whereas the original version of the transformation algorithm worked for the multi-processor case as well.

Chapter 6

Scheduling Systems with Multiple Levels of Criticality

In this chapter, we propose an STTM scheduling algorithm for scheduling systems with multiple levels of criticality for uniprocessor architectures. One aspect that makes our algorithm more versatile is that it assigns priorities to jobs' execution time-slots instead of assigning priorities to the jobs themselves. As a result, a single job can have different priorities at different time intervals throughout its execution. We call our algorithm Push Back Earliest Deadline First PBEDF.

Algorithm Overview. The proposed algorithm takes as input an MC-instance and tries to find a correct schedule for it. If successful, the algorithm will output a correct STTM schedule with one static time-table for every criticality level in the system. The main idea behind PBEDF is to delay the execution of lower criticality jobs, while assuring that they do not miss their deadline in scenarios of criticality equivalent to theirs or lower. This allows higher criticality jobs to execute earlier and gives the system more time to handle a mode switch in case it occurs.

The first step of PBEDF is to generate an initial time-table T_0 for criticality level 1. Next, it delays the execution of the lower criticality time-slots by pushing them backwards in the time-table. In addition to assuring that a job still meets its deadline on the time-table it is being pushed in, it should also have enough time to terminate before its deadline if a mode switch occurs. For dual-criticality systems, this is not a problem as only *LO* jobs will be pushed back, and these jobs are not required to meet their deadline after a switch. To satisfy this criteria for systems with more than two criticalities, a deadline is generated for every time-slot in T_0. This deadline is an indicator that if a time-slot

is pushed further it might miss its deadline after a mode switch. The generation of time-slots' deadlines is described in section 6.2.1.

After determining a safe bound that marks the time to which a job can be safely postponed, delaying the execution of a job is simply achieved by swapping one of its time-slots with another one belonging to a different job executing later in the time-table. In general, a swap is only allowed if it results in a higher criticality job executing earlier while giving time for the less criticality job to terminate before its deadline. More details on how time slots are swapped are found in section 6.1.2.2.

The modified time-table after the swapping of time-slots will constitute the final table for criticality level 1. We refer to this time-table as T_1. The last step of the algorithm is to generate time-tables for higher levels. This process is done iteratively, where each time table T_l is constructed from T_{l-1} for $1 < l \leq L$, L being the highest criticality in the system. Section 6.1.3 details the generation process of higher time-tables.

Motivation. The main motivation for PBEDF is it's ability to schedule mixed criticality systems of multiple levels of criticality and not just dual-criticality systems. As was shown in a survey by Burns *et al.* [54], the majority of work done in mixed-criticality scheduling limits itself to only two levels of criticality. According to the same survey, it was shown that for a scheduling policy to be of practical use, it needs to scale up to possibly four or five levels of criticality. In addition, jobs have dynamic priorities, even within the same criticality level. By dynamic priorities we mean that a job can have different priorities at different time intervals. This allows the scheduling of instances that are not schedulable by the widely used **fixed-priority** or **FPM** policies.

If successful, an STTM schedule is generated which is correct by construction and can be easily integrated into a system as shown in Chapter 7. Finally, experimental results show that for single processor dual criticality instances, PBEDF dominates a state of the art algorithm **MCPI**[9]. For more than two levels it far outperforms **OCBP**[13] as shown in section 6.3

Some Needed Definitions. Our schedule is represented by a set of time-tables. Thus, before we start we need to define the structure of a time-table and to define a time-slot. A **time-table** is a *sorted* set of *disjoint* time-slots. A time-slot can in general be represented by a tuple (t_1, t_2, J_i) which indicates an interval of time $[t_1, t_2)$ where J_i is scheduled to execute. For the purposes of our proposed algorithm we represent a time-slot by the tuple $(t_1, t_2, J_i, type, deadline)$ where *type* determines the type of execution, it can be either *certain* or *uncertain*, and the *deadline* represents the deadline of the time-slot, which can be the same as the deadline of the job or smaller.

Timetables are sorted in ascending order with respect to a slot's beginning time t_1. A constraint that we enforce in the structure of the time-table is that any two *consecutive* time-slots ts_1 and ts_2 in the table, such that $ts_1.t_1 < ts_2.t_1$ (ts_1 executes first) then we must have $ts_1.t_2 = ts_2.t_1$. We say that a time-slot is of criticality l if its job J_i is of criticality l. A time-slot is empty if $J_i = null$, this represents an idle interval in the processor.

6.1 PBEDF for Dual-criticality Systems

For a given dual-criticality problem instance, if successful, the algorithm generates two time-triggered tables T_1 and T_2 for criticality levels 1 and 2 respectively. We distinguish between three different phases of the algorithm where the output for each phase acts as the input for the next one. First, an initial time table T_0 for level 1 is generated. T_0 is not the final schedule for level 1, its role is to provide a starting point for the second phase. In the second phase, described in Section 6.1.2.2, T_0 is transformed into T_1, the final time-table for criticality level 1. Lastly, using T_1, the algorithm generates T_2 as shown in section 6.1.3.

6.1.1 Generating the Initial Time-Table

Before delaying *LO* criticality jobs in favor of pushing forward higher criticality ones, jobs must have initial priorities to begin with. Any single criticality fixed-priority algorithm can be used for this step. We choose EDF because of its optimality for single criticality instances in the uniprocessor case. An initial time-table T_0 is generated by simulating the execution of jobs following EDF priorities while constraining jobs to execute for exactly their C^N. Priorities in T_0 will act as the basis for generating the priorities in T_1, the time-table for criticality level 1.

6.1.2 Generating the Time-Table for Criticality Level 1

In this step, the algorithm tries to give higher criticality jobs higher priority by delaying the execution of lower criticality jobs. Algorithm 2 iterates over the time-table from its end to its beginning, starting from the time-slot that is before last. For each time-slot of criticality less than L, it calls the $pushBack()$ function that tries to delay the time-slot by swapping it with higher criticality time-slots.

For the dual-criticality case, L is equal to 2 and thus only *LO* jobs will be pushed back. But when scheduling systems with multiple criticalities, all jobs that are not of the

highest criticality will be delayed.

input : The timetable T_0
output: The timetable T_1 for level 1

```
1  if T_0.size() < 2 then
2  |  return
3  end
4  id ← T_0.size() − 2
5  while id ≥ 0 do
6  |  if T_0[id].job.crit < L then
7  |  |  pushBack (id, T_0)
8  |  end
9  |  id − −
10 end
11 T_1 ← T_0
```

Algorithm 2: Generating the time-table for criticality 1

The *pushBack*() function is shown in Algorithm 3. It is a recursive function that takes as input a time-slot and a time-table. Each call will loop over the time-table starting from the input time-slot towards the end of the time-table until it find another slot that is swappable with the input slot or until it reaches the end. If a swappable slot is found, i.e *canSwap*() returns true, then *swap*() is called. Based on the result of the swap one or more calls to *pushback*() are done. In order to understand the details of the *pushBack*() function, we start by describing two of its fundamental operations. The first is how to check if two slots should be swapped or not. The second is how to do a swap operation.

6.1.2.1 Swap Conditions

The objective of this step is to identify if two time-slots can be swapped without making any of them miss its deadline. In addition, a swap should only be allowed if it delays a lower criticality job so that a higher criticality job can execute earlier.

Given two time-slots $ts_i = (t_1, t_2, J_i, certain, d)$ and $ts_j = (t_1, t_2, J_j, certain, d)$ such that $ts_i.t_2 \leq ts_j.t_1$, the time-slots are swappable if both conditions 1 and 2 below are satisfied and either condition 3 or condition 4 is true.

The first condition is necessary to make sure that a job is not scheduled before its arrival time by simply making sure that ts_j has arrived before the end of ts_i. Condition 2 is also necessary to guarantee that a time-slot is not pushed after its generated deadline by checking that the deadline of ts_i does not come before ts_j. Condition 3 makes sure that the criticality of ts_i is less than that of ts_j. Condition 4 takes care of the care

Swap Conditions

1. $J_j.arrival < ts_i.t_2$
2. $ts_j.t_1 < ts_i.d$
3. $J_i.crit < J_j.crit$
4. $J_i.crit = J_j.crit$ & $J_i.deadline > J_j.deadline$

where both time-slots are of the same criticality. In that case we can allow the swap if the deadline of J_i is greater that that of J_j. Conditions 3 and 4 can never be both true.

6.1.2.2 The Swap Operation

In a swap, lower criticality jobs are swapped with higher criticality ones delaying the former and pushing forward the high criticality slots. This operation is only done on the initial T_0 to generate the final time-triggered table T_1 for criticality level 1.

Swapping time slots does not always swap the entire slots together.This is either because the slots are not of the same size or because of execution window constraints. A detailed description of the **swap**(ts_i, ts_j) operation is provided in this section, where ts_i and ts_j are two time-slots such that $ts_i.t_2 \leq ts_j.t_1$ i.e. ts_i executes before ts_j in the time table.

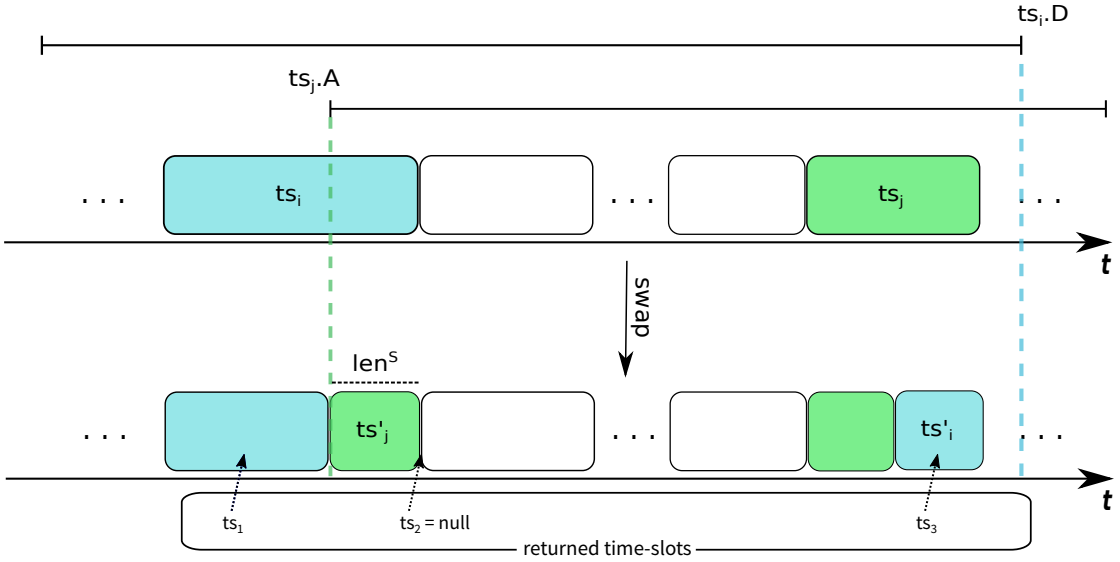


FIGURE 6.1: A swap example

Let $ts_i.A$ be the arrival time of the job executing in the timeslot ts_i. Similarly we will use $ts_i.D$ to denote the deadline of the timeslot ts_i. We say that a swap is a **perfect swap** if we have the following three conditions:

1. $ts_i.t_2 - ts_i.t_1 = ts_j.t_2 - ts_j.t_1$
2. $ts_j.A \leq ts_i.t_1$
3. $ts_i.D \geq ts_j.t_2$

In the case of a perfect swap the two timeslots are exchanged entirely. Otherwise the swap in non-perfect and some parts of one or both timeslots will stay in place.

Let $ts_i^{'}$, $ts_j^{'}$ refer to the modified timeslots after the swap operation as is shown in the example in figure 6.1. In the case of a non-perfect swap, the operation schedules $ts_j^{'}$ as soon as possible. To define $ts_i^{'}$ and $ts_j^{'}$ we need to determine the start and end of the interval of the timeslots.

- $ts_j^{'}.t_1 \leftarrow \max(ts_i.t_1, ts_j.A)$
- $ts_i^{'}.t_2 \leftarrow \min(ts_j.t_2, ts_i.D)$

$ts_j^{'}$ can start either from the start of ts_i or from its arrival time if it is greater than $ts_i.t_1$. $ts_i^{'}$ will end either at the end of ts_j or at its deadline if it is less than $ts_j.t_2$.
Let $len_i^{'}$,$len_j^{'}$ be the candidate lengths for $ts_i^{'}$ and $ts_j^{'}$ respectively.

- $len_i^{'} \leftarrow \min(ts_i^{'}.t_2 - ts_j.t_1, ts_i.t_2 - ts_i.t_1)$
- $len_j^{'} \leftarrow \min(ts_i.t_2 - ts_j^{'}.t_1, ts_j.t_2 - ts_j.t_1)$

$len_i^{'}$ is the estimated length of $ts_i^{'}$. If the distance between the end of $ts_i^{'}$ and the start of ts_j is greater than the length of ts_i then we can move all of ts_i to its new position else the new timeslot will have to start from $ts_j.t_1$.

$len_j^{'}$ is the estimated length of $ts_j^{'}$. If the distance between the start of $ts_i^{'}$ and the end of ts_i is greater than the length of ts_j then we can move all of ts_j to its new position else the new timeslot will have to end at $ts_i.t_2$.

The actual length of the swap denoted by $len^S = min(len_i^{'}, len_j^{'})$. Using it we compute the missing boundaries for the new timeslots:

- $ts_j^{'}.t_2 \leftarrow ts_j^{'}.t_1 + len^S$
- $ts_i^{'}.t_1 \leftarrow ts_i^{'}.t_2 - len^S$

6.1.2.3 The Push Back Function

The *pushBack*() function in Algorithm 3 takes a time-slot's id (sid_1) and a time-table (T_0) as input arguments and tries to delay the execution of sid_1 in the given table. It looks in the time-slots that are scheduled to execute after sid_1 searching for one which it can swap with. This is done by using the *canSwap*() function. *canSwap*() returns true if its arguments meet the swap condition defined in Section 6.1.2.1. If two time-slots can be swapped, the *swap*() function is called. This function works as detailed in section 6.1.2.2. After a swap is completed the function returns three time-slots which represent all possible fragments of the time-slot being delayed. We need these because the *pushBack*() function will try to delay these as well by recursively calling itself. The time-slots returned by this function are indicated in Figure 6.1 by ts_1, ts_2 and ts_3.

```
1  Function pushBack (sid_1, T_0)
2      for sid_2 ← sid_1 to T_0.size() − 1 do
3          if T_0[sid_1].job = T_0[sid_2].job then
4              return
5          end
6          if canSwap(T_0[sid_1], T_0[sid_2]) then
7              (ts_1, ts_2, ts_3) ← swap(T_0[sid_1], T_0[sid_2], T_0)
8              if sid_1 < T_0.size() − 1 then
9                  pushBack (ts_3, T_0)
10             end
11             if ts_2 is not null then
12                 pushBack (ts_2, T_0)
13             end
14             if ts_1 is not null then
15                 pushBack (ts_1, T_0)
16             end
17             return
18         end
19     end
```

Algorithm 3: The PushBack() Function

In the figure ts_i is the time-slot being delayed and ts_j (or a part of it) will be scheduled to execute earlier after the swap. If ts_j's job is not ready before or at the start of execution of ts_i, then part of the ts_i time-slot that executes before the arrival of the ts_j will not be swapped and will stay in place. This part is returned as ts_1. If ts_j's job is ready before ts_i starts then ts_1 will hold null. It could be the case where after a switch, a part in the end of ts_i, the slot to be delayed, did not get pushed back because the swappable part of

ts_i is bigger than ts_j. In such cases the last part of ts_i that stayed in place is returned in ts_2. If that is not the case, then ts_2 will be null as in Figure 6.1. ts_3 returns $ts_i^{'}$ the part of ts_i that has been delayed. ts_3 can never be null otherwise the swap conditions should have failed.

The *pushBack*() function will stop its search when it finds a time-slot it can swap with or when it finds another time-slot belonging to the same job as that of sid_1. The function stops when it finds another time-slot for the same job because this indicates that it can not be pushed further otherwise the found slot would have been pushed.

6.1.3 Generating Time Triggered Tables for Higher Criticalities

T_1 is used to schedule the nominal case where no job executes for more than its normal $WCET$, C^N. If the system switches mode to a higher criticality level, different time-tables might be necessary to schedule the system. To generate a time-table T_l for $2 \leq l \leq L$, the table T_{l-1} is used. In T_l, only jobs whose criticality is l or higher will execute, and jobs of criticality l will be allowed to execute for their C^E. Jobs of criticality higher than l will execute for their C^N.

For the scheduling of the normal scenario, the execution of any job can be known by looking at its reserved slots in T_1. This is not in general the case for T_l, $l > 1$. The system starts by using T_0, after which multiple mode-switches can happen leading to the use of other time-tables before finally reaching the point where T_l is used for the scheduling of the system. Thus, unlike the scheduling of the normal scenario, for higher criticality tables, the execution of a job can not be known by only looking at the table of the current criticality. Nevertheless, a higher criticality time-table is required to have sufficient execution time-slots reserved to all non-terminated jobs after a mode switch. To achieve this requirement, at any given time instant, jobs that did not exceed their C^N are constrained to execute in T_l for no more that the amount they execute in T_{l-1}.

Algorithm 4 shows how T_l is generated. It takes as input T_{l-1}, the lower criticality time-table and l the current level of the table to be generated. It works with the scheduling of time-slots instead of jobs. It will use two sorted lists whose elements are time-slots. *listAll*, which will contain all timeslots that should execute at T_l. Time-slots in this list are sorted by their start time i.e. $ts.t_1$. Also, not all jobs will be allowed to execute when they are ready, this is to enforce the property that a jobs in T_l is not allowed to execute for more than it does in T_{l-1}. We need another list to store the time-slots that are ready and allowed to scheduled on T_l. This list will be sorted by the deadline of the

time-slots and is referred to as *listReady*.

```
1  Function generateTable (T_l, T_{l-1}, l)
2      listAll.insert(T_{l-1}, l)
3      while not listAll.isEmpty() do
4          t ← listAll.first.t_1
5          J_h ← listAll.first.job
6          listReady.insert(listAll.pop())
7          if (not listAll contain execution for J_h) and J_h.crit == l then
8              ts ← TimeSlot(0, J_h.C^U, J_h, J_h.deadline, 'uncertain')
9              listReady.insert (ts)
10         end
11         if listAll.isEmpty() then
12             t_fin ← ∞
13         else
14             t_fin ← listAll.first.t_1
15         end
16         schedule (T_l, l listReady, t, t_fin)
17     end
18 return T_l
```

Algorithm 4: Generate Higher Criticality Tables

Initially both *listReady* and *listAll* are empty. $listAll.insert(T_{l-1})$ inserts the time-slots of T_l that are of criticality l or higher into *listAll*. The function keeps looping until all time-slots in *listAll* are scheduling in T_l. At line 4 in Algorithm 4, t is defined as the start time for inserting slots in T_l. This is the time the first slot in *listAll* is scheduled in T_{l-1}. At that time, the first time-slot in *listAll* is considered to be ready, it is removed from *listAll* and inserted in *listReady*. We define J_h to be the job executed by the moved time-slot. Afterwards, in the *'if block'* from lines 7 to 10, we check if all time-slots of job J_h are ready and have been moved out of *listAll*. If that is the case, and J_h is of criticality l then we add the uncertain execution time-slot of that job to *listReady*. In the last *'if block'* , we set t_{fin} to be either ∞, if there are no more time-slots in *listAll*, or it is set to the time when the next time-slot will become ready. Finally, the algorithm calls the *schedule*() function that inserts the time-slot in *listReady* into T_l between the time interval $[t, t_{fin})$. The jobs in *listReady* are scheduled on T_l between times t and

t_{fin}. The *schedule*() function is described in Algorithm 5.

```
1  Function schedule (T_l, l listReady, t_0, t_fin)
2      t ← t_0
3      while not listReady.isEmpty() and t < t_fin do
4          J_h ← readySlot.first.job
5          exe ← listReady.first.t_2 − listReady.first.t_1
6          uncert ← listReady.first.uncertain
7          if exe > t_fin − t then
8              exe ← t_fin − t
9              listReady.first.t_1 += t_fin − t
10         else
11             listReady.pop()
12         end
13         readySlot ← TimeSlot(t, t + exe, J_h, J_h.deadline, uncert)
14         T_l.insert (readySlot)
15         t ← t + exe
16     end
17 if t < t_fin then
18     T_l.insert (TimeSlot(t, t_fin, null, 0, certain))
19 end
20 return T_l
```

Algorithm 5: The schedule function

Since time-slots in *listReady* are sorted by their deadlines, the *schedule*() function inserts the time-slots from the list to T_l in the interval [t, t_{fin}) in EDF order. All the portions of the inserted time-slots are removed from the list. If the interval is larger than the total amount needed by all the ready jobs, the rest is filled by an empty time-slot.

6.1.4 Example

Example 6.1. *Table 6.1 contains an instance with 3 jobs, two of criticality level 2 and one job of criticality level 1.*

Job	A	D	X	C^N	C^E
1	0	5	2	2	3
2	1	3	2	1	2
3	0	3	1	1	1

TABLE 6.1: The no FPM job instance

Figure 6.2 shows the generated time-tables by PBEDF following the steps presented in this section. The temporary time-table T_0 showing at the bottom of the figure was generated by simulating the *LO* scenario using an EDF scheduling policy. At time $t = 0$, jobs J_1 and J_3 are ready. Since J_3 has an earlier deadline it is scheduled first. At $t = 1$, J_2 arrives and has the earliest deadline of non-scheduled jobs, thus it is scheduled in the interval [1, 2) and J_1 is scheduled last in [2, 4).

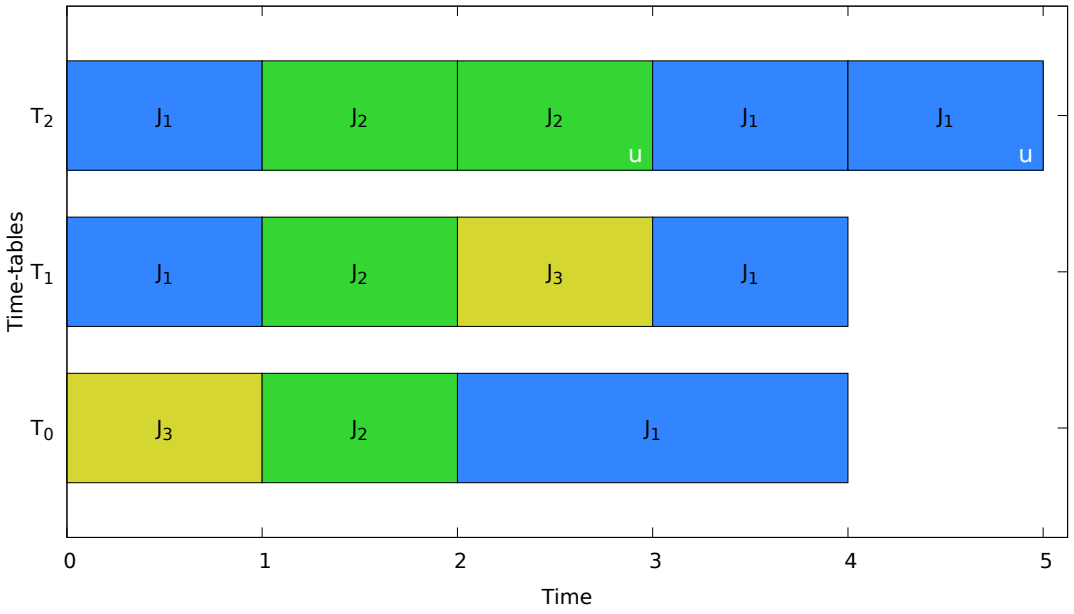


FIGURE 6.2: Schedule for instance in Table 6.1

It is worth noting that the schedule provided in T_0 does not correctly schedule the instance at level 1. Consider the scenario where J_2 and J_3 need to execute for their C^E in order to terminate. Following T_0 a mode switch is detected at $t = 2$ while J_2 does not signal its termination before that time. At that point, 4 time units are needed for the uncertain execution of J_2 and the execution of J_1. This means that both jobs cannot terminate before $t = 5$ and as a result one of them will miss its deadline.

We show how applying the push-back algorithm solves this problem. In this example, only J_3 will be pushed back since it's the only job who is of lower criticality. J_3 checks if it can swap with J_2. This swap is not possible because J_2 arrives at $t = 1$. Then J_3 checks if it can swap with J_1. The swap condition are valid for this case and J_3 can be pushed up until its deadline at $t = 3$. The result of the push back is shown in T_1 in the figure.

Last we need to generate T_2. At the start of this phase, *listAll* will contain 3 time-slots, two for J_1 and one from J_2 taken from time-table T_1. At $t = 0$, the first time slot for J_1 is added to *listReady*. Since it is the only one ready, it is scheduled in [0, 1) and removed from *listReady*. After that, the time-slot for J_2 is scheduled in [1, 2) as it will

be the only one available. At $t = 2$, a time-slot containing the uncertain execution of J_2 is added to *listReady*, as the condition of the *'if block'* at line 7 of Algorithm 4 are satisfied. At $t = 2$, the newly added time-slot is the only one available and thus it is scheduled at $t = 2$. At $t = 3$ the time-slot for J_1 is scheduled at T_1 and thus can be added to *listReady* to be scheduled followed by its uncertain execution. The constructed T_2 is represented in Figure 6.2.

It is easy to verify that the schedule represented by T_1 and T_2 is correct. In [20], Socci *et al.* proved that the example in Table 6.1 cannot be scheduled by any FPM policy. PBEDF is able to scheduled this instance because of the dynamic priorities property it has. In T_1, at $t = 0$, J_1 was given priority over J_3, while at $t = 2$, J_3 was chosen to be scheduled although J_1 was ready.

6.2 PBEDF for Multiple Criticality Systems

For $L > 2$, jobs of criticality higher than 1 can also be delayed. Delaying these jobs until their deadline, as we did before with LO criticality jobs, might not allow them enough time to execute for their C^E in case of a mode-switch. Thus, we need a way to know how much a job can be safely delayed. Keeping in mind that the execution of jobs is delayed only when generating T_1 where they are allocated C^N units of time, jobs of criticality greater than 1 should be allocated C^E units of time on the time table of the same criticality as their own. Thus for all tables with lower criticality level the actual deadline of a job J_i is at least $J_i.deadline$ - $J_i.C^U$. This will allow the job to have the additional C^U execution time to reach its C^E before its deadline, in case the system switched its mode to the job's criticality level. Sometimes the deadline of a job J_i has to be strictly less than $J_i.deadline$ - $J_i.C^U$, one example is the case of J_2 in example 6.2 shown later. To generate a more accurate value for a job's deadline on T_1, a deadline is generated for every slot indicating how much it can be delayed.

6.2.1 Generating Deadlines for Time-slots

The process of generating deadlines for the time-slots of T_1 is described in Algorithm 6. The main loop of Algorithm 6 iterates over the time-slots in T_1 from the last slot to the first one skipping all empty slots. Inside the main loop, the selected time-slot is referred to as ts representing the execution of job J_i. We also define l to be the criticality of

J_i.

input : The timetable T_0

output: The timetable T_0 with deadlines generated for slots

```
1  for ts ← T_0.last to T_0.first do
2      if ts is empty then
3          continue
4      end
5      l ← ts.job.crit
6      if l == 1 then
7          ts.deadline ← ts.job.deadline
8          continue
9      end
10     d ← ts.job.deadline
11     if T_l has no slots for job ts.job then
12         insertLate (T_l,ts.job,ts.job.C^U,d, 'uncertain')
13     end
14     for l ← L to 2 do
15         exec ← ts.t_2 − ts.t_1
16         d = min(d,insertLate (T_l, ts.job, exec, d, 'certain'))
17     end
18     ts.deadline ← d
19 end
```

Algorithm 6: Generate deadlines for slots

The *'if block'* starting at line 6 checks if $l = 1$. If that's the case it sets the deadline of ts to be equivalent to the deadline of J_i. This is valid because jobs of criticality 1 only execute for C^N on T_1.

In the *'if block'* on line 11, the algorithm checks if T_l, the time-table of criticality l, has any execution slots for job J_i. If none is found, then C_i^U of time units are inserted into T_l as late as possible before the deadline of J_i.

After that, on line 14, the algorithm loops over all timetables from T_l till T_2 and inserts ts units of execution to that timetable as late as possible before the time-slot's deadline. If the inserted timeslot finishes at a time that is earlier that its deadline, then the deadline of the time-slot is updated to be equal to the time the slot finished.

The method **insertLate(**T_i, *job*, *exec*, *deadline*, $TYPE$) takes five arguments and inserts one or more slots for the *job* given in time table T_i as late as possible starting from the *deadline* given. The total amount of execution for the inserted slots is equal

to *exec*. $TYPE$ is the type of execution it can be certain or uncertain. The method returns the end time for the first slots it enters in T_i.

Notice that although we are trying to insert the execution of one time-slot from T_1 into T_i. This can result in more than one time-slot inserted in T_i in the case that the nearest empty interval to the *deadline* given is smaller than *exe*. As a result we insert only a portion of *exe* and the rest is inserted earlier in the time-table. The return deadline for the *insertLate*() function is the end time of the latest inserted slot.

Also, the theorem below shows that adding or omitting the deadline generation phase in the dual-criticality case does not change the outcome of PBEDF.

Theorem 6.1. *For dual-criticality systems using deadlines in place of generated deadlines results in the same generated LO time-table* T_1*.*

Proof. When swapping two time-slots, we only look at the generated deadline of the slot being delayed to make sure that it will not be passed. In a dual-criticality scenario HI jobs are never delayed, only LO jobs are pushed back. For LO jobs the generated deadlines are equal to the deadlines. □

6.2.2 Example

Example 6.2. *Table 6.2 represents an MC-instance for a system with 3 criticality levels consisting of 4 jobs.*

Job	A	D	X	C^N	C^E
1	0	2	1	1	1
2	0	4	2	1	2
3	0	5	2	1	2
4	0	5	3	1	3

TABLE 6.2: A four-job instance

The time table at criticality level 1 in Figure 6.3 shows the initial table T_0 obtained from simulating the execution of the jobs with an earliest deadline first priority. We illustrate in this example how the time-slots' deadlines are generated following Algorithm 6.

The algorithm starts by generating the deadline for the latest time-slot in T_0 i.e. J_4. As J_4 is of criticality 3. The variable d is initialized to 5, the deadline of J_4. Since T_3 is initially empty, it does not contain any execution of J_4. Thus the method *insertLate*() is called to insert a time-slot to represent the uncertain execution of J_4 in T_3. This slot is inserted as late as possible but before d which is 5. Thus the time-slot is inserted in [3, 5).

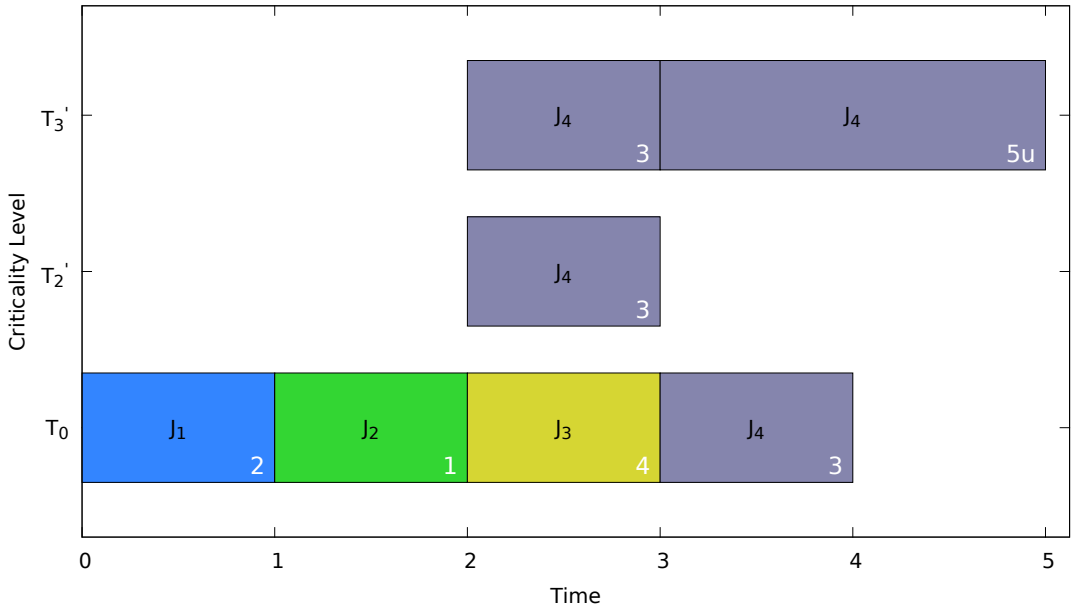


FIGURE 6.3: Generating deadlines for time-slots

Next the algorithm inserts one unit of execution, as late as possible before d, in time-tables T_3 then T_2, the loop at line 14 in the algorithm. In T_3 , the time-slot is inserted at [2, 3] which is the latest empty time interval before 5. As the termination of the new inserted slot is 3 which is smaller than the value of d. d is set to be 3.

As for T_2 the time-slot is inserted at [2, 3], which is the latest possible before d. d remains the same as the termination of the newly inserted time-slot is also 3. After we insert the time-slot in T_2, we exit the loop and the time-slot for J_4 in T_0 is updated to equal $d = 3$.

Second deadline to compute is for J_3. As it is of criticality level 2 and its deadline is 5, an uncertain time-slot is inserted at [4, 5] in T_2, and another one for the certain execution is inserted at [3, 4] right after J_4's slot. As the last slot inserted terminates at $t = 4$, d is updated to be 4 and the time-slot's deadline in T_0 is set to be 4.

In the same manner the deadline for the first two time-slots in T_0 is generated. The generated time-slots' deadlines are shown in the bottom right corner of the time-slots in Figure 6.3.

Applying the push back strategy on table T_1 in Figure 6.3, we start by looping over the time-slots in the table from the slot of job J_3 till the slot for J_1. Let ts_1, ts_2, ts_3 and ts_4 be the timeslots for J_1, J_2, J_3 and J_4 respectively in T_1. We find that the time-slots ts_3 and ts_4 are swappable as the first three rules are true. Thus we call $swap(ts_3, ts_4)$. Next we check if ts_2 can be pushed back. Since its deadline is 1 rule 2 will always fail and thus it cannot be pushed back. The final slot to check is ts_1. We check if it is swappable

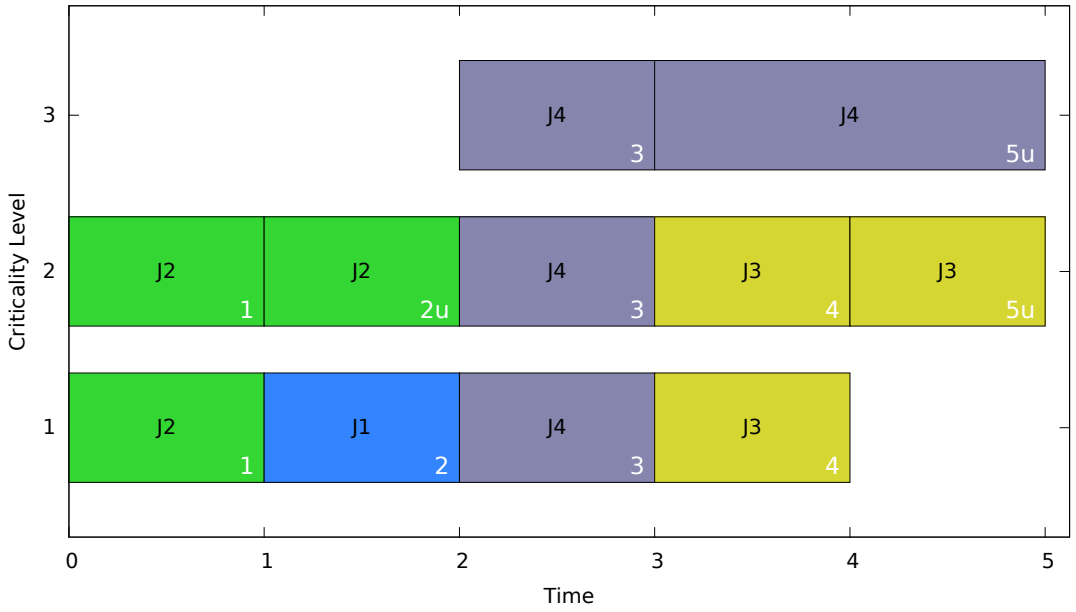


FIGURE 6.4: Swapping time-slots

with ts_2 and again the first three rules are true and we call $swap(ts_1, ts_2)$. After that rule 2 will always be false as ts_1 cannot be pushed behind $t = 2$. The final version of T_1 after the push back strategy is shown in Figure 6.4.

6.3 Experiment Results

To evaluate the performance of PBEDF, we test the percentage of correctly schedulable instances from a set of randomly generated ones at different load values. The *load* metric characterizes the maximum ratio between demand and capacity of the system [55], and for a given assignment of execution times c_i, it is defined by:

$$\ell oad(\mathcal{I}, c) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i \in \mathcal{I}:\ t_1 \le A_i \wedge D_i \le t_2} c_i}{t_2 - t_1}$$

Baruah *et al.* extended the definition of the load metric to mixed-criticality. In [56], the authors proposed the use of two separate load metrics one for the LO mode and the other for the HI mode as follows:

$$Load_{\mathrm{LO}}(\mathcal{I}) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i:\ t_1 \le A_i \wedge D_i \le t_2} C_i^N}{t_2 - t_1}$$

$$Load_{\mathrm{HI}}(\mathcal{I}) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i:\ \chi_i = \mathrm{HI}\ \wedge\ t_1 \le A_i \wedge D_i \le t_2} C_i^E}{t_2 - t_1}$$

For our experimental tests, since we have more than two levels of criticality, we will use the load metric below, which in the case of dual-criticality systems is equivalent to the metric defined by Baruah *et al.* .

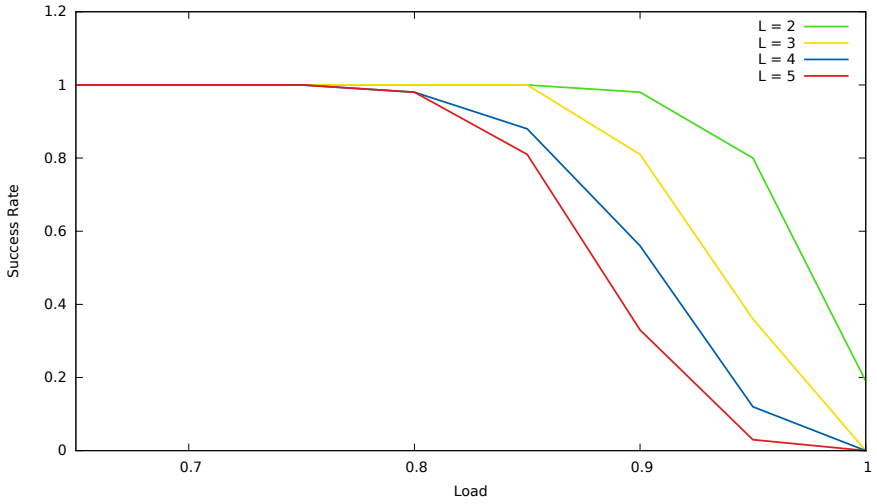
$$Load_\ell(\mathcal{I}) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i:\ \chi_i \ge \ell \ \wedge\ t_1 \le A_i \wedge D_i \le t_2} C_i}{t_2 - t_1}$$

where $C_i = C_i^N$ if $\chi_i > \ell$ and $C_i = C_i^E$ if $\chi_i = \ell$. This extension makes sense, since for a given criticality level, the time-triggered table has to schedule C_i^E for jobs of the same criticality and C_i^N for jobs that have higher criticality.

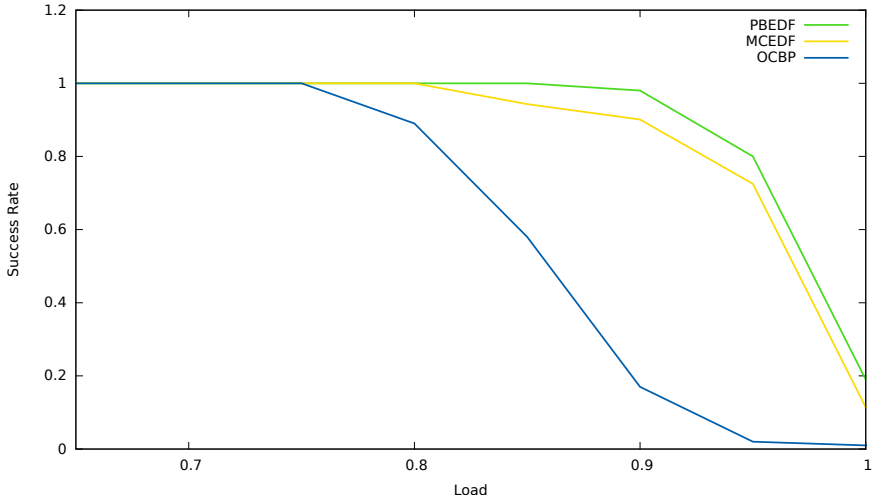
We compare the schedulability of PBEDF, with two scheduling algorithms, MCEDF and OCBP. Although MCEDF was proven dominant over OCBP, it only works for two criticality levels. A large number of problem instances were randomly generated for a given target load and a maximum criticality level L. The load values for all criticality modes are taken to be equal to the target load and move in the range of $[0.6, 1]$, with increments of 0.05. Values of L tested are taken from 2 to 5, representing systems with up to 5 criticality levels. For each L, and load values given, 100,000 random instances consisting of 20 to 100 jobs are generating.

Each of the three algorithms tries to schedule all the generated instances (for MCEDF only dual-criticality instances), the results are shown in Figure 6.5. In Figure 6.5(a), different schedulability values for PBEDF with different criticality levels are evaluated. As expected, the algorithm's success rate decreases when increasing the number of criticality levels. We see that it is able to schedule all instances with load less than 75%. For $L = 2$, the success rate decreases sharply when the load hits 95% , whereas it starts its sharp decrease at 85% load for instances with five criticality levels.

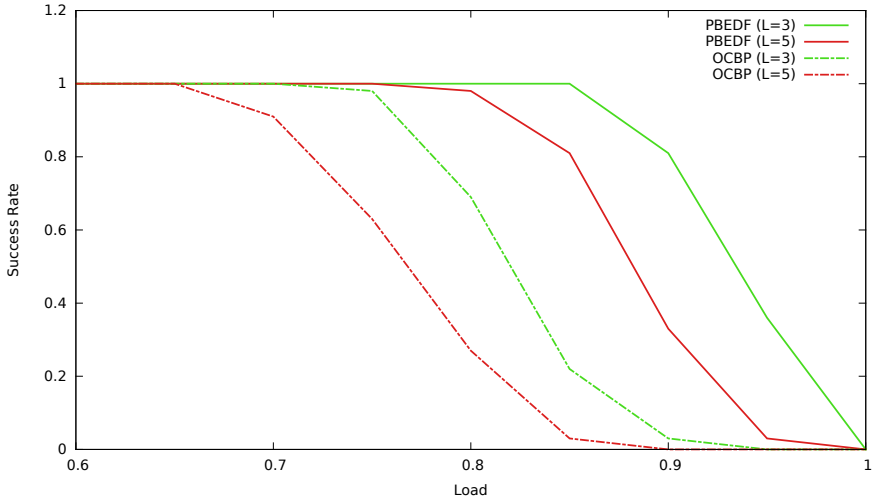
Figure 6.5(b), shows the schedulability of all three algorithms for dual-criticality instances. Tests indicate that PBEDF dominates both algorithms, as we were unable to find any instance that is schedulable by MCEDF but not by PBEDF. Yet this result remains experimental, and a theoretical proof is needed for confirmation. Figure 6.5(c), compares PBEDF with OCBP for instances with higher criticality levels. The solid lines represent PBEDF, and dotted lines for OCBP. The lines in green are for $L = 3$ and the ones in red are for $L = 5$. It is clear from the figure that PBEDF greatly outperforms OCBP. At 85% load, for three criticality levels, PBEDF is able to schedule 100% of the instances while OCBP schedules 22.1%. For the same load value with up to 5 criticality levels, PBEDF has an 81% success rate compared to 3.8% for OCBP.



(a) Schedulability of PBEDF for different criticality levels



(b) Comparison of different algorithms for $L = 2$



(c) Comparison of PBEDF and OCBP performance

FIGURE 6.5: Experimental evaluation of the schedulability of PBEDF

6.4 Chapter Summary

In this chapter, we presented PBEDF, an STTM algorithm for the scheduling of job instances of multiple levels of criticality. We showed that our algorithm can find solutions to instances that can not be scheduled using FPM policies. In addition, the effectiveness of the algorithm was evaluated by experimental results, comparing its ability to schedule randomly generated instances with two other algorithms, OCBP and MCEDF, considering up until five levels of criticality. In the next chapter, we give a component-based design for an MC-system that is able to detect faults and uses PBEDF as a recovery component, enabling it to transition back to a non-faulty state.

Chapter 7

Mixed Criticality Policies as Fault Recovery Strategies

Autonomous systems are gaining increasing popularity in research, as well as industrial applications in different domains such as health care, smart farming, drones and transportation (autonomous cars, railways and others). Since such systems are required to fulfill their tasks without the intervention of a user, autonomous systems should be able to make choices and even adapt in the case of unforeseen events. Unexpected circumstances can jeopardize the mission and/or the safety of the system, putting it in a faulty state that was not anticipated during the design of the system, its analysis and simulation of its interaction with its environment. A necessary functionality of autonomous systems in this case is the ability to detect failures and to recover, moving themselves back to a correct state.

In [6], *Dragomir et al.* proposed a design of FDIR components for autonomous systems. These components are generated in a systematic way and are proven correct by construction. In their work, FDIR components consist of two main subcomponents, diagnosers and controllers. Diagnosers' task is to detect a fault as soon as possible. Controllers are notified by the diagnoser that a fault has occurred and are responsible to take counter measures and bring the system back to a valid state.

In this chapter, we will show one way an STTM scheduling policy can be used as a recovery strategy alongside the FDIR components from [6]. Following in the steps of the work in [6], components of the system are modeled using TA [57]. In section 7.1, an overview of the whole system is given. In later sections, the generation of different components of the system is explained in details.

7.1 Overview of the System

As in previous chapters, the system's workload will be represented as a set of mixed-criticality jobs running on a uniprocessor platform. Yet, in this chapter, to be able to make use of the algorithms that generate the FDIR components, we will design a component-based system, where each component is defined by a timed-automaton. Each of the jobs and the scheduler will be transformed into TAs. It is the responsibility of the scheduler component to organize the execution of the jobs through signaling the start of execution and preemption. In addition, we assume that an executing job will notify the system of its termination once it finishes execution. The scheduler and the job components constitute the core of our system and they model its nominal behavior.

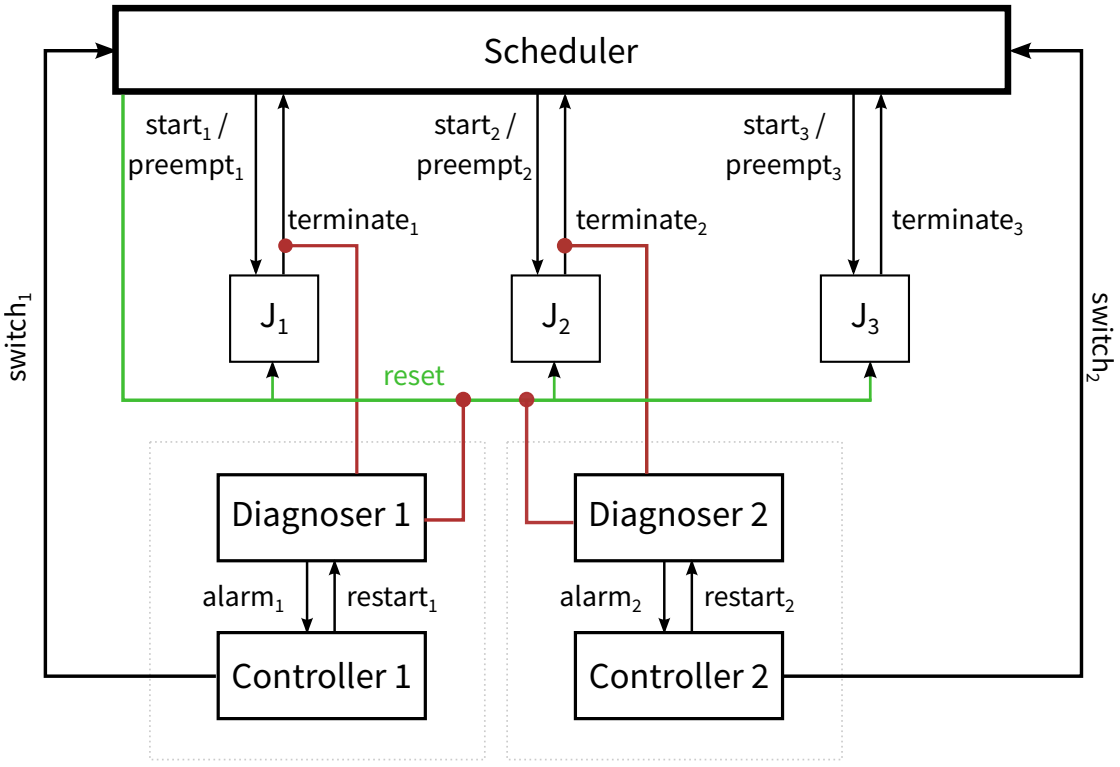


FIGURE 7.1: Overview of the whole system

In order to handle faults, two components are added for each job whose criticality is greater than one, a diagnoser and a controller. These components represent the FDIR module for that job, which is responsible for detecting a failure and recovering from it. The diagnoser monitors the termination of a job, and if it detects that a fault has occurred, it signals an alarm to the controller. The controller in turn, signals to the scheduler that there is a fault and a mode change should occur to handle the fault. After the fault has been dealt with, the controller resets the state of the diagnoser allowing it to monitor for additional faults and signifying that the detected fault has been resolved.

In our model, we consider that a fault has occurred during the execution of a job if it takes more than anticipated to finish. This is characterized by a job executing for its C^N without signaling termination. It is assumed that all faulty jobs will terminate before their C^E though. For consistency with our assumption, jobs of criticality level 1 where $C^N = C^E$ are considered to never fail.

Figure 7.1 shows an overview of the system in Example 7.1. J_3, being of criticality level 1, does not have an attached diagnoser/controller components as it is assumed to always complete without failure. The *reset* link and the red links coming out of the diagnosers are discussed in later sections.

Example 7.1. *Table 7.1 represents an MC-instance for a system with 3 criticality levels consisting of 3 jobs.*

Job	A	D	X	C^N	C^E
1	0	3	2	2	3
2	0	6	3	2	3
3	3	4	1	1	1

TABLE 7.1: MC-problem instance

7.2 Representation of Mixed-Criticality Jobs

To be able to make use of the diagnoser synthesis in [6], components need to be represented as timed automata. A timed automaton is an automaton augmented with a set of timed clocks. A TA is characterized by:

- A set of finite locations, of which one is defined as the initial location
- A set of real-valued clocks
- Guards which are constraints on clocks that are associated to locations
- A set of actions
- A transition relation

A transition moves from one location to another through the execution of an action. Transitions from a specific location are enabled and can be executed only if the guards at that location are satisfied. Actions in different automata, having the same name, indicate synchronized actions in the system. These actions can only occur at the same time when all of their transitions are enabled. Actions that are eager, *i.e.,* that should be fired as soon as possible, are identified by the ε symbol. Time passes at the same rate for all clocks in the system, but a set of clocks can be reset when a transition is fired up. In our design, all jobs of the same criticality level are represented with the

same automaton model, but jobs of different criticality levels need different automata representations.

7.2.1 Low Criticality Jobs

Jobs of criticality level 1 are the simplest, since they are assumed to never fail. To simplify the structure of the automata jobs are always assumed ready at t=0. This is not a problem because a job's execution is controlled by the scheduler that only calls a job to execute after its arrival time. Thus as shown in Figure 7.2 the job is initially in the "ready" state. Two possible interactions from the ready state, either the scheduler enables the start interaction and the job executes, or the job is reset. In the case a job starts and is executing, it can either be preempted by the scheduler, or will terminate no longer than $t = f_1$. f_1 marks the latest time this job is allowed to execute in T_1. Looking at Figure 7.6 we can detect that for J_3, $f_1 = 4$. Again by the correctness of the schedule all jobs of level 1 will finish in the end state. The loop from the end location to idle location and back to end is needed in case a job terminates before its C^N but the scheduler tries to schedule it. In that case the job goes to an idle location and terminates immediately as this action is eager.

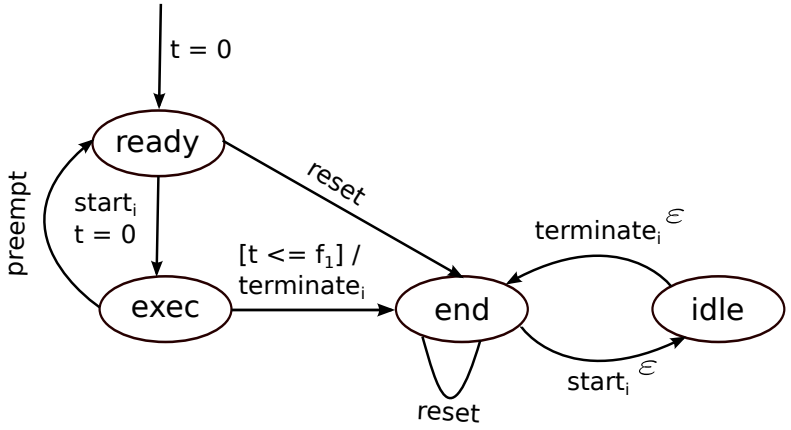


FIGURE 7.2: Automata for jobs of criticality 1

When in the ready state, a higher criticality job may fail causing a mode switch which in our model signals reset to all jobs. When a job of criticality 1 receives a reset it can be either in the ready state or the end state. If it is in the end state it stays there. If it is in the ready state it simply goes to the end state. This transition represents that the low criticality job has been dropped upon a mode switch.

7.2.2 Jobs of Criticality 2

For jobs of criticality level 2, the automata is shown in Figure 7.3. The core of the automata is the same constituting of the ready, exec, and end locations which are all

that is needed to model a fault-free execution. But in this automata we have two main differences.

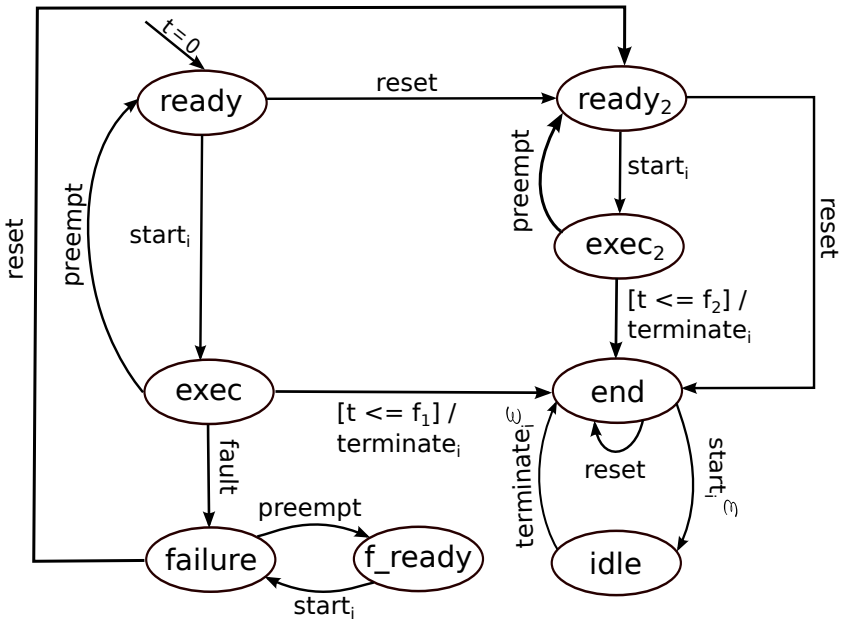


FIGURE 7.3: Automata for jobs of criticality 2

The first is that a job can fail at any time during execution. A fault will transition a system from exec to the failure location. In the case that a scheduler decides to preempt the job, and since the system did not detect the fault yet, the preempt will take the job to the f˙ready location putting the job in a faulty-ready state until it is started again.

The second difference is that in the case of a mode switch from level 1 to 2, the job is not dropped. Instead it goes to the $ready_2$ location, either by a transition from the failure location, in case the job has had a fault or from the ready location in case the mode switch was caused by another job. When we are in the $ready_2$ state, we reach a behavior similar to job of criticality 1. This is because at criticality level 2 jobs of this level are assumed to not fail thus they will start execution until preempted or terminated.

7.2.3 Jobs of Higher Criticality Level

Following the same steps used for creating the automaton for jobs of criticality level 2, automata for higher criticality jobs can be constructed. From each $ready_i$ and $failure_i$ locations we must have a reset to the $ready_{i+1}$ location, for $1 \leq i \leq \ell$ with ℓ being the criticality level of the job. We will also have ℓ number of *ready* and *exec* locations, and $\ell - 1$ $failure$ and f_ready locations with one *end* location. From each *exec* location, there will be a *terminate* action going to the end state, and from the $ready_\ell$ location there will be a reset action going to the end state. Although for jobs were $\ell = L$,

i.e., jobs who have the highest criticality, this last action will never occur but it can be kept for generalization. The automaton for jobs of criticality 3 is shown in Figure 7.4.

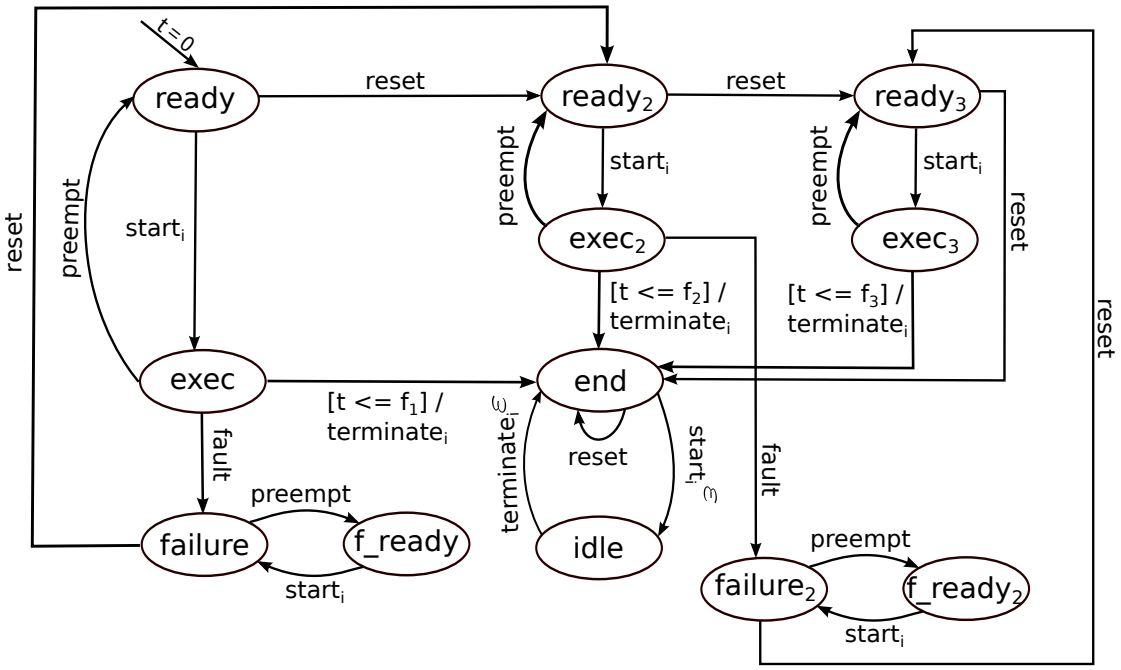


FIGURE 7.4: Automata for jobs of criticality 3

7.3 Fault Detection

Fault detection is the responsibility of the diagnoser component that runs in parallel with the system and decides whether a fault has occurred. One diagnoser is generated for every job whose criticality is higher than 1. These components are generated automatically following the diagnoser synthesis algorithm described in [6]. A brief description on the process of synthesis is presented below, for a more detailed explanation the reader is referred to [6, 58, 59].

7.3.1 Diagnoser Synthesis

The first step in the process is to create the **diagnosis model**, a modified copy of the system model. The modified model has the faulty locations marked. In our example, the *failure* and *f_ready* locations are marked as faulty. To detect the occurrence and propagation of faults in the system, a bit is associated with each location. The bit is set to zero if a fault has not occurred yet and one otherwise. The transition relation is also changed in the diagnosis model, a transitions from q_0 to q_1 will set the bit of q_1 to one if it is labeled with *fault* otherwise, it will set the bit of q_1 to the value of the bit of q_0.

During monitoring, actions are distinguished as either being *control* actions or *internal* actions. Control actions are the ones that influence the fault detection process. In Figure 7.1, the control actions are are *terminate* and *reset*, identified by a red link going from a diagnoser component to that action.

In [6], a set of current states of monitoring W is defined in addition to a Boolean variable indicating whether an alarm has been raised. Initially, W includes all the states that are reachable by only firing *internal* actions. If, at any time, the failure bit is set to 1 for all states in W then an alarm is raised. In all cases, the diagnoser continues monitoring and updating the set W. If a restart action is fired from the controller, then the failure bit is set to zero for all states in W and the alarm is turned off.

7.4 The Recovery Strategy

7.4.1 Controller

The second element in the FDIR component is the controller. In general, the controller's purpose is to perform the recovery operations once a fault is detected, and return the system to a correct state. In our design, the scheduler, described in the next section, will handle the recovery of the system. The controller simply works as an intermediate channel between the diagnoser and the scheduler. Once a diagnoser raises an alarm, the controller triggers the **switch** action that causes the scheduler to be notified that a mode change must occur. After that, the controller signals the diagnoser to restart and awaits for the detection of another fault. The TA of the controller is shown in Figure 7.5.

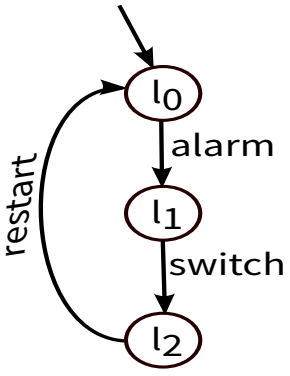


FIGURE 7.5: The controller automaton

7.4.2 Scheduler

The *scheduler* component has to orchestrating the execution of all the jobs when the system is in a correct state and should be able to recover the system in case of a failure. In this section, we present a method in which an STTM scheduling policy can be transformed into an automaton that can fulfill the requirements needed for the

scheduler component. For the transformation to work, it is assumed that, the scheduling policy only allocates the processor to jobs after their arrival time.

In the nominal case, the scheduler should provide every job with C^N execution time before its deadline. In case of failure, the controller will begin a mode switch, which will change the criticality mode of the system to a higher level. The scheduler should be aware of the change in criticality mode and is expected to change its scheduling policy accordingly.

In general, for a job J_i of criticality ℓ, and a given criticality mode χ_{mode} the schedule should provide at least the following guarantees for jobs:

- if $\ell < \chi_{mode}$, no guarantees for execution are given
- if $\ell = \chi_{mode}$, the job is guaranteed at least C_i^E execution time
- if $\ell > \chi_{mode}$, the job is guaranteed at least C_i^N execution time

The Gantt chart in 7.6 shows the schedule generated by PBEDF, the algorithm proposed in Chapter 6 for the instance of Example 7.1.

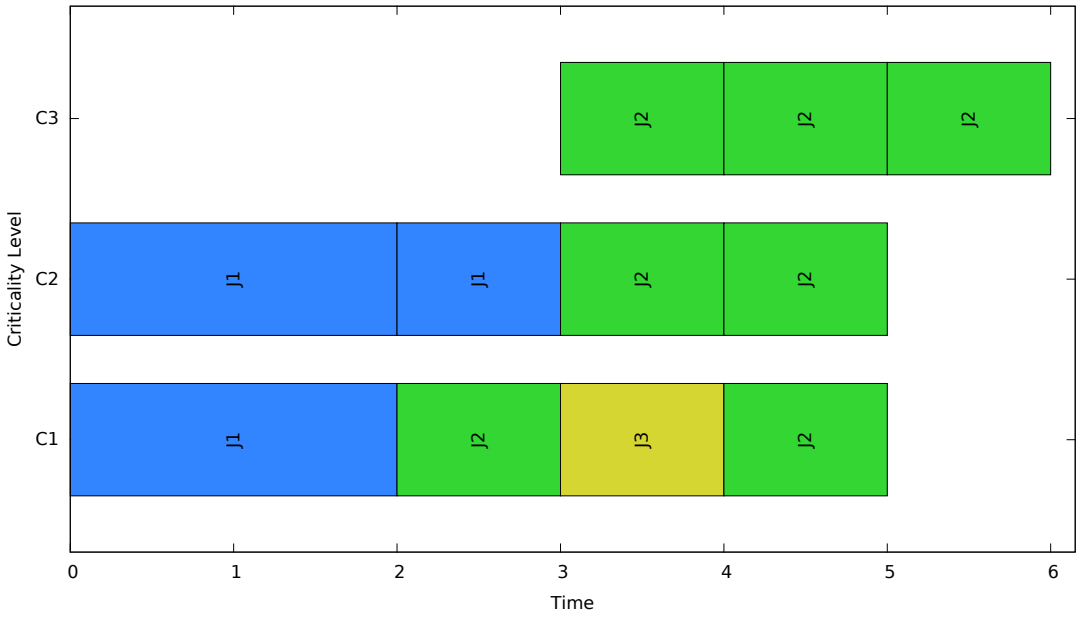


FIGURE 7.6: Gantt chart of the schedule

The scheduler automaton generated from the set of time-triggered tables is presented in Figure 7.7. For convenience, let us label T_1, T_2 and T_3, the time-tables for criticality levels 1, 2 and 3 from Figure 7.6. The first step to generate the scheduler automaton is to defined an initial location, labeled l_0 in Figure 7.7. Since the execution starts in the nominal case which is scheduled by T_1, from the initial location, the only transition

possible is $start_1$ from l_0 to l_1 representing the execution of J_1 in T_1 at $t = 0$. As a general rule, after every $start_i$ action leading to a location l_i, a $terminate_i$ action is created from l. This is the case because we want to allow an executing job to be able to signal its termination. Thus from state l_1, a $terminate_1$ action is defined that will move to location l_2. In addition a timing constraint of $t \leq 2$ is defined on the terminate action since the job is given only two time units to execute in the time-table.

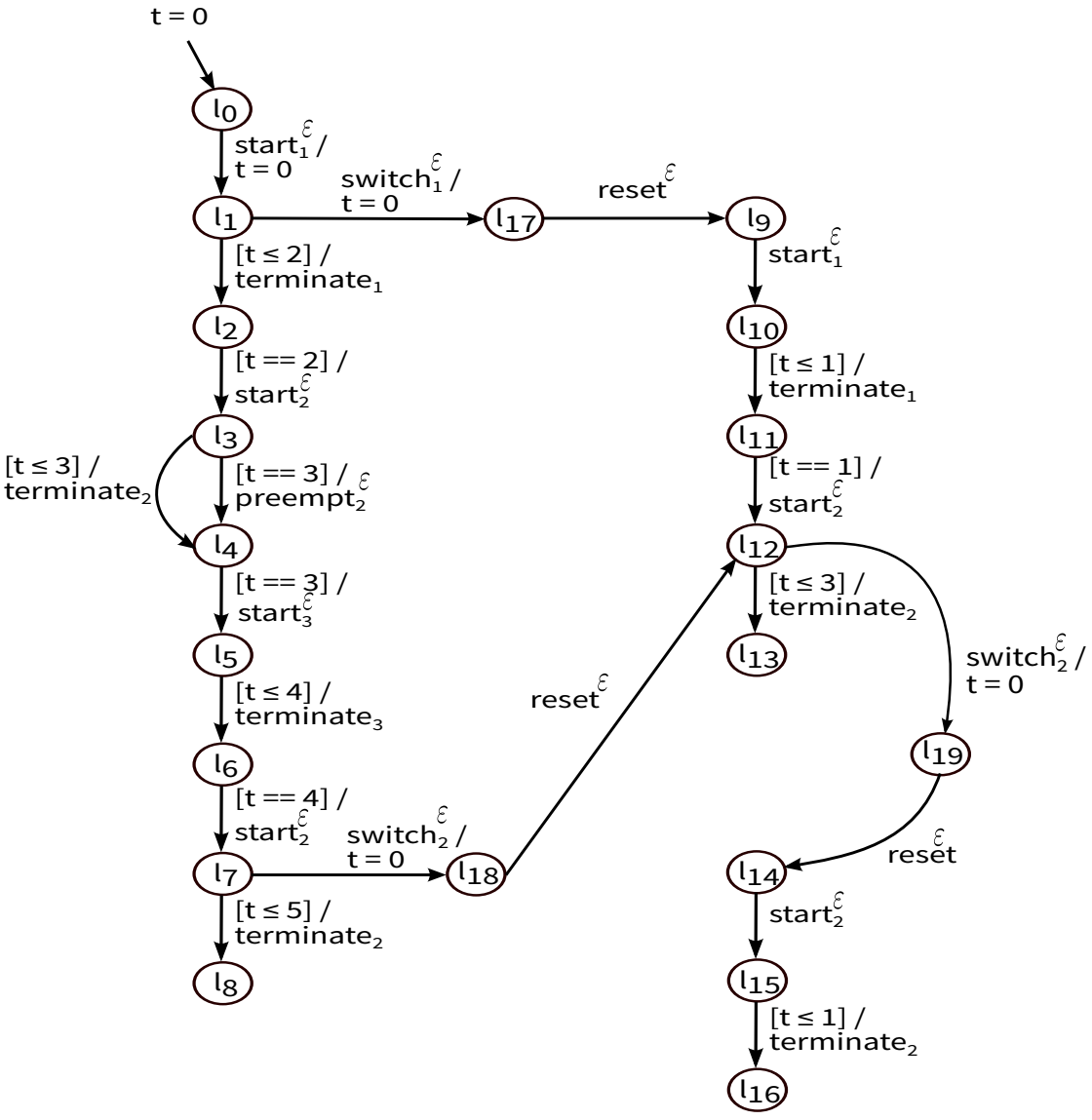


FIGURE 7.7: Automata for the scheduler

Next we need to represent the execution interval [2, 3) for job J_2. Unlike the case for J_1, J_2 is preempted at time $t = 3$ if it does not complete before. The preempt action is added with a time constraint of $t = 3$ as shown in the figure. In general, if we are representing an execution interval of a job, which is not the last one for that job, in the time table, then a preemption action is needed.

The previous steps show how to transform start, termination and preemption actions of a job from a given time-triggered table. The last thing needed is to map the mode switch. This is done by identifying the time instants where a mode switch can occur. This is simple in the case of an STTM policy, it can be identified by the end time of the last execution interval allocated for any job, whose criticality level is greater than the criticality of the time-table. For our example, in table T_1, the execution interval $[0, 2)$ for job J_1 is the last one for that job. Since the criticality of J_1 is higher than that of T_1, a mode switch can occur at $t = 2$. In the automaton, a mode switch is represented by two action, a switch action that is synchronized with the switch signal from the controller, and a reset action to change the execution mode of jobs.

7.5 Chapter Summary

In this chapter, we introduced a component-based design that models a system characterized by a set of MC-jobs and an STTM scheduling policy. We proposed a process to generate timed-automata to represent a time-triggered schedule and jobs of different criticalities. We used the automatic diagnoser synthesis from [6] to detect errors, and the scheduling algorithm from Chapter 6 to handle them.

In the next chapter, we will consider again how an STTM policy can be integrated in the design flow of a system by making use of synchronized timed automata components similar to the work in this chapter. Yet, instead of assuming a preemptive single processor platform, we consider a non-preemptive multi-core MC-system, where we focus on managing shared resources and interference.

Chapter 8

MC-system Design with Coarse-grained Multi-core Interference

To manage the complexity of concurrent-system design, the applications running in the nodes of distributed systems have to be designed in an appropriate high-level model of computation (MoC). In addition, for systems that are timing-critical and compute-intensive, it may be required to introduce the so-called mixed-criticality resource managers (dynamic scheduling policies) that adapt system resource usage to critical run-time situations (*e.g.,* overheating, overload, hardware errors) by giving the highly critical subset of system functions priority over low criticality ones in emergency situations.

However, especially for modern platforms – multi- and many- cores – it is highly non-trivial to manage resources not only because of their inherent parallelism but also because of "parasitic" interference between the cores due to shared hardware resources (buses, FPU's, DMA's, *etc.*). To close the semantical gap between MoCs on one side and resource managers on the other, we compile the MoCs into an expressive automata-based language, used to validate and implement a given MoC/resource manager combination.

8.1 Introduction

In this chapter, we present our design flow for scheduling and deployment of software designs for embedded systems. Modern embedded applications constitute so-called *nodes* of *distributed systems*, *i.e.,* they communicate via buses and networks with other applications (nodes). We consider systems that are not only *timing-critical*, *i.e.,* subject to hard

real-time constraints, but also *mixed-critical*, *i.e.,* able to sustain highly-critical functions even under harsh compute-resource shortage situations. The latter is desirable if the system has to be *autonomic* [60], *i.e.,* able to operate in open and non-deterministic environments. An example of an autonomic mixed timing-critical system is a "fleet of UAV's (unmanned air vehicles) [61]" that coordinate with the leader UAV within strict time bounds to avoid mutual collision. Such systems should not only be correctly specified but also *schedulable in real-time*. The point is that control tasks in many applications are augmented by complex computations that can load the processor significantly (*e.g.,* computer vision, trajectory/route calculation, image/video coding, graphics rendering). In such cases, to meet the high computational demands inside the nodes while keeping their energy consumption, cost and weight manageable it is important to consider multi- (2-10) or even many-core (x100's cores/'accelerators') platforms.

A major obstacle for schedulability analysis of multi-core applications is 'bandwidth interference' [62], *i.e.,* blocking due to conflicts in simultaneous accesses to shared hardware resources, such as buses, FPU's, DMA channels, IO peripherals. Next to interference, the other dimensions in the scheduling problem are (i) possible lack of preemption support in many-core systems, (ii) inter-task precedences (dependencies), commonly implied from the application's model of computation (MoC) and (iii) switching between normal and emergency mode in mixed-critical scheduling. To be able to address all these dimensions at the same time we propose simplifications which make the scheduling problem amenable for known heuristic methods with some adaptations.

We also put the proposed scheduling approach into the context of our design flow, which offers not only scheduling but also deployment on the platform. The deployment is ensured by a compilation tool-chain that is by construction customizable to various MoCs and online scheduling policies by mapping them to an expressive intermediate 'concurrency' language.

In Section 8.2, we introduce one-by-one the main pillars of our design flow, such as MoCs and mixed-criticality. Section 8.3 introduces the structure and assumptions of the proposed flow and illustrates it via a small synthetic application example. Section 8.4 gives a basic explanation of the scheduling algorithm and discusses the results.

8.2 Background

8.2.1 Models of Computation

To manage concurrency and coordination between tasks in parallel and distributed environments Models of Computations (MoCs) have been proposed in the literature. They permit the application designer to define the structure and organize the tasks and their

communication channels in a way that resembles high-level specifications (functional diagrams). MoCs intend to abstract the application's behavior from any implementation detail. Figure 8.1 shows an example: a part of an industrial avionics application modeled in a MoC called Fixed Priority Process Network (FPPN) [63].

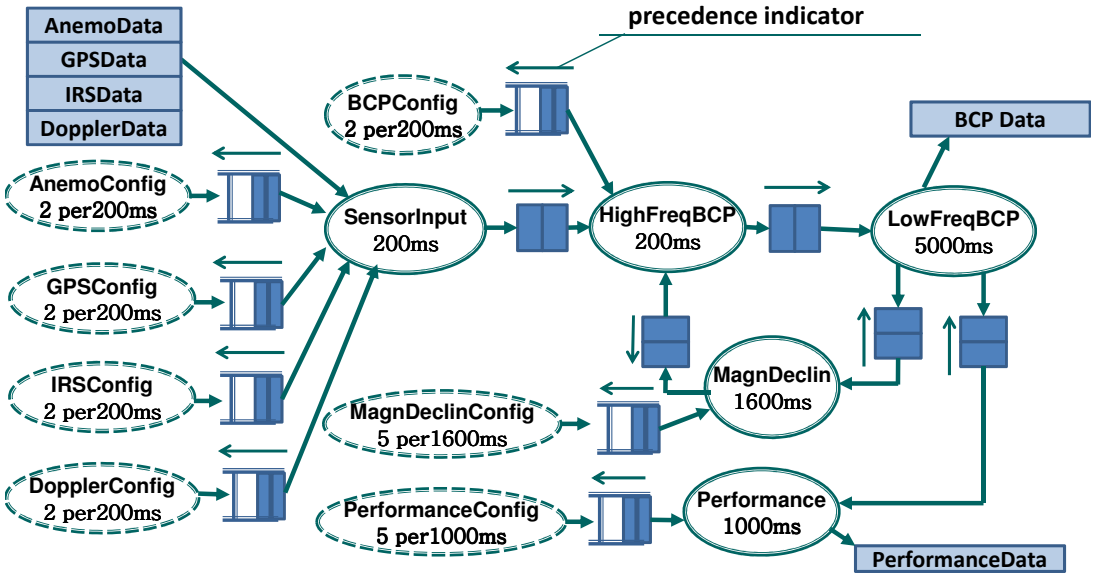


FIGURE 8.1: Application modeled in a MoC: flight management system in FPPN

In the figure we see (1) tasks, *e.g.,* 'HighFreqBCP', *etc.*, annotated by periods, (2) inter-task channels, *e.g.,* between 'DopplerConfig' and 'SensorInput', and (3) precedence relation between tasks, *e.g.,* 'HighFreqBCP' has higher precedence than 'BCPConfig'. The application consumes data from *input buffers*, *e.g.,* 'AnemoData', and produces the results to *output buffers*, *e.g.,* 'BCP Data'. The buffers are supposed to keep the slots for input and output data available during the whole interval between the task arrival and the deadline. As a MoC, FPPN should define the partial ordering of execution and interaction of concurrent activities (tasks), and this is done via the precedence relation, which ensures predictable inter-task communication.

Next to FPPN, many MoCs have been proposed in the literature for embedded multi-core systems, to name just a few: MRDF (multi-rate data-flow, often named SDF – Synchronous Dataflow) [64], Prelude [65], SADF (scenario-aware data-flow) [66] and DOL-Critical [67].

8.2.2 Resource Managers and Concurrency Language

An important property of autonomic embedded systems is their ability to adapt themselves to unexpected phenomena [60]. When a system is compute-intensive (which should be the case when a multi-core implementation is necessary) and time-critical, it has to be able to adapt itself to exceptional shortage in compute resources. In real-time systems, '*resource managers*' are software functions that monitor utilization of compute resources and ensure such adaptation. For this, they apply different mechanisms, such as mixed-criticality, QoS management, DVFS (Dynamic Voltage and Frequency Scaling), *etc.*. Especially the mixed-criticality approaches are gaining more an more interest and have a high relevance for collective adaptive systems [61]. A resource manager is an integral part of an *online scheduler i.e.,* a middleware that implements a customized online scheduling policy.

Unfortunately, there is a considerable semantical gap between the online schedulers and the middlewares that implement MoCs, even though both define software concurrency behavior. We aim at a common approach that can ensure consolidation, by representing both types of middleware in a language that is expressive enough such that it can encompass all possible concurrency behaviors for real-time systems, including their timing constraints. We refer to that common language as *concurrency* language (or backbone language) [68].

We believe that for autonomic timing-critical systems a proper choice of concurrency language is a combination of procedural languages and *task automata*. The latter are timed automata extended with tasks [69, 70]. Timed-automata languages in general are known to be convenient means to specify resource managers, such as QoS [71] and mixed criticality [72].

In our design flow, the concurrency language is BIP. Under 'BIP' we mean in fact its 'real-time dialect' [71], designed to express networks of connected timed automata components. In [73], BIP was demonstrated relevant for distributed autonomic systems. In [67], it was extended from timed to task automata, by introducing the concept of *self-timed* (or 'continuous') automata transitions, *i.e.,* transitions that have non-zero execution time, to model task execution.

In our approach, the applications are still programmed in their appropriate high-level MoC because in many cases an automata language, though being appropriate for resource managers, may still be too low-level for direct use in application programming. Instead, we assume automatic *compilation* of higher-level MoCs into the concurrency language. Due to well-known high expressive power of automata to model concurrent systems this must be possible for most MoCs. In an ideal case, the compilation would

be configured by a user-defined set of grammar rules for automatic translation of the user's preferred MoC into automata.

8.2.3 Concurrency Language based Representation of System Nodes

Figure 8.2 gives a generic structure of a concurrency language model of a distributed-system node running an application expressed in a certain MoC. We also zoom into the BIP model of an important component.

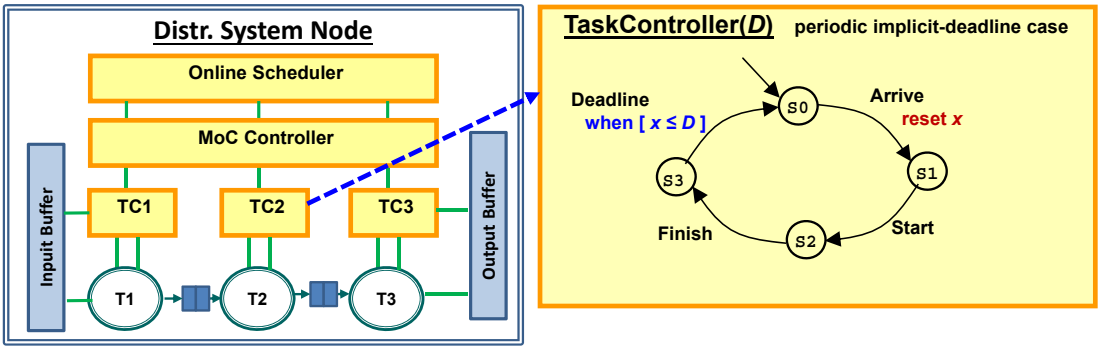


FIGURE 8.2: Concurrency language representation of a timing-critical application

The basic components of the model are automata, *i.e.,* finite-state machines that can interact with other components by participating in a set of interactions with other automata as they make discrete *transitions* (basic steps of execution). In our model, we have one automaton per application task and one per inter-task channel, and also an automaton to control each task – the so-called *task controller*. There is also an automaton that ensures proper task execution order according to MoC semantics, we refer to that component as MoC controller. One can also introduce an automaton that would further restrict the ordering and the timing of task executions – the online scheduler. This component would impose user-programmed scheduling policy. Note that automata can be hierarchical, *i.e.,* they can represent a composition of more primitive automata.

In Figure 8.2, we zoom into a task controller for periodic tasks whose deadline is equal to the period. It consists of a cyclic sequence of states, with initial state 'S0' and first transition 'Arrive', which models task arrival and is followed by transition 'Start', which corresponds to starting a new iteration of task execution, called a *job*. The 'Start' transition is followed by 'Finish' transition when the job finishes. After the finish, the deadline-check transition 'Deadline' is executed. The deadline is checked as follows: upon task arrival a so-called clock variable x is reset to zero. This variable acts as a timer indicating the time elapsed since the last clock reset. After the job has finished we check whether the deadline D was respected, *i.e.,* whether $x \leq D$.

Note that in our design flow, the given task controller is both time- and event-driven, as the tasks arrive periodically (in a time-driven way) but start when the MoC controller

would enable the 'Start' interaction, thus indicating that the task predecessors have finished (in an event-driven way).

The respective *schedule optimization problem* is to find a multi-task schedule where all tasks fit into their respective scheduling windows while respecting precedence constraints implied by the MoC and finite-resource constraints (*e.g.,* non-zero time for computation) with mutually exclusive access to resources by different tasks. The latter requirement introduces certain peculiarities when the platform is a multi-core. The problem is solved by an *offline scheduling algorithm*, which gives the solution in terms of parameters to be given for *online scheduling policy*.

8.2.4 System Scheduling Aspects

Figure 8.3 illustrates the schedulability conditions of a timing-critical distributed system.

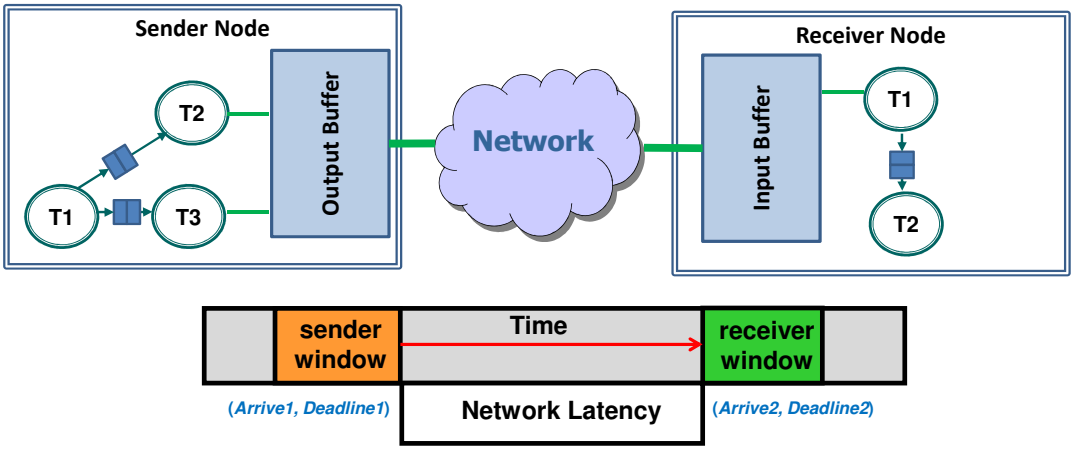


FIGURE 8.3: A simple distributed system and its iteration window

The figure illustrates a simple single-rate two-node system (sender and receiver) and a timing window of a single system iteration. The iteration window consists of three different sub-windows. The first one is between the arrival (*i.e.,* the release) time and the deadline of the sender tasks. In this window the sender should prepare the output to be sent to the receiver in its buffer. The next sub-window corresponds to the network delay, and the third one is the window for holding the data at the destination node, this window represents the arrival time and the deadline of the tasks at the receiver. Note that subsequent system iteration windows may overlap in time (*i.e.,* pipelined executions, when iteration period is smaller than the iteration window size), and that this model can be generalized to multi-rate system (in which case one iteration would correspond to a hyperperiod) with multiple buffers. Note that in a distributed system

different nodes may need to maintain sufficient alignment of their local time models by running a clock synchronization protocol.

In our current work, we still mostly focus on the design of a single system node, with common or distinct scheduling windows (non pipelined – for simplicity) for different tasks of the given application running in the node.

8.2.5 Multi-core Interference Aspects

When dealing with multi-core platform architectures as targets for timing critical applications a particular serious problem arises. Spontaneous unpredictable or hardly predictable 'parasitic' timing delays – '*interference*' – manifest themselves when multiple cores run in parallel. Interference appears when cores await response from resources that are in use by other cores. This is illustrated in Figure 8.4.

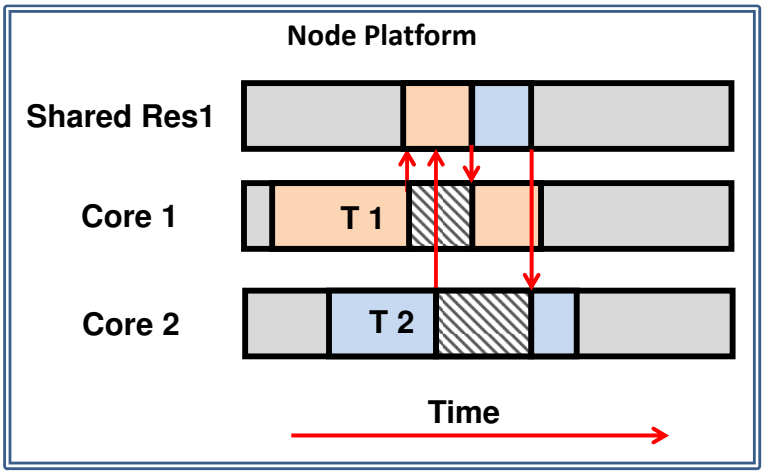


FIGURE 8.4: Multi-core interference

The concerned resources can be either hardware or protected logical (software) resources. Shared hardware resources that can cause interference are global buses, bus bridges and switches, coprocessors, peripherals, and even FPU's (if they are shared between cores to save on-chip area). Software shared resources are, for example, mutex-lock segments in the source code and calls for mutually exclusive services in the system runtime environments.

Interference can be *coarse-grain* or *fine-grain*. In the former case the accesses to the shared resource occurs in 'coarse' blocks, called superblocks [74], which occur just once or a few times per task execution. Often a task has one superblock to read all the input

data from global to local memory at the start and to write the data at the end. Fine-grain interference is sporadic and can occur a large number of times per task execution, *e.g.,* bus accesses due to loads/stores in the memory.

In a design flow for mono-core systems the 'worst-case execution time analysis' conveniently precedes 'schedulability analysis', as the task's WCETs do not depend on the schedule. On the contrary, in a multi-core system, because of interference task execution delay may significantly change depending on which tasks are scheduled on the other cores. Therefore part of task WCET analysis may have to be re-done when schedules are analysed, which is a major obstacle in the design of timing-critical systems based on multi-cores [62].

Luckily, coarse-grain interference can be *controlled* by scheduling the superblocks in a way that the resource conflicts are eliminated. To achieve this, in a 'controlled' schedule, potentially conflicting superblocks are executed sequentially. At the same time, uncontrollable fine-grained interference can be for as much as possible transformed into coarse-grained one by 'concentrating' the resource-access intensive parts of source code together into coarse-grained superblocks, which can be controlled. The controlled interference approach is well-known in the literature. For example, in [75], coarse-grained blocks of accesses to global bus are considered as special sub-tasks which are scheduled in an optimal static order.

In our scheduling algorithm, we assume controlled coarse-grained interference, whereas the remaining fine-grained interference that could not be transformed into coarse-grained one is assumed to be taken into account either via extra WCET margins or, more conservatively, by modeling complete tasks as superblocks. In addition, though different resources (*e.g.,* different FPU's and different memory banks) can be accessed independently and though different superblocks can have different timing costs, we make a simplifying assumption that there is only one shared resource and the duration of all superblocks is the same, we denote it δ. In a way, we consider superblocks as instances of a special task whose WCET is δ.

A particular form of such interference that manifests itself in our design approach is called *engine interference* [67]. In our concurrency model, governed by automata, one can distinguish task-concurrency control operations which correspond to discrete transitions of the automata components that constitute the system. All discrete transitions are coordinated via a single control thread called the *engine*. Suppose that δ is the worst-case time to handle one discrete transition. Then the runtime overhead of task concurrency control operations can be conveniently modeled as interference between superblocks of size δ. In addition to the necessary accesses to the engine needed to coordinate task concurrency, each coarse-grained block of accesses to any resource can

be, in principle, delegated to the engine as well. For this, the compiler would have to represent each superblock as a discrete transition or, if it is large, as a sequence of transitions. Therefore, the engine interference can be generalized to subsume other forms of coarse-grained interference.

In the present work, engine interference is the only form of interference that is automatically modelled by our tools. Compared to [67], the novelty is that in the present work we control this form of interference in the scheduling. Our scheduling algorithm assumes that there is one shared resource, and we model the engine as such. Further, it assumes that all superblocks are explicitly represented by special tasks with equal WCET δ, and we model the task-controller transitions as such.

To manage the remaining fine-grained interference we advocate the *time-triggered scheduling* approach, *i.e.,* letting the tasks start at fixed time instances even if previous tasks finish earlier. This approach does not make worst-case response-times of tasks worse, while it significantly reduces the complexity of a fine-grain interference analysis (which would compute the WCET margins) and improves its accuracy. The point is that when tasks do not shift their execution earlier upon earlier completion of previous tasks the number of task pairs that can potentially run in parallel (and hence interfere) is significantly reduced, which effectively cuts the number of analysis cases to be covered.

8.2.6 Mixed-Criticality Aspects

In adaptive autonomous systems one has to provide for unexpected situations. In terms of scheduling this means allocating worst-case amount of resources with a significant extra margin. To damp the high costs that such margins incur, the allocated extra resources are given, 'on an interim basis', to less-critical and less important functions in the system which can be stopped at any time to free up the resources in the case when highly-critical and highly-important functions need them. This reasoning leads to a generic mixed-criticality resource management approach, see Figure 8.5.

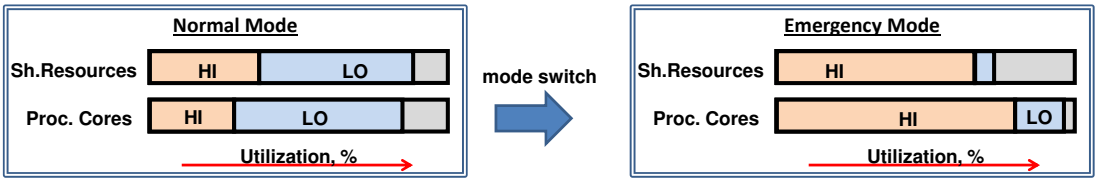


FIGURE 8.5: Mixed-criticality resource management

We currently consider a common case of having just two levels of criticality. Less-critical functions are given low criticality level, commonly denoted 'LO'. Highly-critical functions

are given high criticality level, commonly denoted 'HI'. For example, in a UAV system LO can correspond to mission critical and HI to flight-critical functions.

As shown in Figure 8.5, in case of emergency the HI tasks get high resource utilization margins. However in normal mode of operation these margins are never used and are given to LO tasks. Only when emergency situation occurs where HI tasks need more resources a 'mode switch' from normal to emergency mode is performed by the resource manager whereby the extra margins are 'claimed' by HI tasks. In our approach, the respective resource management policy is implemented in concurrency language as part of the 'online scheduler' automaton component [72].

There are two distinct approaches to free up the resources from LO tasks in the case of mode switch. The first approach is *dropping* the LO tasks (*i.e.,* instantaneous aborting them with possibility to resume their execution later on). The second approach is putting the LO tasks in *degraded mode*, *i.e.,* signalling them to do less computations and accesses to shared resources at the cost of the lower output quality or missed deadlines. A major challenge in mixed criticality scheduling is that the mode switch may occur at any time not known in advance and that it is required to guarantee schedulability no matter whether and when the switch occurs [76].

From schedule optimization point of view the task dropping makes it easier to find optimal solutions as decision to start execution a LO task at a certain time has less consequences for the available possibilities to schedule a HI tasks. On the contrary, for online scheduler it is much more difficult to implement dropping than degraded mode.

As explained in the previous section, to better handle interference we use the time-triggered scheduling, to be more specific, we use STTM online policy [76, 77], which is a generalization to mixed-criticality scheduling. Recall that, in this policy, the normal and the emergency modes each have a time-triggered table. A switch from normal to emergency table can occur at any time instant, while it should be guaranteed that if HI critical tasks need to claim their extended resource budgets reserved for unpredictable situations then they will always get them in full amount. Though this appeared to be by far not trivial, in [77] we have proved theoretically and experimentally that this approach is as optimal in the worst case as the event-triggered approach.

8.2.7 Related Work

Different previous works address related problems, some of them are discussed in this subsection. Reference [78] is an extension of [76] which calculates STTM tables for multi-rate synchronous mixed-critical systems. This work is restricted to uniprocessor platforms. Task automata verification [70] has unprecedented expressive power, but may

be subject to scalability issues for industrial-scale systems unless it is applied with some approximations and abstractions. The superblock approach [74] is a well-recognized technique to address even fine-grain interference and it has been further developed in related work. However a problem remains still open, see [62], concerning calibration of fine-grain analysis techniques to typical processor and bus architectures that are deeply pipelined and have other performance optimization features.

The semantics of synchronous systems is relaxed even further compared to [78] in the direction of functionally equivalent asynchronous pipelined execution with self-timed synchronization, following the philosophy of Kahn Process Networks (KPN). The goal was to support embedded signal processing and multimedia stream-processing in general. Rich liveness, memory boundedness, code generation and real-time throughput/latency analysis theories have been developed for these models, termed *data-flow MoCs*. An interesting survey for expressive real-time analyses is given in [66]. Paper [75] studies optimal handling of coarse-grained interference in simple variants of such MoCs.

In data-flow MoCs, *classical* concepts such as release times, periods and deadlines are replaced by self-timed iterations, long-run throughput and multi-iteration latency bounds. The 'FPPN' MoC, adopted in our flow, can be seen a data-flow MoC which, in a way, 'attempts' to reconcile itself to classical concepts. Also our work extends the data-flow-related MoCs to mixed criticality.

Only a few scheduling techniques mentioned above are integrated in software engineering toolchains that have both real-time scheduling and code generation. First of all, ADA programming language and its Ravenscar profile are de facto standards for mono-core hard real-time systems. They integrate the most trusted and safe multi-task programming and scheduling techniques for tasks that have no explicit precedence constraints. For the case of distributed systems that work was extended to multiple mono-core platforms or partitions communicating via bus and network protocols in the context of TASTE design flow [79]. This work requires extension in order to treat multi-core system nodes and precedence-constrained task models such as FPPN.

Prelude design flow [65] represents an ongoing work on scheduling and deployment of multi-rate synchronous systems defined with more expressive ('synchronous-language') semantics than the ones assumed in our flow and in [78]. It should be noted that the price to be paid for higher expressive power is that it becomes much less obvious how to generate a semantics-preserving task-graph or data-flow MoC model in this case that could provide an input to a precedence-constrained scheduling tool.

A variant of superblock approach was implemented in DOL-BIP-Critical design flow [67].

Task-automata verification is integrated into Times and UPPAAL tools [69]. CompSoC design flow [80] deploys applications based on scheduling algorithms for data-flow MoCs.

8.3 Design Flow

8.3.1 Underlying Paradigm

There is neither a single MoC nor a single online scheduling policy that would be recognized universal for all timing-critical systems. This is especially the case for multiprocessor and distributed systems and when interference, task-dependency and mixed-criticality challenges are to be considered. The policies and MoCs will continue intensive evolution whereas industrial systems need rapidly adjustable implementations, while the corresponding analysis techniques need a basis to establish formal proofs for them. Therefore our target design flow is customizable, at least conceptually, to different MoCs and policies by compiling the MoC and representing the scheduling policy in a common task-automata based concurrency language, for which, in our design flow, we use BIP. Therefore, we do not create a custom middleware specialized for FPPN MoC and for STTM scheduling policy, but instead we express them in BIP [67, 68]. The BIP implementation of the system on top of BIP runtime environment (RTE) should not leave the underlying platform any significant real-time scheduling decision freedom but should map the user-programmed scheduling policies to basic operating system mechanisms, like threads and dynamic priorities [67, 81].

The main contribution of our work is handling coarse-grained interference in the context of mixed-critical systems with precedence constraints between multi-rate tasks. We address the complex problem by practically meaningful simplifications. We assume that the task system is synchronous-periodic or can be over-approximated as such by periodic servers. A synchronous system can be represented by a semantically-equivalent static task graph, [63, 78], conveniently presentable to a list-scheduling heuristic, which, in turn, has reputation of reasonable performance for comparable instruction-level scheduling problems [82]. Moreover, we present a design flow where applications can be both programmed and scheduled. Other design flows that have this property, *e.g.,* [61, 65, 67, 69, 79, 80], do not take into consideration all the aspects we do but in return offer other features, *e.g.,* distributed-system/network support or expressive power. We compare to [67] in the next section. Related scheduling techniques [66, 70, 74–78, 83] also have some restrictions, while in return offering important theoretical properties and features.

8.3.2 Flow Structure and Assumptions

Our design flow is shown in Figure 8.6. At the input we take the application specified as a MoC instance (*i.e.,* a network of task elements connected to channel elements and annotated by parameters) and functional code for the tasks. From the MoC instance the tools derive a task-graph for offline scheduling. The task graph describes the application hyperperiod in terms of job nodes and precedence edges. The 'jobs' are task executions and the precedences are derived from the semantics of the given MoC. The application is translated into concurrency language – BIP. The schedule obtained from the offline scheduler is translated into parameters of the online-scheduler model specified in BIP.

The joint application-scheduler model (with a basic structure as previously outlined in Figure 8.2) is translated by the BIP compiler into a C++ executable. The executable is linked with BIP RTE (the 'engine') and executes on a platform on top of the real-time operating system.

When running on the platform, the binary executable encounters interferences, as discussed in Section 8.2.5. Handling interference requires a feedback loop from the binary executable to the offline scheduler tool. Next to the worst-case execution times (WCET's) of tasks, the worst-case execution time δ of coarse-grained superblocks should be obtained and back-annotated at the input of the scheduler tool, and then the flow should be re-iterated (at most once, as the 'pure' WCET should not depend on the schedule).

We put the following requirements on our target design flow. We assume FPPN as application MoC. The offline scheduler should support non-preemption, precedence constraints implied from the FPPN and take into consideration coarse-grained interference. The online scheduler should support task migration and task dropping. The online scheduling should be based on STTM scheduling policy for mixed criticality.

The main reason of assuming non-preemption is lack of support of preemption in the current version of BIP language and RTE engine. Though preemption can be modeled and simulated [72], it cannot yet be executed in real-time mode. This is subject of future work. A justification for considering non-preemption is frequent lack of support of preemption in multi-core platforms that have a large number (> 8) cores (so-called many-core platforms and graphical accelerators).

In our design flow, we reuse certain elements from the previous'DOL-BIP-Critical' flow [67] The name of the MoC involved in that flow was DOL-Critical. It is closely related to FPPN, and the same specification language, named DOL-C, is currently used to specify instances of both FPPN and DOL-Critical models. FPPN has more general notion of

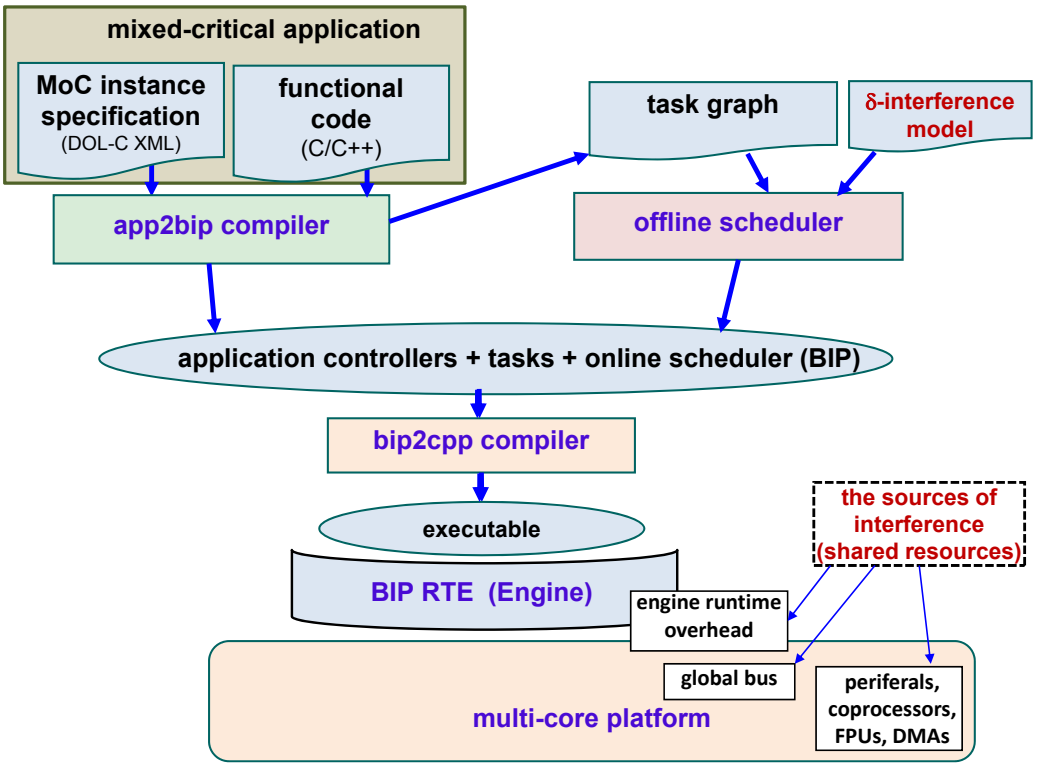


FIGURE 8.6: Design flow

task precedence than DOL-Critical, as it supports precedences between any pair of tasks, and not only between equal-rate periodic tasks.

There were essential differences in the scheduling assumptions taken in the previous flow, where the tasks were executed essentially in as-soon-as-possible (ASAP) fashion *i.e.,* immediately after the previous task mapped to the same partition. Instead we impose time-triggered start of each task, which should significantly simplify the analysis of bandwidth interference. The offline scheduler of previous flow had the advantage of supporting time partitioning, degraded mode and excluding the interference between HI and LO criticality levels.

Currently in our work, we have a version of the offline scheduler that satisfies the desired criteria, except that the interference models presented at the input of this tool are currently restricted to those for BIP engine interference of implicit-deadline periodic task controllers. Though advanced interference detection methods are known in related work [84], we still miss them in our flow. If such tools were available we could adapt or extend the δ-interference model assumed in the offline scheduler. Next to this, the online scheduler is not yet properly integrated, as it still does not support dropping and task migration, though such features are within reach, *e.g.,* a restrictive form of

BIP-component migration is demonstrated in [67] and thread API's offer means for dropping.

In the remainder of this section we discuss the currently available tools and illustrate their use by concrete examples. For multi-core experiments presented here, we use a LEON4 platform with four cores implemented on FPGA, using RTEMS OS with symmetric multiprocessing. For this platform, as measurements show, the worst-case execution time of one BIP interaction step takes: $\delta = 1\ ms$.

8.3.3 An Example Illustrating the Flow

Figure 8.7 gives a synthetic application example with three tasks. The 'split' task puts two small (a few bytes) data items to the two output channels and sleeps for around 1 ms to imitate some task execution time. Tasks 'A' and 'B' read the data. Task 'A' sleeps alternately for 6 ms and 12 ms, to model 'normal' and 'emergency' workload levels. This task models a high-criticality task. Task 'B' supports two modes of execution: normal and degraded. In normal mode it sleeps for 6 ms, in degraded mode it skips all execution, even reading the input data. This task models a low-criticality task.

All tasks have the same periodic scheduling window, with period and deadline being 25 ms. In a real application, this would correspond to the time during which the two imaginary input data buffers should be read, computations should be done and the output buffers should be written.

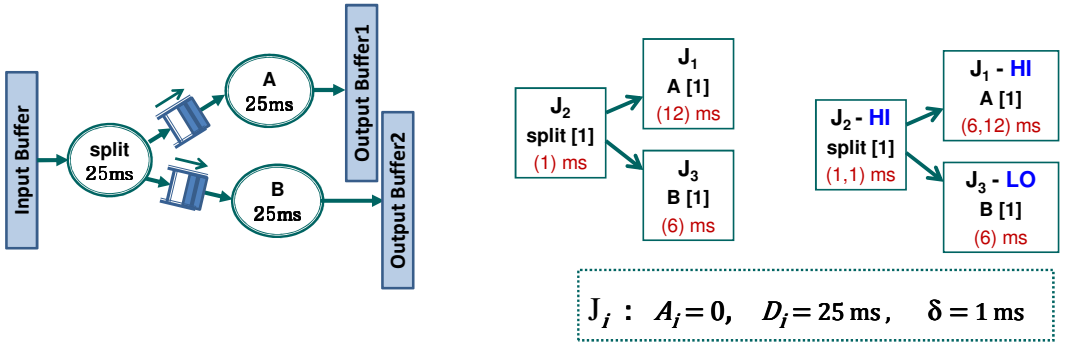


FIGURE 8.7: Three-task example: MoC (left), ordinary task graph (middle) and mixed-criticality task graph (right)

The middle part of the figure gives the 'ordinary' (*i.e.,* non mixed-critical) variant of the task graph. Every task is represented by a job. The jobs are numbered: $J_i = J_1, J_2, J_3$ and annotated by their worst-case execution times. Their individual arrival times A_i and deadlines D_i are the same in this example. The right part of the figure corresponds to the 'mixed-critical' variant of the same graph. The execution times of highly-critical

tasks are represented by a two-valued vector: normal-mode time and emergency-mode time.

The engine runtime overhead (as it will become clear later) constitutes $4\delta = 4$ ms per task (in total 12 ms). Therefore, when assuming ordinary execution times this example cannot run on a single core, as the total execution time amounts to $12 + 1 + 12 + 6$=31 ms, which is larger than the 25 ms deadline. The offline scheduler evaluates the load (*i.e.,* maximal demand-to-capacity ratio) of this example to 31/25=124 %. Therefore it predicts that at least two cores are necessary.

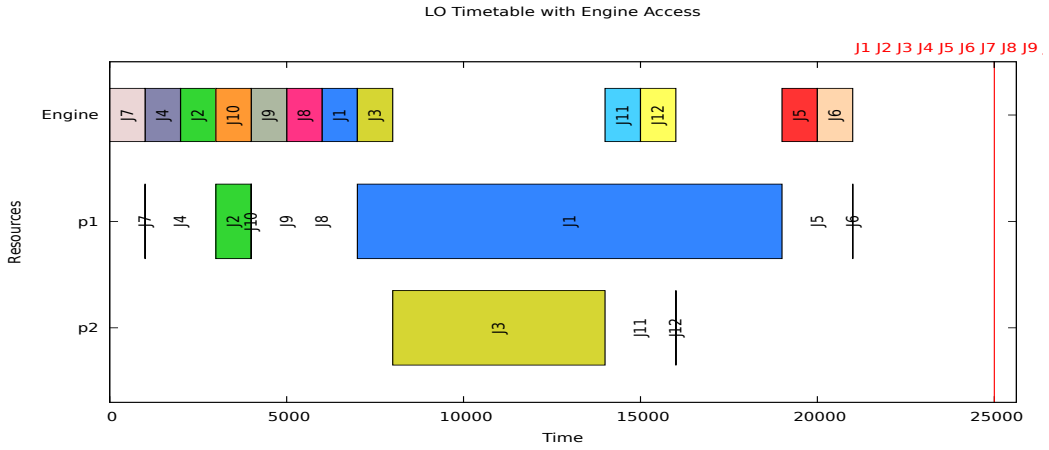
On the other hand, in the mixed-criticality case we consider the two execution modes – normal and emergency – separately. In the normal mode Task 'A' has execution time 6 ms, which is 6 ms less, and we have a load $25/25 = 100$ %, for which a single-core may be sufficient. In the emergency mode the execution time of Task 'A' is again 12 ms, but we drop Task 'B', which saves us $6 + 4$=10 ms and leads to the load of 21/25=84 %, which again may be doable on a single core. Thus, mixed criticality can help to use the cores more economically.

The tool generates the schedules for the ordinary graph and for the mixed-critical one, as shown in Figure 8.8. Figure 8.9 shows the Gantt charts of executing the two variants of the schedule on the LEON4 board.

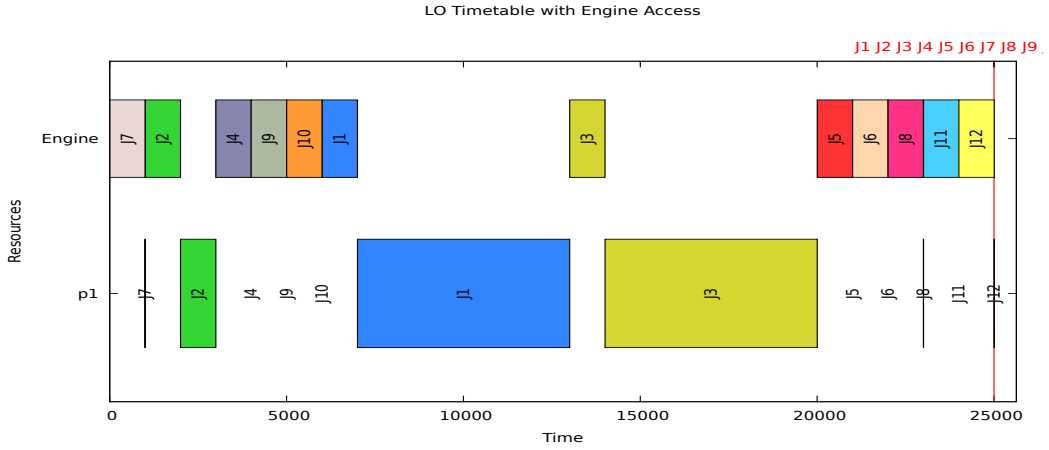
In every Gantt chart, the first line shows the execution of the BIP Engine on 'Core 0'. One may wonder why a whole core would have to be reserved to a runtime environment. This is due to lack of support of preemption in current BIP RTE. Moreover, it should be noted that in many-core systems (or graphical accelerators), this is justifiable, as in practice there are plenty of cores available – *e.g.,* 16 per shared-memory cluster in [85] – and no preemption is allowed. On the contrary, a platform such as LEON4 supports preemption and does not assume one thread per core. For such platforms in future work we intend to interleave high-priority engine control thread with a lower-priority task-execution thread on Core 0. Note that the engine thread executes also the BIP components responsible for control operations, such as the task controllers, the MoC controller and the online scheduler.

Recall that the shared resource on which interference-modeling is currently supported by the tools is the engine. As we see in Figure 8.8, every task execution is prefixed and suffixed by two δ-accesses to Core 0. In the ordinary schedule, Task 'split' and Task 'A' are mapped to Core 1 and Task 'B' to Core 2.

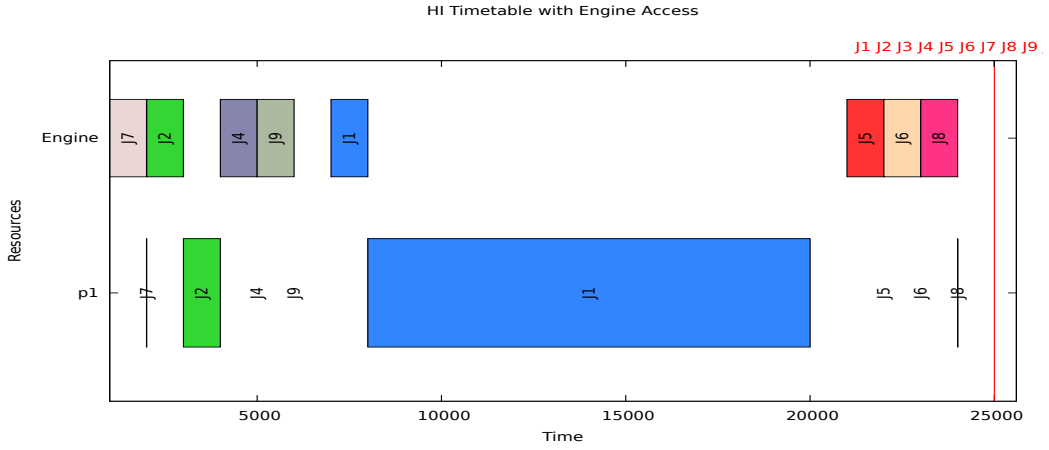
The platform-measurement charts in Figure 8.9 show two periods, one in normal and one in emergency mode. The offline scheduler 'ordinary' solution assumes the overall worst-case, whereas the mixed criticality solution distinguishes two modes. Comparing



(a) Ordinary Schedule

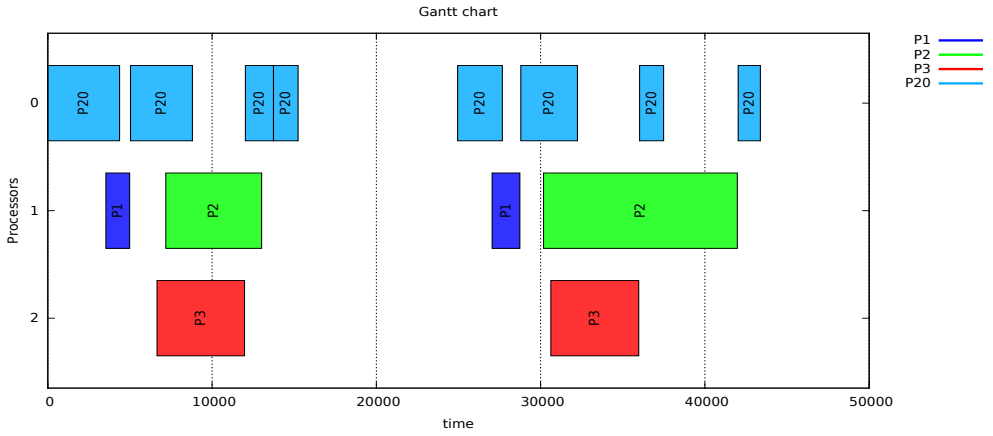


(b) MC Schedule: Normal Mode

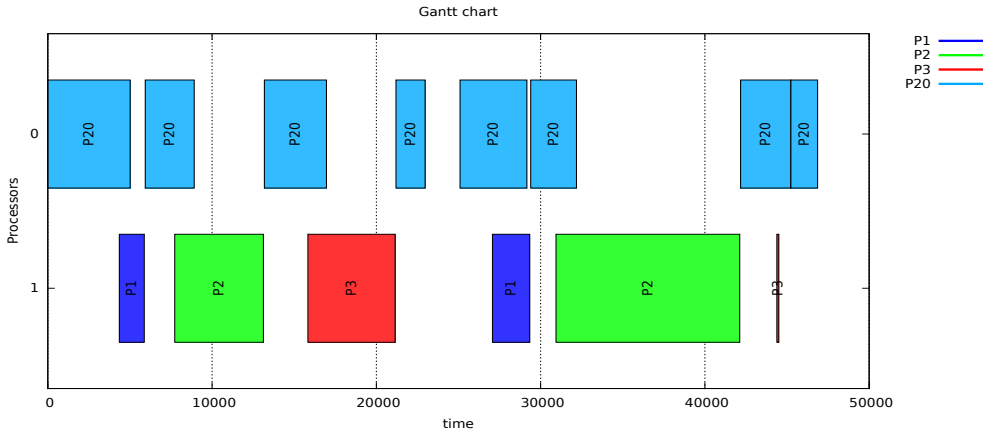


(c) MC Schedule: Emergency Mode

FIGURE 8.8: Three-task example: offline-scheduler solutions



(a) Ordinary Execution Traces (No mode switch in the second period)



(b) Mixed-critical Execution Traces (Dropping J_3 in the second period)

FIGURE 8.9: Three-task example: platform execution traces

the corresponding segments of Gantt charts of the solutions and measurements we see a match, though not a perfect one. This is because the offline scheduler output is not yet supported as input to the online scheduler. We see that in the emergency mode MC case the offline scheduler drops task 'B' altogether, whereas the online scheduler still makes a short execution of Task 'B' in degraded mode.

Because of current temporary lack of tool integration we had to do manual modifications in the concurrency model that was automatically generated from FPPN, in order to ensure that the online behavior matches the offline solution. Note that a possibility for the user to refine the behavioral model by such modifications is itself an attractive design-flow property. We made modifications in the mixed-criticality variant of the design, in order to introduce the switch from normal to emergency mode. We ensure that if Task 'A' executes beyond its normal-mode execution time then Task 'B' is executed in degraded mode. These modifications are shown in Figure 8.10.

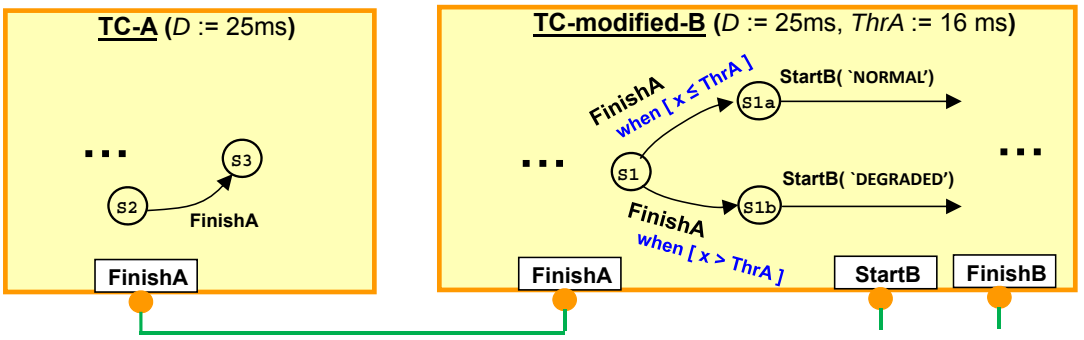


FIGURE 8.10: Three-task example: manual modification introducing a mode switch

We have modified the structure of the TC for Task 'B', which originally was as shown in Figure 8.2, by introducing a new transition between the 'Arrive' and 'Start' for Task 'B'. This transition is synchronized with 'FinishA' transition in the TC of Task 'A'. We check the value of clock 'x' which measures the time since the begin of the current period. If this value is larger than a certain threshold $ThrA$ then 'B' is executed in degraded mode.

8.4 Algorithm Description

In this section, we zoom into a particular tool in our design flow – the offline scheduler. We give some basic idea on the scheduling problem, the δ-interference model and the scheduling algorithm. Finally we show schedulability-evaluation experiments with random benchmarks.

A scheduling problem instance consists of a DAG task graph obtained automatically from a MoC; we have seen examples in Figure 8.7. The nodes, J_i are obtained from tasks and are annotated by parameters (A_i, D_i, χ_i, C_i), where $[A_i, D_i]$ give the job scheduling window (between arrival and deadline relative to the hyperperiod), χ_i gives the job criticality level ('LO' or 'HI') and C_i is a vector that gives the execution time in the normal and emergency modes. The problem instance also includes the selected number of cores (not counting the engine core) and some information on interference, currently we only take the value of δ, whose meaning is interference at the start of each job. The δ-interference model is shown at the left side of Figure 8.11.

This model can be described by a hypothesis that we have a global system controller *i.e.,* the automaton obtained from a combination of all concurrency-model automata present in the system. Lets call it by abuse of terminology the 'engine'. The engine controller makes discrete transitions (control steps), each step costing execution time δ at the control core. At certain steps the engine spawns a job on a compute core taken from a pool of cores. For this, an idle core is selected and reserved at the beginning

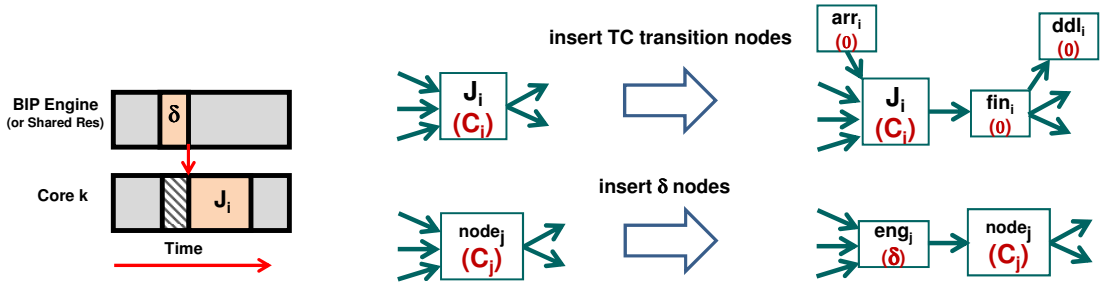


FIGURE 8.11: Engine ('Delta') interference and its modeling in the task graph

of the step. The steps that do not spawn any computations are modeled as steps that spawn a job with zero execution time. The engine does not make execution steps all the time, for some time intervals it may decide to do idle-waiting.

As we have seen in Figure 8.2, a periodic controller can be modeled as a system component that, for a given task, lets the engine make four subsequent steps corresponding to the following *transitions*: arrival, start, finish and deadline check. The real computation job is, in fact, triggered by the 'start' step, the other steps do not trigger any computations. Therefore, as shown in Figure 8.11, to model periodic jobs we insert three corresponding zero-execution time 'satellite' jobs. The arrival job becomes an extra predecessor of the original job, the finish job becomes the new successor after which all the original successors follow, where we also introduce a new successor – the deadline-check job. Now it should become clear why in our example in the previous section every job execution is prefixed and suffixed by two δ-steps. To model the part of the job that is executed on the engine Figure 8.11 shows the second graph transformation, which inserts a δ-predecessor at every job. Except for the execution time, the newly inserted 'satellites' get the same characteristics (*i.e.,* scheduling window and criticality) as the original job.

The scheduling algorithm is applied in our design flow to a graph obtained from the original MoC after it has been post-processed by the two graph transformations defined above. The algorithm generates the two schedules for the two execution modes. These schedules act online as tables for time-triggered execution, see *e.g.,* Figures 8.8(b) and 8.8(c).

First the normal-mode table is generated. This is done using global fixed-priority simulation that takes precedences into account. This algorithm is also known as *list-scheduling*. As mentioned before, we assume non-preemptive scheduling. The algorithm has been adapted to take into account two types of resources: a single control core and a pool of compute cores. In order to execute, every job first needs one instance of both resource

types for time duration δ to execute its δ-predecessor and then during its own time duration C_i continues running on the compute core only, whereas the control core is available to spawn another job. The algorithm maintains a *list of ready jobs* (and hence its name). As soon as the control core and a compute core become available to start another job the algorithm picks the *highest-priority* ready job and starts its simulated execution. A job is considered *ready* to execute if two conditions hold. Firstly, the job scheduling window $[A_i, D_i]$ must be already begun, *i.e.,* for the current simulated time t we have $t \geq A_i$. Secondly, all DAG predecessors of the job (if any) must be finished.

The *priorities* for selecting the next job to be scheduled are obtained from an earliest-ALAP-first (ALAP means 'as late as possible') fixed priority table. Job's ALAP time gives the latest time when it may complete its execution such that neither that job nor any of its transitive DAG successors will miss the deadlines. ALAP times are computed recursively, from the sinks to the sources, taking into account the execution times. Before ALAP times are calculated, the deadlines of the HI jobs are reduced by the value of their execution time uncertainty, *i.e.,* the difference between their execution times in the emergency and normal modes. Those are the effective deadlines that should be met to avoid missing the deadlines if a switch to the emergency mode occurs. These 'effective' deadlines give a HI job higher priority with respect to a LO job whose nominal deadline is the same. It is due to this reason that in our Three-Task example, in its mixed-criticality variant, (see Fig. 8.7, 8.8(a)), the HI Task 'A' is scheduled before the LO Task 'B'.

The emergency mode table is calculated such that at any moment of time a switch from normal to emergency mode may take place such that the HI jobs may continue without being preempted or migrated in the middle of execution to another core. To this end, the schedule start times in the normal mode are regarded as job arrival times in the emergency mode. Further, in this mode, we simulate only the HI jobs (while the LO jobs are dropped) taking into consideration only HI-to-HI job precedences while keeping the same job-to-core mapping and the same relative order of HI-job execution as in the normal mode. When a job deadline miss is detected in any of the two modes the algorithm fails.

Since our variant of list scheduling algorithm does not use dynamic priority tables and the static table can be obtained by simple topological sort algorithm, the complexity of our algorithm is the same as the one of list scheduling. Our implementation of this algorithm according to [77] has complexity:

$$O(V(\log V + M) + E)$$

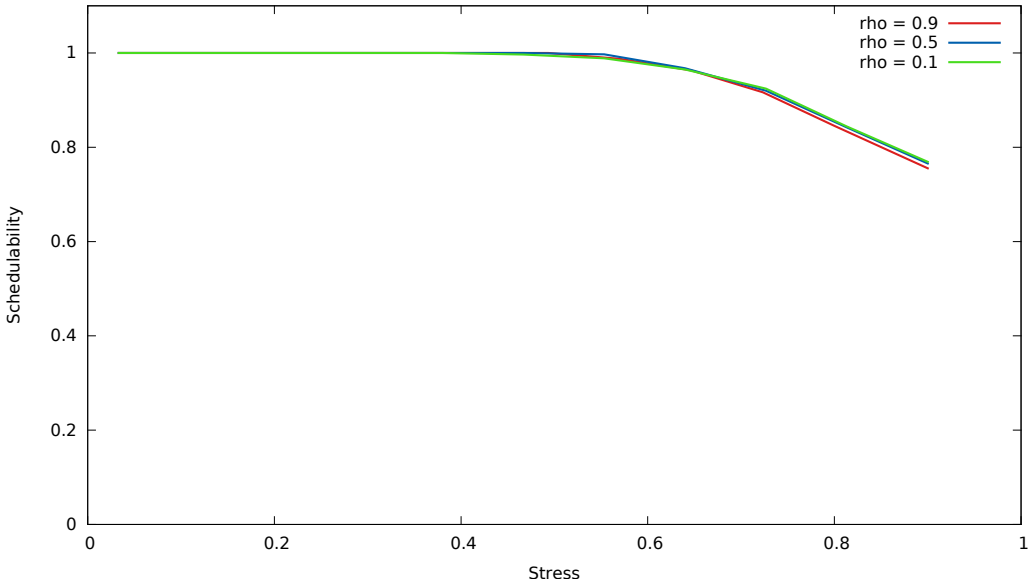
where V, E is the number of nodes, edges respectively and M is the number of processors.

8.5 Experiments

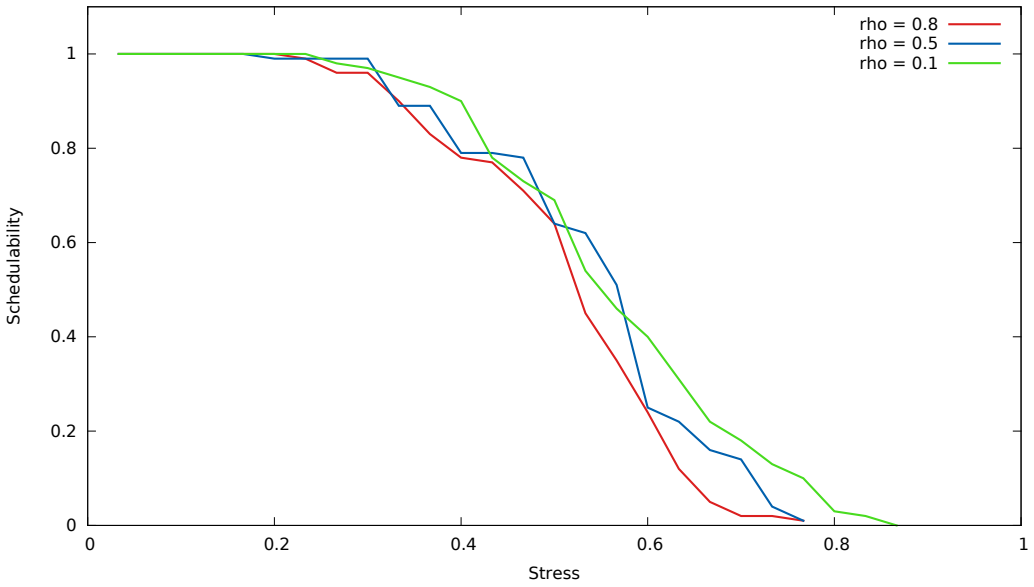
We have performed experiments of measuring the success rate of the algorithm for random generated ordinary and mixed-critical benchmarks that have different level of *normalized stress*, which is a peak resource utilization metric – see [77, 83] – ranging from 0 to 100 %. For mixed-criticality experiments, the stress for both modes of execution was maintained equal. We assumed instances with 10 jobs and no precedence constraints.

Experiments for three different values of ρ were made: 0.1, 0.5 and 0.8, where ρ is the ratio between the stress due to δ-jobs only and the stress due to all jobs. As expected, the mixed-criticality instances are much harder to solve than ordinary ones by the same algorithm. In future work it will be interesting to implement an exact algorithm, *e.g.,* using SMT solvers, to evaluate the optimality of our algorithm experimentally.

We noticed that, counter-intuitively, no significant sensitivity to the value of ρ was detected. A possible reason is that ρ appears to have only a weak connection to the ratio between δ and average task execution time. In future work a better load-related metric for the proportion of interference in the total workload will be investigated.



(a) Ordinary Benchmarks



(b) Mixed-critical Benchmarks

FIGURE 8.11: Schedulability results for random benchmarks

Chapter 9

Conclusion

In this thesis, we have studied schedulability and its correctness in mixed-criticality systems. An MC-system is a real-time system having different applications of different levels of criticality integrated on the same computation platform. Although this integration provides many benefits in terms of reduced cost, power consumption, size and weight, yet at the same time it adds difficulties in certification and scheduling due to having applications of different levels of criticality (e.g., safety critical and mission critical) share resources.

We presented new results concerning the algorithmic complexity of correctness testing algorithms for scheduling a fixed set of dual-criticality jobs on single processor. The procedure of testing the correctness of scheduling policies is typically part of the scheduling algorithm itself. Its complexity has a direct impact on problem solving complexity. Since the problem has been proven *at least* NP-hard, a major question is whether it is *at most* in the complexity class NP [4]. In Chapter 4 we refuted a lemma that implies that the cost of a single scenario is, in general, polynomial. This refuted the original argument of [4] for the problem being in class NP. Nevertheless, in [47] an erratum of [4] a higher order polynomial cost was established, and hence the problem is indeed in class NP.

Two characteristics related to correctness testing, are sustainability and predictability. Sustainability has been studied and well formalized in the literature [48, 49] for both the single and mixed-critical scheduling policies. Although sustainability implicitly implied predictability in single criticality systems, we have given an example of an MC-policy that was proven to be MC-sustainable yet is not predictable. We have shown that testing for correctness using simulation based tests can be problematic if a scheduling policy is not predictable.

Acknowledging the difficulty of proving an MC-scheduling policy predictable and the difficulty of testing by simulation of worst case execution scenarios in non-predictable policies, we proposed a weaker form of predictability that covers a larger class of scheduling policies. We have also shown that the well known class of FPM scheduling policies in dual-criticality, single processor case is not predictable but weakly-predictable.

We proposed a canonical correctness test that is applicable for weakly predictable policies. The correctness test verifies the correctness of the policy by evaluating it for a small number of basic scenarios. We proved that the canonical test can be applicable in such cases and we studied the computational complexity showing that this special class belongs to NP. We showed that these results can be extended for multiprocessor platforms in the case of FPM, policies having FPM-equivalent tables.

In Section 5.4 we have introduced a new "economical correctness testing" procedure that applies to a "reasonable" restriction of FPM. The proposed test consists of transforming a given scheduling policy to a time-triggered policy. Though FPM is a mode-switched policy, the economical test for it has a complexity as if it were an ordinary mode-ignorant FP policy, it is just $O(n \log(n))$, whereas the "canonical testing" of FPM is $O(n^2 \log n)$. It is fair to mention, however, that, by contrast, the canonical testing procedure is applicable to a wider class of policies.

It would be interesting to experimentally evaluate, for those task systems that can be modeled as fixed job systems, the effectiveness of simulation compared to analytical response time approaches. One way is to test how often would a system be deemed schedulable by an exact simulation based test but be found "non-schedulable" by an analytical response time or an utilization-based formula.

We introduced an STTM scheduling algorithm for mixed criticality systems with multiple levels of criticalities. By means of an example, we showed that our policy is able to schedule instances that require dynamic priorities and cannot be solved by FPM polices. Experimental results showed that the presented algorithm outperforms two of the state of the art algorihtms.

Automatic fault detection and recovery is important for the correctness and stability of autonomous systems. In an attempt to demonstrate that a mixed-criticality scheduling policy can be used as a recovery strategy, we integrate our algorithm with a diagnoser and a controller to form an FDIR component that is used to detect and recover failures in a component-based system.

Computational demands are increasing and multiprocessor are used more often. The extension of our algorithm to the multiprocessor case is an interesting future work. One simple solution is to use one of the algorithms discussed in Chapter 2 to statically allocate

jobs to different cores, and use PBEDF to schedule each set separately. Theoretical analysis and proof of dominance can add value to the experimental results presented. Yet another interesting extension is to provide the lower criticality jobs some of their execution demands, instead of discarding them after a mode change.

Finally, we have proposed a scheduling algorithm and a design flow for timing-critical multi-core applications, taking into account coarse-grained interference, using the interference from the controlling run-time environment as an example. In our design flow, we demonstrate the concept of using task automata as a concurrency language, which can be used to program the custom resource managers, such as mixed-criticality ones.

Different directions for future work can be taken to extend the design workflow presented in Chapter 8. Missing features, such as run-time environment to support task migration and dropping of tasks can be studied. The interference model can be extended to other resources (*e.g.,* buses and peripherals) and to more general task controllers and models of computation. One can investigate how to improve the non-preemptive scheduler for better support of mixed criticality. For reference the implementation of exact algorithm with exhaustive search can be considered. List scheduling can be replaced by topological permutation scheduling as it is a more powerful offline global fixed-priority heuristic for the case where there is no preemption and jobs have non-zero arrival times [82].

Appendix A

Proof of Time-triggered Transformation Algorithm

A.1 Proof of Direct Correctness

The aim of this appendix is to prove that if the original policy $\mathcal{P}$ is correct and reasonable then the transformed policy $\mathcal{T}(\mathcal{P})$ is also correct. To do so, we need to give some definitions and support lemmas. Let $TT_J^{HI^*(LO|HI\text{-}J')}$ be the termination time of J in **HI*** obtained from $\mathcal{T}(\mathcal{P})$ (respectively, **LO**, **HI-J′** obtained from $\mathcal{P}$).

Lemma A.1. *If at any time we switch from **LO** to **HI***, then all the non-terminated jobs will have enough time reserved in **HI*** to terminate their work.*

Before presenting the proof, first, let us comment that, according to our rules to construct **HI***, no HI jobs get disabled forever because eventually Rule (5.4.1a) becomes true, since all LO jobs eventually terminate. Thus, all HI jobs get a total time C^E reserved in **HI***. Consequently, if a job switches at time t, then all HI jobs are guaranteed to get $C^E - T_j^{HI*}(t)$, but need to get at least $C^E - T_j^{LO}(t)$.

Therefore the lemma can be equivalently stated as follows:

*no non-switched HI job makes more progress in **HI*** than in **LO**.*

Formally:

$$\forall t \ , \ T_j^{LO}(t) < C_j^N \Rightarrow T_j^{LO}(t) \geq T_j^{HI*}(t)$$

Proof of Lemma A.1. At time $t = 0$ the lemma thesis is obviously true, and with progress of time it can be invalidated only during the time when a job is scheduled in **HI***. However, as long as $T_j^{LO}(t) < C_j^N$ job J_j can only be scheduled when either (5.4.1b) or (5.4.1c) is true, but they both imply that we have $T_j^{LO}(t) \geq T_j^{HI*}(t)$. □

Definition A.2 (Busy Interval)**.** Consider a work-conserving policy and an instance **J**. A *busy interval* is an open time interval (τ_1, τ_2) in $\mathcal{S}$ that is a maximal time interval where the set of ready jobs is never empty (assuming jobs that are disabled are not considered ready).

We denote by BI the set of jobs that run in a given busy interval. In between busy intervals, there are closed, sometimes single-point, *idle intervals*. For **HI***, we would like to distinguish between two types of idle intervals. A *blocked interval* if it is idle and inside this interval there are HI jobs that have arrived and not yet terminated, but are disabled because neither of the rules (5.4.1a), (5.4.1b), (5.4.1c) is true. An *empty interval* where the the job queue is empty and there are no ready HI jobs to schedule.

For instance in Fig. 5.4 in **HI*** there are two busy intervals: (0,8) and (8,11), thus we have a blocked interval of size 0 at time 8. This blocked interval appears under the following circumstances. Immediately before time 8, J_1 is enabled by Rule (5.4.1a) while J_2 is disabled. Then at time 8, J_1 gets disabled (because it terminates) while immediately after that time J_2 is enabled by Rule (5.4.1c) to continue its execution after that time.

The following proposition is well-known for fixed-priority policies, but needs to be re-established because we added the rules that can disable jobs.

Lemma A.3. *If J^{least} is the least priority (*i.e., *the latest-deadline) HI job, then it terminates at the end of some busy interval BI^{HI*}.*

Proof. Let us assume by contradiction that J^{least} terminates inside a busy interval at time t. This means that at time t there is another enabled job (by definition of busy interval). If that is so, then J^{least}, having the least priority, should not be running at time t. □

Lemma A.4. *Let $BI^{HI*} : (a, b)$ be a busy interval in **HI***. At time a, the set of non-terminated HI jobs is the same in tables **LO** and **HI***, and for each job in this set, the job's cumulative execution progress until time a in **LO** is the same as in **HI***.*

Proof. Consider time a. The lemma thesis is obvious for any job that did not arrive yet, so in the sequel we consider only those jobs that have arrived.

If a HI job J does not terminate before time a in **LO** then it is non-terminated in **HI*** before that time as well by Lemma A.1. In addition, by the same lemma we have:

$$T_J^{HI*}(a) \leq T_J^{LO}(a) \tag{A.1.1}$$

On the other hand, if job J is non-terminated in **HI*** by time a, then the fact that it is not enabled at time a (by lemma condition) implies that Rule (5.4.1a) is false and hence the job is non-terminated in **LO** as well. Combined with the earlier observations, we conclude that the sets of non-terminated jobs at time a in these two tables are equal. In addition, also Rule (5.4.1b) is false, which means:

$$T_J^{HI*}(a) \geq T_J^{LO}(a) \tag{A.1.2}$$

Combining (A.1.1) and (A.1.2) we have the equality of the cumulative progress. □

Corollary A.5. *Let $BI^{HI*} : (a, b)$ be a busy interval in which some job switches i.e. reaches C_j^N progress. Let J_s be the first such job, and let t_s be the time at which the switch occurs. Then during the interval (a, t_s) tables **HI***, **HI-**J_s and **LO** are identical.*

Proof. Notice that **HI-**J_s and **LO** are equal by construction in $(0, t_s)$ and hence in (a, t_s) as well. Let us compare **LO** and **HI***. At time a the set of non terminated jobs in these two tables are equal (Lemma A.4). In interval (a, t_s) no job switched yet, therefore all the jobs that run in **HI*** should satisfy Rule (5.4.1c), which is due to the fact that the other two rules imply that some job has already switched. As long as Rule (5.4.1c) holds, the **HI*** table replicates the **LO** table, and because it fills time interval (a, t_s) continuously, as $t_s \in BI^{HI*}$, we have proved our corollary. □

Theorem A.6. *Let J^{least} be the least priority HI job in the priority table applied in the HI mode. (Note that in the reasonable policy this is always a latest-deadline HI job). Then*

$$\exists J' \ : \ TT_{J^{least}}^{HI*} \leq TT_{J^{least}}^{HI\text{-}J'}$$

Proof. Let $BI^{HI*} = (a, b)$ be the busy interval in which J^{least} terminates. By Lemma A.3, $TT_{J^{least}}^{HI*} = b$. By Lemma A.4, job J^{least} is not yet switched at start of this interval, and since this job terminates at the end of BI^{HI*}, we know also that it switches inside this interval as well, so Corollary A.5 applies for this interval.

Let us assume that $BI^{HI*} = (a, b)$ is followed by an empty interval, *i.e.,* an idle interval which appears due to termination of all HI jobs that have arrived so far. Because in this case all the jobs that are ready in interval BI^{HI*} have terminated by time b, we have:

$$b = a + \sum_{j \in BI^{HI*}} (C_j^E - T_j^{HI*}(a))$$

Let J_s be the first job to switch in BI^{HI*}, at time t_s. By Lemma A.4 and Corollary A.5, we have that the same jobs, with the same remaining execution time as in **HI*** will run

from time a in **HI-**J_s before the switch and, by construction after the switch the same set of jobs as in **HI*** may arrive and become ready, and in **HI-**J_s, under EDF policy, the ready jobs will occupy the processor until all of them have terminated – which is the same behavior as for **HI*** in this case. Therefore $BI^{HI*} = BI^{HI\text{-}J_s}$ and J^{least}, being the least-priority job, will terminate at time b in both tables.

Let us now examine the other case, in which $BI^{HI*} = (a, b)$, the busy interval where J^{least} terminates, is followed by a blocked interval , *i.e.,* the idle interval which appears because at time b the rules for table **HI*** have disabled all ready jobs. Also in this case J^{least} by our hypothesis and Lemma A.3 will terminate at time b, but in this case by construction not all jobs of BI^{HI*} terminate by time b:

$$b < a + \sum_{j \in BI^{HI*}} (C_j^E - T_j^{HI*}(a)) \tag{A.1.3}$$

Let J_s be the first job to switch in BI^{HI*}, at time t_s. Again by Lemma A.4 and Corollary A.5 we observe the same initial state and subsequent behavior in tables **HI*** and **HI-**J_s of all non-terminated HI jobs during the time interval $(a, t_s]$. So we conclude that all jobs of BI^{HI*} run in **HI-**J_s after time a continuously, and at time a their total remaining work is equal to:

$$\sum_{j \in BI^{HI*}} (C_j^E - T_j^{HI*}(a))$$

In line with equation (A.1.3), in order to complete this workload, table **HI-**J_s has to continue execution after time b. New jobs may arrive before the termination of the busy interval $BI^{HI\text{-}J_s}$. This busy interval executes all these jobs, J^{least} being the last one to terminate. So we have:

$$BI^{HI*} \subseteq BI^{HI\text{-}J_s}$$

and

$$TT_{J^{least}}^{HI\text{-}J_s} \geq a + \sum_{j \in BI^{HI*}} (C_j^E - T_j^{HI*}(a)) \tag{A.1.4}$$

Combining (A.1.3) and (A.1.4), and observing that $TT_{J^{least}}^{HI*} = b$, we have that also in this case in **HI-**J_s the least-priority job terminates no earlier than in **HI***. This completes the proof of Theorem A.6. □

Theorem A.7 (Transformation Correctness)**.** *For a given problem instance, if the original policy $\mathcal{P}$ is correct and reasonable then the transformed policy $\mathcal{T}(\mathcal{P})$ is also correct.*

Proof. From Lemma A.1 we know that in any possible scenario all the HI jobs will have enough time allocated in HI^* to terminate. The termination time of J^{least} is guaranteed

to meet the deadline due to the hypothesis that it meets deadline in the original policy and Theorem A.6. Now let us prove that also the HI jobs with higher priority in the EDF table PT_{HI} meet their deadlines. Let $J^{\overline{least}}$ be the next least priority HI job after J^{least} in the EDF table. Let $\mathbf{J}$ be the currently examined problem instance and let $\overline{\mathbf{J}}$ be the instance obtained from $\mathbf{J}$ by reducing the criticality of J^{least} to LO. Since J^{least} was the HI job with lowest priority, it only executed when no other job was ready to execute. For this reason, the HI-mode table $\overline{\mathbf{HI}*}$ obtained for this new instance coincides with **HI*** except that the intervals where J^{least} was running are idled. So, $J^{\overline{least}}$ will terminate in **HI*** at the same time as in $\overline{\mathbf{HI}*}$, where by Theorem A.6 applied to instance $\overline{\mathbf{J}}$ it will terminate no later than the latest termination under the original policy. Obviously, also the latest termination of the original policy for job $J^{\overline{least}}$ is the same for both $\mathbf{J}$ and $\overline{\mathbf{J}}$. Because by our hypothesis this policy is correct we conclude that $J^{\overline{least}}$ meets its deadline. Iterating this reasoning recursively, we argue that all HI jobs meet their deadline in **HI***, and thus we have our thesis. □

A.2 Proof of Reverse Correctness

In this section we prove the reverse correctness of the transformation algorithm, *i.e.,* that for a reasonable original policy we have that $\mathcal{T}(\mathcal{P})$ can succeed only if the original policy succeeds.

Similarly to the previous section, we first give some supplementary definitions and lemmas.

The *total remaining workload* when the original policy executes basic scenario sc at time t is defined as:

$$WL^{sc}(t) = \sum_{j \in \mathbf{J}} (C_j(\chi_j^{sc}) - T_j^{sc}(t))$$

where χ_j^{sc} is the criticality behavior shown by J_j in scenario sc and $C_j(\chi_j^{sc})$ is the execution time of J_j in sc. Since sc is a basic scenario, $C_j(\chi_j^{sc})$ can be either C_j^N or C_j^E. Similarly the total remaining *HI-job workload* is given as:

$$WL_{\text{HI}}^{sc}(t) = \sum_{j \in \mathbf{J}: \chi_j = \text{HI}} (C_j(\chi_j^{sc}) - T_j^{sc}(t))$$

For table **HI*** we have:

$$WL^{HI*}(t) = WL_{\text{HI}}^{HI*}(t) = \sum_{j \in \mathbf{J}: \chi_j = \text{HI}} (C_j^E - T_j^{HI*}(t))$$

Lemma A.8. *Given a reasonable original policy, we have that:*

$$\forall sc, t \quad WL^{HI*}(t) \geq WL^{sc}_{HI}(t)$$

Proof. Before the mode switch in sc, for any HI job j that did not terminate by time t in sc, we have that $C_j(\chi^{sc}_j) \leq C^E_j$ by construction and $T^{sc}_j(t) \geq T^{HI*}_j(t)$ by Lemma A.1. On the other hand, for a HI job that has already terminated we have that $C_j(\chi^{sc}_j) - T^{sc}_j(t) = 0$. By the above remarks we have $C^E_j - T^{HI*}_j(t) \geq C_j(\chi^{sc}_j) - T^{sc}_j(t)$ for all HI jobs j.

After the switch in sc, a reasonable policy will always execute a HI job when some HI workload is ready (because the EDF policy is work-conserving and LO jobs have been dropped). Next to this, observe that some jobs that are ready in sc may be at the same time disabled in **HI***. Thus after the switch, the total HI workload will decrease in sc at least as fast as in HI^* □

Recall that a reasonable policy after the mode switch becomes priority-based and schedules HI jobs using the EDF priority table of HI jobs. Therefore, in this table we can identify the least priority job J_{least}.

Theorem A.9 (Worst Case Scenario)**.** *Let us consider a reasonable original policy. Then, for the least priority job J_{least} we have:*

$$\forall sc', \quad TT^{HI\text{-}J_s}_{least} \geq TT^{sc'}_{least}$$

*where J_s is the first job to switch in the busy interval of **HI*** where J_{least} terminates and sc' is either the LO basic scenario or any job-specific HI scenario (* i.e., *from the canonical basic set).*

In other words, HI-J_s is the worst-case scenario for J_{least}.

Proof. In this proof we will use three **observations**:

1. after the switch we have $WL^{sc}_{\text{HI}} = WL^{sc}$.
2. consider two HI-job specific scenarios sc and sc' and some time instant t at or after the switching time of both scenarios; if at time t J_{least} did not yet terminate in neither of the two scenarios and $WL^{sc}(t) \geq WL^{sc'}(t)$, then $TT^{sc}_{least} \geq TT^{sc'}_{least}$; (this is so because after the switch a reasonable policy applies EDF, and for a fixed-priority policy the remaining workload has a monotonic impact on the termination time of the least priority job).

3. In the theorem statement we can ignore the case where sc' is the LO scenario without loss of generality. This is because there always exists a HI scenario where J^{least} terminates at the same time or later, for example HI-J^{least}.

Let t_s be the time when J_s switches in **HI***. We know by Corollary A.5 that $WL^{HI*}(t_s) = WL_{\text{HI}}^{HI\text{-}J_s}(t_s)$. Then, by Lemma A.8:

$$\forall sc' \qquad WL_{\text{HI}}^{HI\text{-}J_s}(t_s) \geq WL^{sc'}(t_s) \tag{A.2.1}$$

i.e., no scenario has more workload at time t_s than the scenario $HI\text{-}J_s$.

In the rest of the proof we assume that $t_{s'}$ is the switch time of another HI-job specific basic scenario $sc' = HI\text{-}J_{s'}$ and we compare that scenario to $sc = HI\text{-}J_s$.

For the scenarios where $t_{s'} \leq t_s$ the statement of the theorem is proved by the above stated Observation 2 and Equation (A.2.1), as we have established the workload inequality for a time t_s that is at or later than the switch in the both scenarios.

Let us prove the theorem statement for the other case, $t_{s'} > t_s$. Let $t_{least} = TT_{least}^{LO}$, *i.e.,* the time at which J_{least} terminates in the LO scenario. Note that we can ignore the case $t_{least} < t_{s'}$, as in this case $TT_{least}^{sc'} = TT_{least}^{LO}$ and Observation 3 applies. So, we can assume $t_{s'} \leq t_{least}$. Due to this assumption, we also have: $t_{s'} \leq TT_{least}^{HI*}$ and $t_{s'} \leq TT_{least}^{HI\text{-}J_s}$. Adding to this that $t_s < t_{s'}$ we see that $t_{s'}$ falls inside the busy interval where J_{least} terminates in the end, both for **HI*** and $HI\text{-}J_s$. By construction, t_s belongs to the same busy interval BI^{HI*} that ends at TT_{least}^{HI*}, thus WL^{HI*} will constantly decrease in this interval. At time $t_{s'}$, we will have $WL^{HI*}(t_{s'}) = WL^{HI*}(t_s) - \mid (t_s, t_{s'}) \mid$. By a similar reasoning on the busy interval $BI^{HI\text{-}J_s}$, we have $WL^{HI\text{-}J_s}(t_{s'}) = WL^{HI\text{-}J_s}(t_s) - \mid (t_s, t_{s'}) \mid$.

Thus, using equality $WL^{HI*}(t_s) = WL_{\text{HI}}^{HI\text{-}J_s}(t_s)$, which we established earlier, we have:

$$\begin{aligned} WL^{HI\text{-}J_s}(t_{s'}) &= WL^{HI\text{-}J_s}(t_s) - \mid (t_s, t_{s'}) \mid \\ &= WL^{HI*}(t_s) - \mid (t_s, t_{s'}) \mid \\ &= WL^{HI*}(t_{s'}) \end{aligned}$$

Therefore, for time $t_{s'}$ we can repeat the same reasoning as we did for time t_s in the case $t'_s \leq t_s$, which concludes the proof.

□

Theorem A.10 (Reverse Correctness)**.** *For a given problem instance on single-processor, under the assumption that the original policy is reasonable and weakly predictable, we have that if the policy $\mathcal{T}(\mathcal{P})$ is correct then the original policy is correct as well.*

Proof. Our thesis can be rewritten as:

$$(\forall j \quad TT_j^{HI*} \leq D_j) \quad \Rightarrow \quad (\forall sc, \forall i \quad TT_i^{sc} \leq D_i)$$

We prove the theorem for $J_i = J_{least}$ and then extend this argument from J_{least} to other jobs J_i by induction, in the same way as we did in the proof of Theorem A.7.

Suppose by contradiction that J_{least} misses its deadline in the original policy while all jobs meet their deadlines in the transformed policy. We have:

$$TT_{least}^{HI*} \leq D_{least} < TT_{least}^{HI\text{-}J_s} \tag{A.2.2}$$

where $HI\text{-}J_s$ is the worst case scenario for J_{least} according to Theorem A.9. We distinguish two cases:

1. J_{least} **terminates before an "empty interval".**

   By the reasoning of the proof of Theorem A.6, we have:

   $$TT_{least}^{HI*} = TT_{least}^{HI\text{-}J_s}$$

   which contradicts (A.2.2).

2. J_{least} **terminates before a "blocked interval ".** Considering $BI^{HI*} = (a, b)$, as in the proof of Theorem A.6, and observing that, by Lemma A.4, $T_j^{HI\text{-}J_s}(a) = T_j^{HI*}(a)$ we have that:

   $$TT_{least}^{HI\text{-}J_s} = a + \sum_{j \in BI^{HI\text{-}J_s}} (C_j^E - T_j^{HI*}(a)) \tag{A.2.3}$$

   Let J_e be the last job to terminate in **HI***. For this job, by construction:

   $$TT_e^{HI*} \geq a + \sum_{j \in \mathbf{J}} (C_j^E - T_j^{HI*}(a)) \tag{A.2.4}$$

   The right side of Equation (A.2.4) is no less than the right side of Equation (A.2.3). Therefore, $TT_e^{HI*} \geq TT^{HI\text{-}J_s}$. Also, in EDF: $D_{least} \geq D_e$. From these observations and (A.2.2), we have:

   $$TT_e^{HI*} \geq TT_{least}^{HI\text{-}J_s} > D_{least} \geq D_e$$

thus J_e will miss its deadline in **HI***, which contradicts the theorem assumptions.

□

Acronyms

CA Certification Authority.

CCT Canonical Correctness Test.

DAL Design Assurance Levels.

ECT Economical Correctness Test.

EDF Earliest Deadline First.

FDIR Fault Detection Isolation and Recovery.

FPM Fixed Priority per Mode.

MC Mixed Criticality.

MCEDF Mixed Criticality Earliest Deadline First.

MCPI Mixed Criticality Priority Improvement.

OCBP Own Criticality Based Priority.

PBEDF Push-Back Earliest Deadline First.

STTM Single Time Table per Mode.

TA Timed-Automata.

WCET Worst Case Execution Time.

List of Symbols

ℓ	A criticality level.
T_ℓ	Time table for criticality level ℓ.
χ_{mode}	The criticality mode of the system.
J^E	The emergency set.
$\mathcal{I}$	An instance containing a set of jobs.
J	A job in an instance.
A	Arrival time of a job.
X	Criticality level of a job.
D	Deadline of a job.
n	Number of jobs in an instance.
L	Number of criticality levels in an instance.
c	A scenario of an instance.
$\mathcal{S}$	A schedule for a given scenario.
ℓ	The criticality level of a scenario.
ts	A time slot in a time table.
C^N	Normal WCET estimate of a job.
C^E	Emergency WCET estimate of a job.
C^U	Uncertain execution estimate of a job.

Bibliography

[1] Seo-Hyun Jeon, Jin-Hee Cho, Yangjae Jung, Sachoun Park, and Tae-Man Han. Automotive hardware development according to iso 26262. In *13th International Conference on Advanced Communication Technology (ICACT2011)*, pages 588–592. IEEE, 2011.

[2] Leslie A Johnson et al. Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk, October*, 199, 1998.

[3] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium*, RTSS'07, pages 239–243. IEEE, 2007.

[4] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Trans. Comput.*, 61(8):1140 –1152, aug. 2012.

[5] Kunal Agrawal and Sanjoy Baruah. Intractability issues in mixed-criticality scheduling. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[6] Iulia Dragomir, Simon Iosti, Marius Bozga, and Saddek Bensalem. Designing systems with detection and reconfiguration capabilities: a formal approach. In *International Symposium on Leveraging Applications of Formal Methods*, pages 155–171. Springer, 2018.

[7] Alan Burns and Sanjoy Baruah. Timing faults and mixed criticality systems. In *Dependable and Historic Computing*, pages 147–166. Springer, 2011.

[8] Sanjoy Baruah and Alan Burns. Implementing mixed criticality systems in ada. In *International Conference on Reliable Software Technologies*, pages 174–188. Springer, 2011.

[9] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Multiprocessor scheduling of precedence-constrained mixed-critical jobs. In *Real-Time Distributed*

Computing (ISORC), 2015 IEEE 18th International Symposium on, pages 198–207. IEEE, 2015.

[10] Hang Su and Dakai Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 147–152. IEEE, 2013.

[11] Pontus Ekberg and Wang Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-time systems*, 50(1):48–86, 2014.

[12] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 147–155, July 2008.

[13] Sanjoy K. Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium*, RTAS'10, pages 13–22. IEEE, 2010.

[14] Haohan Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 183–192, Nov 2010.

[15] Sanjoy Baruah, Alan Burns, and Robert I Davis. An extended fixed priority scheme for mixed criticality systems. *Proc. ReTiMiCS, RTCSA*, pages 18–24, 2013.

[16] Yao Chen, Kang G Shin, and Huagang Xiong. Generalizing fixed-priority scheduling for better schedulability in mixed-criticality systems. *Information Processing Letters*, 116(8):508–512, 2016.

[17] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 93–102. IEEE, 2013.

[18] Rhan Ha and J. W S Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. Int. Conf. Distributed Computing Systems*, pages 162–171, Jun 1994.

[19] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium*, RTSS '11, pages 3–12. IEEE, 2011.

[20] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Time-triggered mixed-critical scheduler. *Proc. WMC, RTSS*, pages 67–72, 2013.

[21] Jens Theis, Gerhard Fohler, and Sanjoy Baruah. Schedule table generation for time-triggered mixed criticality systems. *Proc. WMC, RTSS*, pages 79–84, 2013.

[22] Mathieu Jan, Lilia Zaourar, Vincent Legout, and Laurent Pautet. Handling criticality mode change in time-triggered systems through linear programming. In *Ada User Journal, Proc of Workshop on Mixed Criticality for Industrial Systems (WMCIS'2014)*, volume 35, pages 138–143, 2014.

[23] S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34–43, Nov 2011.

[24] Thomas Fleming. *Extending mixed criticality scheduling.* PhD thesis, University of York, 2013.

[25] Huang-Ming Huang, Christopher Gill, and Chenyang Lu. Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):126, 2014.

[26] Qingling Zhao, Zonghua Gu, and Haibo Zeng. Integration of resource synchronization and preemption-thresholds into edf-based mixed-criticality scheduling algorithm. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 227–236, Aug 2013.

[27] A Burns, Robert Davis, et al. Adaptive mixed criticality scheduling with deferred preemption. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 21–30. IEEE, 2014.

[28] Sanjoy K Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *European Symposium on Algorithms*, pages 555–566. Springer, 2011.

[29] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proc. ECRTS'12*, pages 145–154. IEEE, 2012.

[30] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *Journal of the ACM (JACM)*, 62(2):14, 2015.

[31] Sanjoy Baruah and Zhishan Guo. Mixed-criticality scheduling upon varying-speed processors. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 68–77. IEEE, 2013.

[32] Sanjoy Baruah and Zhishan Guo. Scheduling mixed-criticality implicit-deadline sporadic task systems upon a varying-speed processor. In *2014 IEEE Real-Time Systems Symposium*, pages 31–40. IEEE, 2014.

[33] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium*, RTSS'09, pages 291–300. IEEE, 2009.

[34] Karthik Lakshmanan, Dionisio De Niz, Ragunathan Rajkumar, and Gines Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 169–178. IEEE, 2010.

[35] Owen R Kelly, Hakan Aydin, and Baoxian Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1051–1059. IEEE, 2011.

[36] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[37] D. de Niz and L.T.X. Phan. Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 111–122, April 2014.

[38] Romain GRATIA, Thomas ROBERT, and Laurent PAUTET. Adaptation of run to mixed-criticality systems. *JRWRTC 2014*, page 25, 2014.

[39] Neil C Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times.* Citeseer, 1991.

[40] Zaid Al-bayati, Qingling Zhao, Ahmed Youssef, Haibo Zeng, and Zonghua Gu. Enhanced partitioned scheduling of mixed-criticality systems on multicore platforms. In *The 20th Asia and South Pacific Design Automation Conference*, pages 630–635. IEEE, 2015.

[41] Haohan Li and Sanjoy K. Baruah. Outstanding paper award: Global mixed-criticality scheduling on multiprocessors. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012*, 2012.

[42] Sanjoy K Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, 2004.

[43] Lui Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, Sep 1990.

[44] A. Burns. The application of the original priority ceiling protocol to mixed criticality systems. In L. George and G. Lipari, editors, *ReTiMiCS, RTCSA*, pages 7–11, 2013.

[45] Vicent Brocal, Patricia Balbastre, Rafael Ballester, and Ismael Ripoll. Task period selection to minimize hyperperiod. In *ETFA2011*, pages 1–4. IEEE, 2011.

[46] Ismael Ripoll and Rafael Ballester-Ripoll. Period selection for minimal hyperperiod in periodic task systems. *IEEE Transactions on Computers*, 62(9):1813–1822, 2013.

[47] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Pontus Ekberg, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Erratum for scheduling real-time mixed-criticality jobs. 2018.

[48] Sanjoy K. Baruah and Alan Burns. Sustainable scheduling analysis. In *Real-Time Systems Symposium (RTSS 2006)*, pages 159–168, 2006.

[49] Zhishan Guo, Sai Sruti, Bryan C Ward, and Sanjoy K Baruah. Sustainability in mixed-criticality scheduling. 2017.

[50] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Euromicro Conf. on Real-Time Systems*, ECRTS'13, pages 93–102. IEEE, 2013.

[51] Dario Socci. *Scheduling of Certifiable Mixed-Criticality Systems*. PhD thesis, VERIMAG Research Center, Université Grenoble Alpes, 2016.

[52] Rany Kahil, Dario Socci, Peter Poplavko, and Saddek Bensalem. Algorithmic complexity of correctness testing in mc-scheduling. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 180–190. ACM, 2018.

[53] Rany Kahil, Dario Socci, Peter Poplavko, and Saddek Bensalem. Predictability in mixed-criticality systems. In *to appear at RTCSA2018*, 2018.

[54] Alan Burns and Robert I Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys (CSUR)*, 50(6):82, 2017.

[55] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, Inc., 2000.

[56] Haohan Li and Sanjoy Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Intern. Conf. on Embedded Software*, EMSOFT '10, pages 99–108. ACM, 2010.

[57] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[58] Stavros Tripakis. Fault diagnosis for timed automata. In *International symposium on formal techniques in real-time and fault-tolerant systems*, pages 205–221. Springer, 2002.

[59] Franck Cassez and Stavros Tripakis. Fault diagnosis of timed systems, 2009.

[60] Martin Wirsing, Matthias M. Hölzl, Mirco Tribastone, and Franco Zambonelli. ASCENS: engineering autonomic service-component ensembles. In *FMCO'11*, pages 1–24, 2011.

[61] Sagar Chaki and David Kyle. DMPL: Programming and verifying distributed mixed-synchrony and mixed-critical software. Technical report, Carnegie Mellon University, 2016.

[62] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2013.

[63] Peter Poplavko, Dario Socci, Paraskevas Bourgos, Saddek Bensalem, and Marius Bozga. Models for deterministic execution of real-time multiprocessor applications. In *DATE'15*, 2015.

[64] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[65] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard, and Claire Pagetti. Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset. In *19th International Conference on Real-Time and Network Systems*, 2011.

[66] Sander Stuijk, Marc Geilen, Bart D. Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *SAMOS'11*. IEEE, 2011.

[67] Georgia Giannopoulou, Peter Poplavko, Dario Socci, Pengcheng Huang, Nikolay Stoimenov, Paraskevas Bourgos, Lothar Thiele, Marius Bozga, Saddek Bensalem, Sylvain Girbal, Madeleine Faugere, Romain Soulat, and Benoit Dupont de Dinechin. DOL-BIP-critical: A tool chain for rigorous design and implementation of mixed-criticality multi-core systems. Technical Report 363, ETH Zurich, Laboratory TIK, Apr 2016.

[68] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. A timed-automata based middleware for time-critical multicore applications. In *Proc. SEUS'15*. IEEE, 2015.

[69] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times — a tool for modelling and implementation of embedded systems. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464. Springer, 2002.

[70] Elena Fersman, Pavel Krcál, Paul Pettersson, and Wang Yi 0001. Task automata: Schedulability, decidability and undecidability. *Inf. Comput.*, 205(8):1149–1172, 2007.

[71] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10. ACM, 2010.

[72] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Modeling mixed-critical systems in real-time BIP. In *ReTiMiCs'2013*, 2013.

[73] Saddek Bensalem, Marius Bozga, Jacques Combaz, and Ahlem Triki. Rigorous system design flow for autonomous systems. In *ISoLA'14*, pages 184–198, 2014.

[74] Rodolfo Pellizzoni, Bach Duy Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in cots-based embedded systems. In *RTSS'08*, pages 221–231, 2008.

[75] Sundararajan Sriram and Edward A. Lee. Determining the order of processor transactions in statically scheduled multiprocessors. *VLSI Signal Processing*, 15(3):207–220, 1997.

[76] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *RTSS '11*, pages 3–12. IEEE, 2011.

[77] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Time-triggered mixed-critical scheduler on single- and multi-processor platforms (revised version). Technical Report TR-2015-8, Verimag, 2015.

[78] Sanjoy Baruah. Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In *RTNS'12*, pages 11–19. ACM, 2012.

[79] M Perrotin, E. Conquet, P Dissaux, T. Tsiodras, and J Hugues. The TASTE toolset: turning human designed heterogeneous systems into computer built homogeneous software. In *ERTSS'10*, 2010.

[80] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

[81] Alexandros Zerzelidis and Andy J. Wellings. A framework for flexible scheduling in the RTSJ. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.

[82] M.J.M. Heijligers. *The Application of Genetic Algorithms to High-Level Synthesis.* PhD thesis, Univ. of Eindhoven, 1996.

[83] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Multiprocessor scheduling of precedence-constrained mixed-critical jobs. In *ISORC'15*, pages 198–207. IEEE, 2015.

[84] Hardik Shah, Andrew Coombes, Andreas Raabe, Kai Huang, and Alois Knoll. Measurement based wcet analysis for multi-core architectures. In *RTNS '14*. ACM, 2014.

[85] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *DATE'14*. EDAA, 2014.