TAMING UNCERTAINTY



ENABLING PREDICTABILITY

SYSTEMATIC TIMING ANALYSIS OF REAL-TIME SYSTEMS ACROSS COMPUTATION AND COMMUNICATION

SRINIDHI SRINIVASAN

Taming Uncertainty, Enabling Predictability

Systematic Timing Analysis of Real-Time Systems Across Computation and Communication

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr. S.K. Lenaerts, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op woensdag 01 juli 2026 om 16:00 uur

door

Srinidhi Srinivasan

geboren te Chennai, India

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

Voorzitter:	prof.dr.ir. J.F. Groote
Promotor:	prof.dr.ir. N. Meratnia
Co-promotor:	dr.ir. G.R.R.J.P. Nelissen

Promotiecommissieleden:	prof.dr. L. de Almeida (University of Porto)
	prof.dr. J.J. Chen (RWTH Aachen University)
	dr. M. Ashjaei (Mälardalen University)
	dr. S.S. Craciunas (Technical University of Denmark)
	dr.ir. M.C.W. Geilen

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Taming Uncertainty, Enabling Predictability

Systematic Timing Analysis of Real-Time Systems Across Computation and Communication

by Srinidhi Srinivasan

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-6739-3

Keywords: *real-time systems, timing analysis, schedulability analysis, response time analysis, time-sensitive networking, schedule abstraction graph, reachability analysis, self-suspending tasks, precedence constraints, time-aware shapers, credit-based shapers, real-time communication, state-space exploration*

The work presented in this thesis was carried out within the Interconnected Resource-aware Intelligent Systems (IRIS) group of the Department of Mathematics and Computer Science at Eindhoven University of Technology. This research was partially funded for a duration of one year by the AIMS5.0 project. The AIMS5.0 project is supported by the Chips Joint Undertaking and its members, including top-up funding by the National Funding Authorities of the participating countries, under Grant Agreement No. 101112089. The work presented in this thesis was conducted under the auspices of the research school IPA (Institute for Programming Research and Algorithmics). IPA Dissertation Series No. 2026-13.







Copyright © 2026 by Srinidhi Srinivasan, Eindhoven, the Netherlands.
Typeset using LaTeX
Cover design by Srinidhi Srinivasan (nidhi.2396s@gmail.com)
Printed by ProefschriftMaken.

To my brother,
whose support was a calm, steady hum,
never wavering
even when the world grew too loud.

SUMMARY

Real-time systems are systems in which correctness depends not only on producing the correct result but also on producing it within the required timing bounds. In many modern systems, including safety-critical and business-critical applications, violation of timing constraints can lead to unacceptable outcomes even when the underlying computation is correct. Ensuring that such systems consistently satisfy their timing requirements before deployment is therefore a fundamental challenge in the design and analysis of real-time systems.

Analysing the timing behaviour of these systems is challenging because their execution can vary across many possible scenarios. Variability in factors such as task release times, execution durations, precedence constraints among tasks, suspension intervals while tasks wait for remote computation or communication, can lead to a large number of possible execution scenarios, each potentially producing a different schedule and resulting in different timing properties. As systems become more complex, the number of such scenarios grows rapidly, making it difficult to reason about all possible outcomes in a systematic way.

The primary objective of this thesis is to improve the analysis of timing behaviour of realistic real-time systems. Modern systems often involve more complex task behaviour, where tasks may consist of multiple parts with precedence constraints or may temporarily pause while waiting for external operations. They also increasingly rely on communication networks to exchange data between different parts of the system. Together with the growing scale of such systems, these factors complicate the analysis of timing constraints. This thesis therefore proposes methods that enable systematic timing analysis across several challenging settings.

The first contribution of this thesis is related to the analysis of real-time systems that exhibit more expressive application-level task behaviour. In many practical systems, tasks cannot be described solely by simple recurring patterns: their acti-

vations may depend on events, interactions with other tasks, or delays between successive executions. Such behaviour increases the number and diversity of possible execution scenarios, making systematic timing analysis more difficult. This thesis develops methods to analyse the timing behaviour of real-time systems in which tasks may contain multiple parts with precedence constraints and may suspend while waiting for external operations. The proposed techniques model these precedence constraints and suspension intervals explicitly, and systematically reason about the execution scenarios that arise from such task behaviour.

The second contribution of the thesis concerns the analysis of timing behaviour in communication networks based on Time-Sensitive Networking (TSN). In these systems, the entities to be scheduled are network messages rather than processor jobs, and timing guarantees must be provided for packet transmissions across communication links. Analysing message latency in such networks is challenging because transmissions interact across multiple links, traffic shaping mechanisms regulate how packets are released and transmitted, and the internal structure of network switches influences how packets are queued and forwarded. These factors introduce complex dependencies that affect end-to-end communication delays. This thesis therefore investigates how communication timing properties in TSN networks can be analysed systematically. These methods provide analysis techniques for reasoning about how different traffic shaping mechanisms affect message latency, and support the analysis of both latency at individual network switches and end-to-end latency across communication paths.

The third contribution relates to the scalability of timing analysis for increasingly complex real-time systems. As the number of tasks, sources of timing variability, and interactions between system components grow, the number of execution scenarios that must be considered can increase significantly. This makes it difficult to analyse large systems while maintaining complete coverage of relevant scheduling possibilities. The thesis therefore investigates techniques that enable scalable timing analysis while preserving correctness. These techniques reduce the number of execution scenarios that must be explored while still ensuring complete coverage of relevant scheduling behaviours.

To enable these analyses, this thesis builds upon and enhances the Schedule Abstraction Graph (SAG) framework, originally introduced in [46, 48], which provides an abstraction-based approach for reasoning about scheduling behaviour. The core idea of the SAG is to represent the system's possible schedules as a graph which captures relevant execution scenarios without analysing each schedule individually. Instead of representing every concrete execution scenario, the framework groups scenarios with equivalent scheduling behaviour into abstract states and explores their transitions. This enables the systematic analysis of

timing behaviour across a wide range of execution scenarios.

Using this abstraction-based approach, the thesis extends the analysis to task models with precedence constraints and suspension behaviour, adapts the framework to analyse message transmissions in Time-Sensitive Networking, and develops techniques that improve the scalability of the analysis for larger systems. Taken together, these contributions improve the ability to analyse the timing behaviour of realistic real-time systems across several challenging settings.

SAMENVATTING

Real-time systems zijn systemen waarbij de correctheid niet alleen afhangt van het produceren van een correct resultaat, maar ook van het produceren van dit resultaat binnen de benodigde tijdsgrenzen. In veel moderne systemen, waaronder veiligheids- en bedrijfskritieke toepassingen, kunnen schendingen van tijdsbeperkingen leiden tot onacceptabele gevolgen, zelfs als de onderliggende berekening correct is. Om die reden is het waarborgen van de consistente naleving van deze tijdsbeperkingen alvorens inzet van zulke systemen een fundamentele uitdaging in het ontwerp en de analyse van real-time systems.

De analyse van het timingsgedrag van deze systemen is uitdagend, omdat de uitvoering ervan kan variëren, afhankelijk van vele mogelijke scenario's. Variabiliteit in factoren zoals taak vrijgavetijden, uitvoeringsduren, precedentierelaties tussen taken, onderbrekingsintervallen wanneer taken wachten op externe computaties of communicatie, kunnen elk leiden tot een groot aantal mogelijke uitvoeringsscenario's, waarvan elk mogelijk een andere planning produceert, die resulteert in andere timingseigenschappen. Naarmate systemen complexer worden, groeit het aantal van deze scenario's snel, wat redeneren over alle mogelijke uitkomsten op een systematische wijze bemoeilijkt.

Het primaire doel van dit proefschrift is het verbeteren van de analyse van timingsgedrag van realistische real-time systems. Moderne systemen omvatten veelal complex taakgedrag, waar taken uit verschillende delen met precedentierelaties kunnen bestaan of tijdelijk kunnen pauzeren, wachtende op externe handelingen. Ze hangen ook in toenemende mate af van communicatienetwerken, om data tussen verschillende delen van een systeem uit te wisselen. Samen met de toenemende schaal van zulke systemen, bemoeilijken deze factoren de analyse van tijdsbeperkingen. Om die reden introduceert dit proefschrift methodes die systematische timingsanalyse over verschillende uitdagende scenario's mogelijk maken.

De eerste contributie van dit proefschrift is gerelateerd aan de analyse van real-time systems waarvan de taken expressief applicatieniveaugedrag vertonen. In vele werkelijke systemen kunnen taken niet simpelweg met terugkerende patronen beschreven worden: hun activaties kunnen afhangen van evenementen, interacties met andere taken of vertragingen tussen opeenvolgende uitvoeringen. Zulk gedrag verhoogt het aantal en de diversiteit van de mogelijke uitvoeringsscenario's, wat systematische timingsanalyse bemoeilijkt. Dit proefschrift beschrijft methodes voor de analyse van timingsgedrag van real-time systems waarin taken uit meerdere delen met precedentierelaties mogen bestaan, en waarbij deze taken zichzelf mogen pauzeren, wachtende op externe handelingen. De voorgestelde technieken modelleren deze precedentierelaties en pauzeringsintervallen expliciet, en redeneren systematisch over de uitvoeringsscenario's die voortkomen uit zulk taakgedrag.

De tweede contributie van dit proefschrift betreft de analyse van timingsgedrag in communicatienetwerken gebaseerd op Time-Sensitive Networking (TSN). Binnen deze systemen zijn de entiteiten die gepland worden netwerkberichten, in plaats van processortaken; timingsgaranties moeten behouden worden voor pakkettransmissies over communicatielinks. Het analyseren van berichtenvertraging in zulke netwerken is uitdagend, vanwege het feit dat berichtenstromen op elkaar inwerken over meerdere links, traffic-shaping mechanismes reguleren hoe pakketten vrijgegeven en verzonden worden, en dat de interne structuur van netwerkswitches beïnvloedt hoe pakketten in de wachtrij worden geplaatst en verzonden worden. Deze factoren introduceren complexe afhankelijkheden die end-to-end communicatievertragingen beïnvloeden. Om die reden onderzoekt dit proefschrift hoe communicatie timingseigenschappen binnen TSN netwerken op systematische wijze geanalyseerd kunnen worden. Deze methodes bieden analysetechnieken voor het beredeneren over hoe verschillende traffic-shaping mechanismes berichtenvertraging beïnvloeden, en ondersteunen de analyse van zowel individuele netwerkswitches, als end-to-end vertraging over hele communicatiepaden.

De derde contributie is gerelateerd aan de schaalbaarheid van timingsanalyses voor steeds complexer wordende real-time systems. Naarmate het aantal taken, bronnen van timingsvariabiliteit, en wisselwerkingen tussen systeemcomponenten groeit, kan het aantal scenario's dat moet worden overwogen significant toenemen. Dit maakt het uitdagend om zowel grote systemen te analyseren, evenals volledige dekking van relevante planningsmogelijkheden te behouden. Daarom onderzoekt dit proefschrift technieken die schaalbare timingsanalyse mogelijk maken, waarbij correctheid behouden blijft. Deze technieken verminderen het aantal uitvoeringsscenario's die meegenomen moeten worden, maar waarborgen toch een complete dekking van relevante planningsgedragen.

Om bovenstaande analyses mogelijk te maken, bouwt dit proefschrift voort op het Schedule Abstraction Graph (SAG) framework, geïntroduceerd in [46, 48], dat een abstractie-gebaseerde aanpak biedt voor het redeneren over planningsgedrag. Het kernidee van SAG is het representeren van de mogelijke systeemplanningen als een grafiek, die relevante uitvoeringsscenario's vastlegt, zonder het individueel analyseren van ieder scenario. In plaats van dat elk concreet uitvoeringsscenario concreet wordt gerepresenteerd, groepeert het framework scenario's met gelijk planningsgedrag in abstracte staten, en verkent het hun overgangen. Dit geeft de mogelijkheid tot systematische analyse van timingsgedrag over een grote verscheidenheid aan uitvoeringsscenario's.

Gebruikmakend van deze abstractie-gebaseerde aanpak, breidt dit proefschrift de analyse uit naar taakmodellen met precedentierelaties en pauzeringsintervallen, stemt het dit framework af op het analyseren van berichtentransmissies in Time-Sensitive Networking, en ontwikkelt het technieken die de schaalbaarheid van de analyse bij grotere systemen verbeteren. Samengevat, verbeteren bovengenoemde contributies het vermogen om tijdsgedrag van realistische real-time systems te analyseren binnen verschillende uitdagende omgevingen.

TABLE OF CONTENTS

List of Figures **xvii**

CHAPTER 1: Introduction **1**

1.1 Research Objectives, Contributions, and Questions 4

1.2 Thesis Structure 6

PART I: Schedule Abstraction Graph and Extensions to Heterogeneous Workloads

CHAPTER 2: The Schedule Abstraction Graph **11**

2.1 System Model 12

2.2 The Analysis Problem 15

2.3 Existing Analysis Approaches 17

2.4 Schedule Abstraction Graph in the State of the Art 20

2.5 Intuition Behind the Schedule Abstraction Graph 20

2.6 Formal Definition of the Schedule Abstraction Graph 27

2.7 Scope of the Schedule Abstraction Graph in this Thesis 31

CHAPTER 3: Analysis of Event-Driven Delay-induced (EDD) Tasks **33**

3.1 System Model 37

3.2 Analysis 42

3.3 Expansion Phase 46

3.4 Merging Phase 55

3.5 Correctness 57

3.6 Evaluation 58
3.7 Conclusion 62

PART II: Extending the Schedule Abstraction Graph to Time-Sensitive Networking

CHAPTER 4: Time-Sensitive Networking (TSN) **67**
4.1 Overview of Time-Sensitive Networking 69
4.2 Structure of a TSN Switch 70
4.3 Challenges of integrating TSN into the SAG model 73
4.4 System Model of TSN 74
4.5 Traffic Shaping Mechanisms 77

CHAPTER 5: Analysis of Time-Aware Shapers on a TSN switch **83**
5.1 System Model 84
5.2 Latency Analysis Algorithm for TAS 88
5.3 Possibly Eligible Packets and Eligible Transmission Intervals 92
5.4 Expansion Phase 103
5.5 Merge Phase 104
5.6 Proof of Correctness 105
5.7 Evaluation 106
5.8 Conclusion 110

CHAPTER 6: Analysis of Credit-Based Shapers (CBS) **113**
6.1 System Model 115
6.2 Analysis Methodology 118
6.3 Eligible Packets 122
6.4 Generation of New States 129
6.5 Merging of States 133
6.6 Proof Of Correctness 134
6.7 Improving the Analysis Scalability 136
6.8 Experimental Results 138
6.9 Conclusion 142

PART III: Improving Scalability and Structural Efficiency of the Schedule Abstraction Graph

CHAPTER 7: Improving the Scalability and Structural Efficiency of the SAG **147**

7.1 Structural Limitations of State-Centric SAG — 149

7.2 Nodes as Structural Abstractions — 150

7.3 Case Study I: Event-Driven Delay-Induced (EDD) Tasks — 151

7.4 Case Study II: Time-Aware Shapers in Time-Sensitive Networking — 155

7.5 Evaluation — 157

7.6 Discussion and Conclusion — 167

CHAPTER 8: Conclusion and Future Work **171**

8.1 Answering the Research Questions — 172

8.2 Key Insights — 174

8.3 Final Remarks — 175

List of Abbreviations **177**

List of Symbols **179**

Bibliography **190**

IPA Dissertation Series **197**

Acknowledgements **201**

List of Publications **207**

About the Author **209**

LIST OF FIGURES

2.1 Example task set consisting of two periodic tasks, τ_1 and τ_2 and the jobs generated by them within a 10 time unit observation window — 16
2.2 All feasible schedules arising from the possible arrival-time and execution-time combinations — 17
2.3 Concrete states corresponding to every time instance of a schedule — 22
2.4 Reduction of a schedule to its relevant concrete states — 23
2.5 Reduction of multiple schedules to its abstract states. — 24
2.6 Schedule Abstraction Graph for jobset in Figure 2.1 — 26
2.7 Schedule Abstraction Graph for jobset in Figure 2.1 with merging technique applied — 27

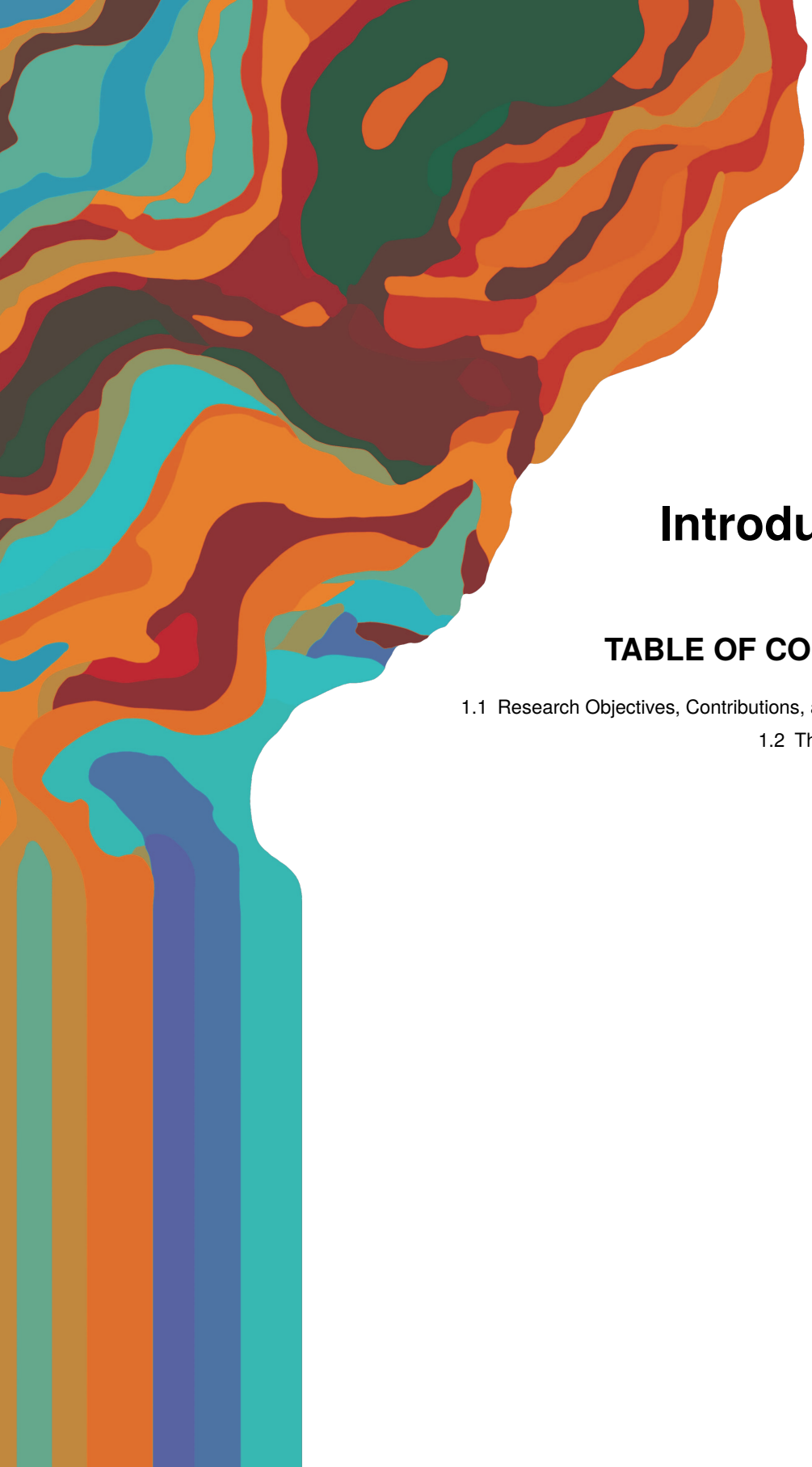
3.1 An example of an EDD task. Numbers on the edges model the minimum and maximum suspension time between nodes — 39
3.2 An example of generation of a schedule abstraction graph for EDD tasks — 44
3.3 Deriving the availability intervals of the cores in $\tilde{s}$. — 53
3.4 Experimental results — 60

4.1 Switch architecture with m ingress ports, n egress ports, and o queues per egress port (one per traffic class). — 72
4.2 Flow set and the corresponding set of packets released during the first 10 time units — 78
4.3 Example illustrating TAS scheduling — 79
4.4 Example illustrating CBS scheduling — 81

5.1 Example packet set. — 90
5.2 SAG built using the packet set in Figure. 5.1 — 90
5.3 Example that shows how eligible transmission intervals are calculated for packet p_3 — 101
5.4 Example that shows how overlapping gates are configured at 50% overlap — 108
5.5 Packets' latency comparison for different gate overlap scenarios. 109

6.1 State Transition Diagram for Example 6.2 — 118
6.2 Example showing the calculation of ι_{C_a} — 125
6.3 Example showing how g_{C_2} projects the coordinates in ϵ_{C_b} from Example 6.2 to the possible start times of $p_{2,2}$ (blue polygon projected to the yellow region), and how h_{C_2} projects the output of g_{C_2} to the possible finish times of $p_{2,2}$ (Yellow region projected to the pink polygon) — 131
6.4 Over-approximating the set of credit–time coordinates into a compact parallelogram representation — 137
6.5 Network architecture with a single switch — 139
6.6 Network architecture with two switches — 139
6.7 Experimental Results — 140

7.1 Two EDD states with the same scheduling history but different timing information are grouped into a single node. — 153
7.2 Two TAS states with the same transmission history but different egress-port availability intervals are grouped into a single node. 156
7.3 Runtime as a function of utilisation for self-suspending task systems with 2 (top left), 4 (top right), and 8 (bottom) processors. 160
7.4 Runtime as a function of the number of jobs for systems with 4 (left) and 8 (right) tasks. — 161
7.5 Runtime as a function of explored states (v1) and nodes (v2, v3) for systems with 4 (top left), 8 (top right), and 12 (bottom) tasks. 162
7.6 Runtime as a function of explored entities for DAG-based task systems on 4 processors with 2 (left) and 4 (right) tasks. — 164
7.7 Runtime as a function of explored entities for DAG-based task systems on 6 processors with 4 (left) and 6 (right) tasks. — 164
7.8 Ratio of states to nodes across different gate overlap variants and timing configurations. — 166
7.9 Speedup of v2 and v3 relative to v1 as a function of the number of states. — 166



1

Introduction

TABLE OF CONTENTS

1.1 Research Objectives, Contributions, and Questions

1.2 Thesis Structure

When considering systems and applications, the first concern is usually whether they work correctly. If a system is designed for a particular purpose, it is expected to achieve exactly that. This idea is often referred to as functional correctness. In many cases, this is sufficient. As long as the correct result is produced, the system is considered to be 'working properly'. However, there are many systems for which functional correctness alone is not enough.

Let us consider a simple example from the automotive domain. Modern cars contain sensors that can detect when a crash is about to happen. When such a situation is detected, the sensor sends a signal to the airbag system, instructing it to inflate. Assuming that the sensor detects the crash correctly, the message is sent from the sensor to the actuator, and the airbag eventually inflates, everything works correctly from a purely functional point of view. However, if the airbag inflates several seconds before or after the crash, the outcome is clearly unacceptable.

This example highlights an important fact, i.e., in some systems, *when* an action happens is just as important as whether it happens at all. In these systems, producing the correct result too late or too early can be just as harmful as producing the incorrect result. Systems with this property are known as real-time systems.

In real-time systems, the correctness has two parts. First, the system must perform the expected action correctly. Second, it must perform that action within clearly defined time bounds. These time bounds are often expressed as deadlines. If a deadline is not met, the system is considered to have behaved incorrectly, even if the underlying computation was correct.

Real-time systems appear in a wide variety of forms. Some are safety-critical, including airbag controllers, aircraft navigation equipment, and autonomous driving systems. In these systems, failing to meet timing requirements can have devastating consequences, such as threats to human life, damage to property, and environmental or health impacts. Other real-time systems, while not safety-critical, may still place great importance on time bounds. For example, in an industrial printing system, actions such as loading paper, applying ink, and printing must occur in the correct order and at the correct times. If either of the two are not respected, large batches of printed material may be wasted, leading to significant financial loss. These systems are often referred to as business-critical real-time systems.

A common misconception when reasoning about time-related system behaviour is that the solution is simply to make everything run as fast as possible. However, speed alone does not address the core problem of guaranteeing predictable timing behaviour. A system that always responds within known and guaranteed time limits is far more useful than one that is usually fast but unpredictable.

Real-time requirements are not limited to individual components or applications. They also appear in communication between components within a larger system. As systems become more complex and distributed, the timing of messages exchanged between components becomes critical. This has led to the development of networking technologies such as Time-Sensitive Networking (TSN), that provide mechanisms that guarantee predictable communication delays.

Failure to respect timing requirements in real-time systems can lead to unsafe or otherwise undesirable outcomes. Consequently, it is necessary to establish that these timing constraints are always satisfied. The key challenge is therefore determining, before deployment, whether a system will consistently behave correctly within its required time bounds.

Real-world systems are often far less predictable than desired. The same system can behave differently depending on its environment, its input, or the particular situation it is in. Guaranteeing that a system will behave safely and correctly in all possible situations is inherently challenging.

This difficulty becomes more pronounced as real-time systems grow in size and complexity. Modern systems rarely consist of a single component that operates in isolation. Instead, they are made up of many interacting components that run concurrently, share resources, and exchange information. The timing behaviour of one component can influence and be influenced by the behaviour of many others.

As a result, the number of possible system states and execution scenarios increases rapidly. Small variations in timing, workload, or communication delays can lead to very different system behaviours. This makes it increasingly difficult to reason about all possible situations without systematic methods of analysis. Hence, guaranteeing that timing constraints are satisfied under all relevant conditions requires systematic analysis techniques.

To address this challenge, systematic timing analysis techniques are required to determine whether real-time systems satisfy their timing requirements. This thesis investigates methods for analysing the timing behaviour of real-time systems and determining whether their timing constraints are satisfied under all relevant scenarios. The work presented in this thesis considers two key aspects of real-time systems. The first relates to real-time task scheduling, in which multiple jobs are executed on processing resources. The second focuses on communication behaviour, where messages are transmitted through time-sensitive networks.

For both aspects of systems, the objective is to determine whether timing bounds are respected in all possible scenarios and to report relevant timing properties of its applications, such as their best-case and worst-case response time. Several approaches have been proposed to perform such analyses. Traditional

real-time analysis techniques typically rely on analytic bounds that estimate worst-case behaviour without exploring all possible system executions but assuming a worst-case scenario. While these methods are computationally efficient, they often rely on conservative assumptions and may therefore produce overly pessimistic results. At the other end of the spectrum, exhaustive approaches such as model checking explore the system behaviour in detail and can provide accurate results, but they typically suffer from scalability limitations as system complexity increases. As a result, existing techniques face a fundamental trade-off between scalability and analytical accuracy, making it difficult to analyse realistic real-time systems accurately and efficiently.

To address this challenge, this thesis investigates abstraction-based methods for analysing the timing behaviour of realistic real-time systems. The aim is to develop analyses that remain accurate enough to capture relevant system behaviour while scaling better than naive exhaustive exploration. To realise this, the thesis builds on the Schedule Abstraction Graph (SAG) framework [46, 48], which provides a structured way to represent and explore sets of scheduling scenarios without the need of analysing every concrete execution scenario independently.

1.1 Research Objectives, Contributions, and Questions

This thesis addresses the challenge of analysing the timing behaviour of complex real-time systems. In particular, it considers systems whose behaviour is difficult to analyse using traditional methods because of expressive task structures, suspension behaviour, communication-level timing dependencies, and the rapid growth in the number of relevant execution scenarios.

The objective of the thesis is to develop analysis methods that can reason systematically about such systems while maintaining a practical balance between accuracy and scalability. More specifically, the thesis makes three main contributions.

First, it develops timing analysis methods for more expressive application-level real-time task models, including tasks with precedence constraints and suspension behaviour.

Second, it adapts the analysis to the domain of Time-Sensitive Networking, where the entities of interest are packet transmissions and the objective is to reason about latency in communication networks.

Third, it investigates techniques for improving the scalability of the analysis as systems become larger and timing variability increases. These objectives are formalised through the following research questions.

RQ1: *To what extent can the timing behaviour of real-time systems be analysed accurately and in a scalable way across both the communication and computation domains?*

This question is addressed by developing and evaluating abstraction-based analyses for increasingly complex system models in both communication and computation domains. The thesis first investigates the accuracy of the proposed analyses by comparing their results against the state of the art. It then develops techniques to improve scalability while maintaining the same level of accuracy. In this way, the thesis examines how the control of represented state information and exploration structure affects both analytical accuracy and computational feasibility.

RQ2: *How can the response time of limited-pre-emptive real-time tasks with segment-level precedence constraints following a DAG structure and with self-suspension behaviour be analysed?*

This thesis considers a rich task model that is relevant for modern real-time systems, where tasks may consist of multiple execution segments with non-linear, DAG-structured precedence constraints and may also exhibit self-suspension behaviour. Such self-suspensions can arise, for example, when part of a task is offloaded for execution elsewhere, or when a task must wait for a network communication or data transfer to complete before resuming. Capturing these behaviours faithfully is important for representing the timing behaviour of complex limited-pre-emptive real-time tasks. The thesis focuses on the question of how such systems can be represented and analysed. In particular, it presents the first methodology for analysing limited-pre-emptive tasks that combine DAG-structured precedence constraints and self-suspensions within a unified model, thereby enabling a systematic treatment of these rich behavioural characteristics.

RQ3: *How can an abstraction-based timing analysis, in particular the Schedule Abstraction Graph, be generalised to the domain of Time-Sensitive Networking?*

RQ3.1: Which principles of Schedule Abstraction Graph-based timing analysis can be reused, with suitable modelling changes, when moving from analysing computation on processors to communication on TSN networks?

RQ3.2: Which aspects of Time-Sensitive Networking require the development of fundamentally new modelling and analysis techniques within the Schedule Abstraction Graph framework?

This question addresses how abstraction-based timing analysis, in particular the Schedule Abstraction Graph, can be generalised from application-level

computation on processors to Time-Sensitive Networking (TSN). Because the framework was originally designed for the analysis of jobs executing on processors, its application to packet-based communication requires more than a direct transfer to a new domain. The thesis therefore investigates which modelling principles can be retained and reapplied, with suitable modifications, and which aspects of TSN demand entirely new modelling and analysis techniques. In particular, it examines the similarities between computation and communication that allow parts of the original framework to be reused, while also developing new abstractions for network-specific behaviour such as packet interactions, queueing effects, and traffic shaping across links and switches. Special attention is given to TSN shaping mechanisms such as TAS and CBS, whose analysis had not previously been addressed within the Schedule Abstraction Graph framework. The thesis thereby shows how the framework can be extended to support systematic latency analysis in TSN systems.

1.2 Thesis Structure

Following the introduction, the thesis is divided into three main parts, each addressing a specific aspect of real-time system analysis using Schedule Abstraction Graphs.

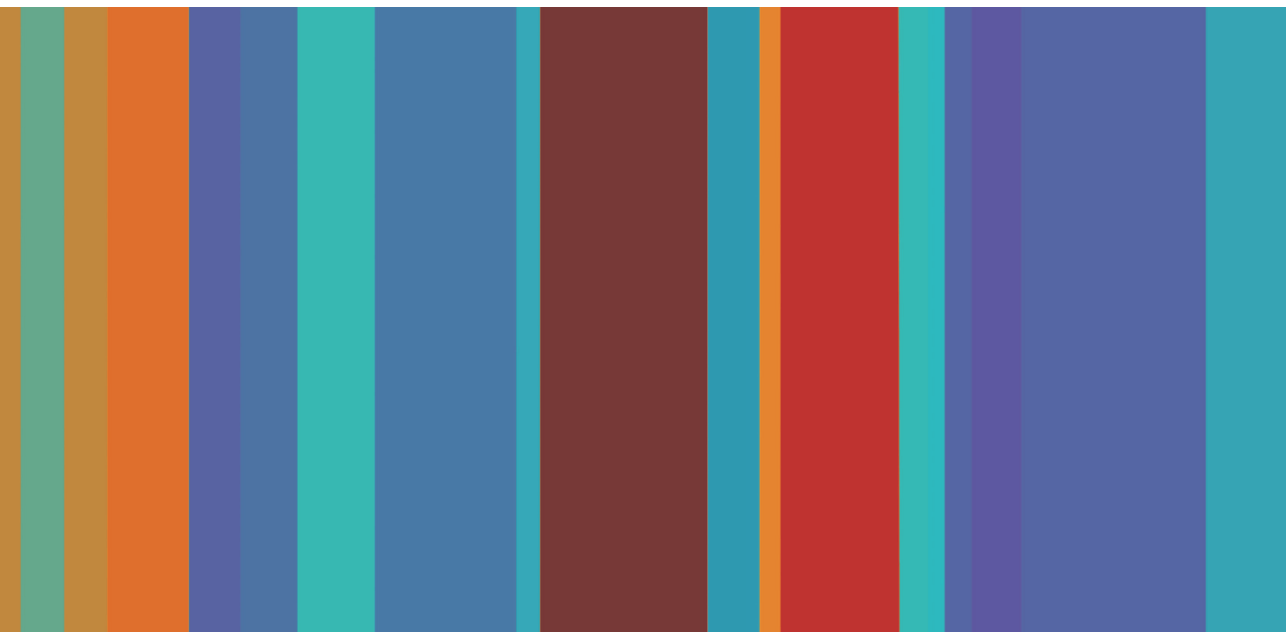
Part I, *Schedule Abstraction Graphs in Real-Time Task Scheduling*, focuses on computation-level behaviour, where jobs execute on processing resources. This part begins with Chapter 2, which introduces the background and fundamental concepts needed for the analysis framework used throughout the thesis and provides an understanding of Schedule Abstraction Graphs, followed by Chapter 3, which develops the analysis for more expressive task models.

Part II, *Schedule Abstraction Graphs in Time-Sensitive Networking*, shifts the focus to real-time communication. It starts with background on time-sensitive networking in Chapter 4, followed by analyses of two traffic-shaping mechanisms in Chapters 5 and 6.

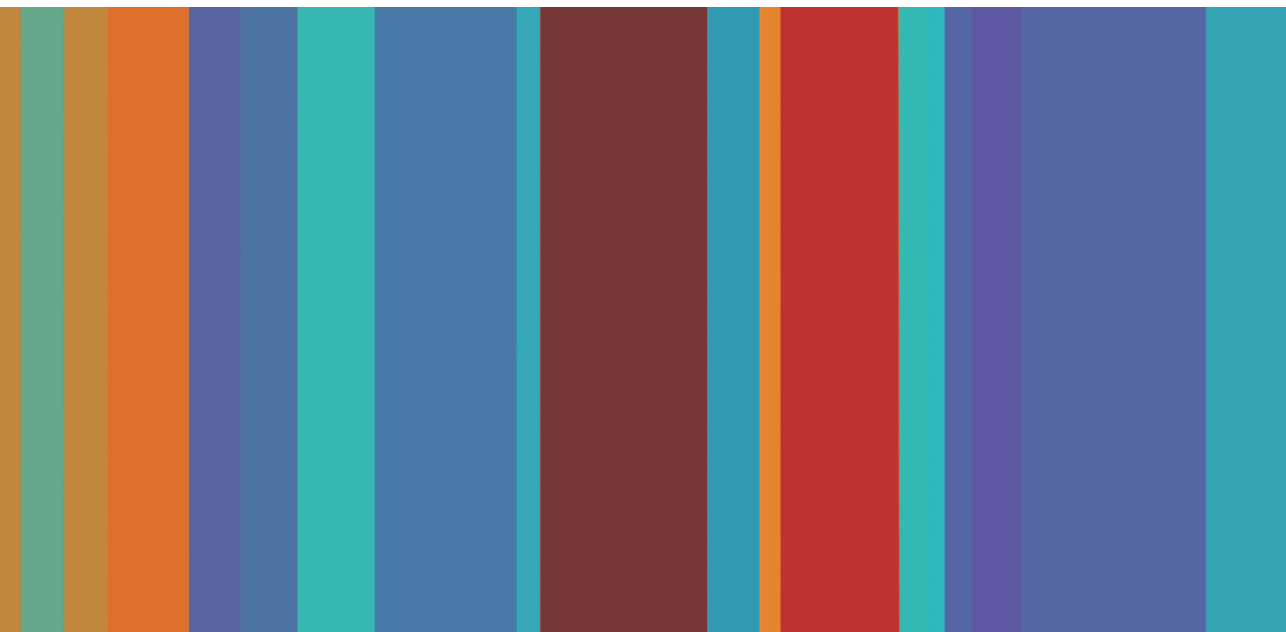
Part III, *Improving the Scalability of Schedule Abstraction Graphs*, addresses scalability challenges that arise across both computation and communication domains. This part presents techniques for improving the scalability of the underlying analysis framework and discusses how these improvements apply across the considered settings.

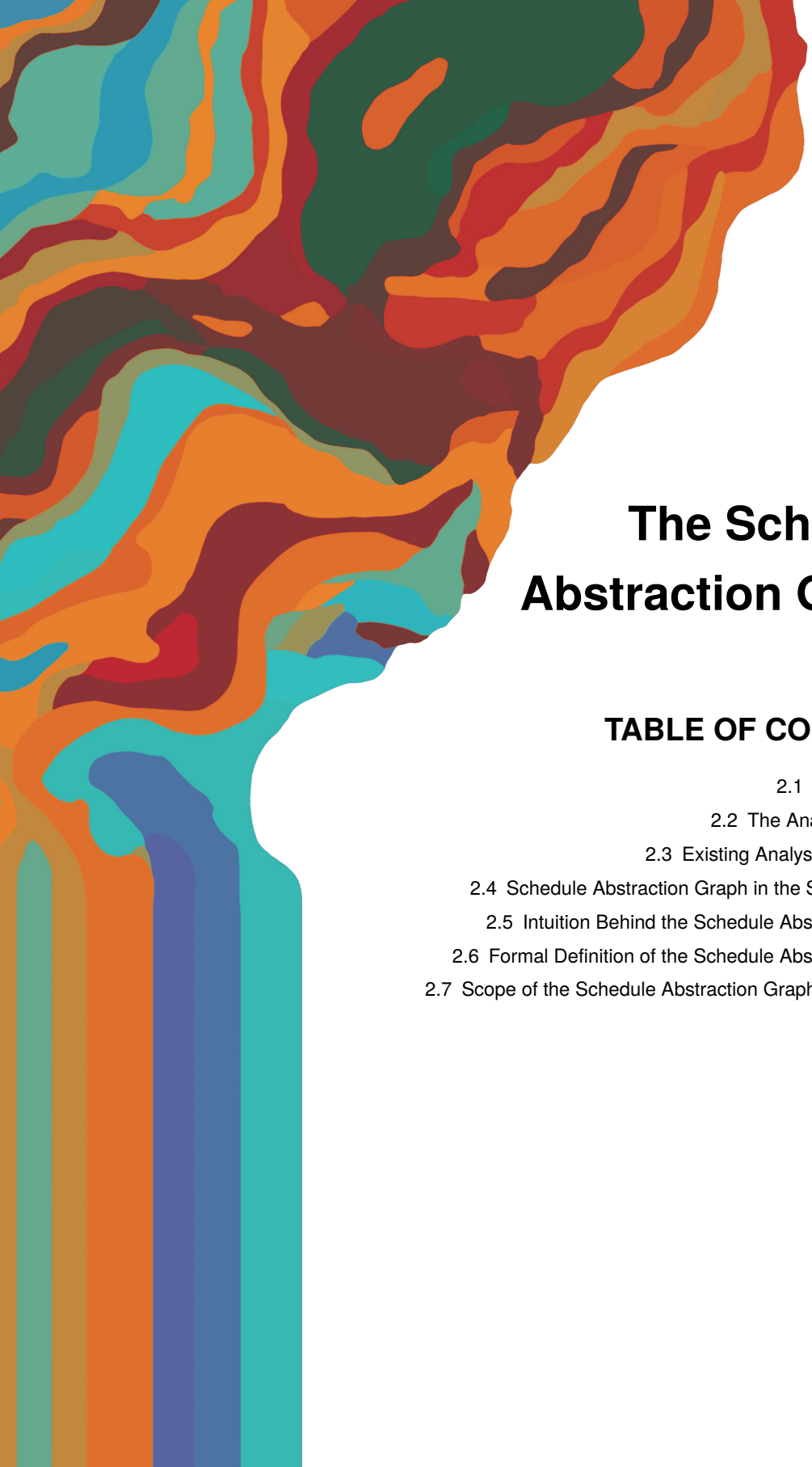
The thesis concludes with Chapter 8, which summarises the main contributions and outlines directions for future work.

PART I



Schedule Abstraction Graph and Extensions to Heterogeneous Workloads





2

The Schedule Abstraction Graph

TABLE OF CONTENTS

2.1 System Model

2.2 The Analysis Problem

2.3 Existing Analysis Approaches

2.4 Schedule Abstraction Graph in the State of the Art

2.5 Intuition Behind the Schedule Abstraction Graph

2.6 Formal Definition of the Schedule Abstraction Graph

2.7 Scope of the Schedule Abstraction Graph in this Thesis

Chapter 1 motivated the need for schedulability analyses for real-time systems. This chapter provides the formal system model and the background required to understand the contributions of this thesis. We begin by defining the real-time system model used throughout the thesis, including tasks, timing parameters, and assumptions on the scheduling policies used by the system. These definitions establish a precise foundation for the analysis problems considered later.

Next, we review common schedulability analysis approaches from the literature, with particular emphasis on the trade-off between scalability and accuracy and on the limitations that motivate the use of reachability-based abstractions. Their key limitations are also discussed. Finally, we introduce the Schedule Abstraction Graph (SAG) framework [46, 48], which forms the basis of the methods developed in the subsequent chapters.

2.1 System Model

To reason precisely about schedulability, a formal description of the system under analysis is required. This section introduces the system model used throughout this chapter, including the task and job models, the platform assumptions, and the scheduling policies considered. Together, these definitions establish the foundation for the analysis framework introduced later.

2.1.1 Task Model

A real-time system performs a number of activities over time. Each activity represents some piece of work that the system must carry out. These activities may occur once, repeatedly, or sporadically. In this thesis, the focus is on activities that occur in a regular, repeating manner.

These repeating activities are modelled as tasks. For example, a sensor may need to check for new data at regular intervals, or one component may need to communicate with another component periodically.

The system is described as a finite collection of tasks,

$$\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}.$$

Each task is defined by a collection of timing-related attributes specifying how regularly it repeats, when its first execution may occur, how much variation there can be in its arrivals, for how long it can execute, and what its deadline is. These attributes represent the minimum information required to characterise a task. In some settings, a task may additionally be assigned a priority parameter.

Accordingly, a task τ_i is defined by the tuple

$$\tau_i = \left(T_{\tau_i},\ O_{\tau_i},\ J_{\tau_i},\ C_{\tau_i}^{\min},\ C_{\tau_i}^{\max},\ D_{\tau_i},\ \pi_{\tau_i}\right)$$

where:

- $T_{\tau_i} \in \mathbb{R}_{>0}$ denotes the period of the task.
- $O_{\tau_i} \in \mathbb{R}_{\geq 0}$ denotes the offset of the task, specifying the earliest time at which the task becomes active.
- $J_{\tau_i} \in \mathbb{R}_{\geq 0}$ denotes the arrival jitter.
- $C_{\tau_i}^{\min}, C_{\tau_i}^{\max} \in \mathbb{R}_{>0}$ denote the minimum and maximum execution time requirements.
- $D_{\tau_i} \in \mathbb{R}_{>0}$ denotes the relative deadline of the task.
- $\pi_{\tau_i} \in \mathbb{N}_{\geq 0}$, when defined, denotes a task-level priority parameter used by the scheduling policy to determine execution order.

2.1.2 Job Model

A task describes a recurring activity. Each individual occurrence of a task gives rise to a concrete unit of work, called a job.

If a task repeats periodically, it generates a potentially infinite sequence of jobs over time where every two jobs arrival is separated by the task period.

The collection of jobs generated by all tasks is denoted by $\mathcal{J}$. Each job $j \in \mathcal{J}$ is associated with exactly one task.

For a task $\tau_i \in \mathcal{T}$, the k-th job generated by τ_i, for $k \geq 0$, is denoted $j_{i,k}$ and has a concrete arrival time $a_{j_{i,k}}$ such that

$$a_{j_{i,k}}^{\min} \leq a_{j_{i,k}} \leq a_{j_{i,k}}^{\max},$$

where $a_{j_{i,k}}^{\min}$ and $a_{j_{i,k}}^{\max}$ are bounds on the job $j_{i,k}$'s arrival time that can be derived from the properties of task τ_i as follows

$$a_{j_{i,k}}^{\min} = O_{\tau_i} + k \cdot T_{\tau_i},$$

$$a_{j_{i,k}}^{\max} = O_{\tau_i} + k \cdot T_{\tau_i} + J_{\tau_i}.$$

Similarly, each job has a concrete execution time $C_{j_{i,k}}$ bounded by

$$C_{j_{i,k}}^{\min} \leq C_{j_{i,k}} \leq C_{j_{i,k}}^{\max},$$

where

$$C_{j_{i,k}}^{\min} = C_{\tau_i}^{\min}, \quad C_{j_{i,k}}^{\max} = C_{\tau_i}^{\max}.$$

Each job has an absolute deadline defined by

$$d_{j_{i,k}} = O_{\tau_i} + k \cdot T_{\tau_i} + D_{\tau_i}.$$

2.1.3 Platform Model

Jobs are executed on processors. A platform may contain one or more processors, and the characteristics of these processors determine the scheduling model under consideration.

The simplest platform model is the *single-core* or *uniprocessor* system, in which there is only one processor. At any given time, this processor can execute at most one job, and each job can execute on at most one processor at a time. Single-core systems are useful for introducing the basic scheduling concepts and analysis techniques in a clean and intuitive way. However, many modern computing systems must handle increasing performance and timing demands that are difficult to satisfy with a single core alone.

For this reason, modern systems are often based on *multicore* or multiprocessor platforms, in which several processing cores are available. Such platforms are now widely used in embedded, communication, and general-purpose computing systems. Broadly, multicore platforms can be divided into two categories.

Homogeneous multicore platforms. In a homogeneous platform, all processors are identical in their computational capabilities. Any job can in principle execute on any processor with the same execution characteristics. This model is widely used because it simplifies both scheduling and analysis.

Heterogeneous multicore platforms. In a heterogeneous platform, processors differ in their capabilities, speed, architecture, or specialised functionality. As a result, the execution behaviour of a job may depend on the processor on which it is scheduled. Such platforms are increasingly common in modern systems, for example in big.LITTLE architectures that combine high-performance cores with energy-efficient cores.

On multicore platforms, it is also necessary to distinguish between different scheduling approaches.

Partitioned scheduling. Under partitioned scheduling, each task is assigned to a specific processor, and all jobs generated by that task execute only on that

processor. After this assignment has been made, each processor can be analysed largely as an independent uniprocessor system. This approach simplifies runtime scheduling, but the partitioning itself may be difficult to determine efficiently.

Global scheduling. Under global scheduling, jobs are allowed to execute on any processor in the platform, and they may migrate from one processor to another during execution, depending on the scheduler. This approach provides greater flexibility and can improve processor utilisation, but it also makes scheduling and timing analysis more complex.

2.1.4 Scheduling Policies

When multiple jobs are ready to execute simultaneously, a scheduling decision must be made. This decision is determined by a scheduling policy, which specifies how jobs are selected for execution.

Scheduling policies may assign priorities at the task level or at the job level. In task-level fixed-priority (Task Level Fixed Priority (TLFP)) scheduling, all jobs of a task share the same priority. In job-level fixed-priority (Job Level Fixed Priority (JLFP)) scheduling, priorities may differ across jobs. Scheduling policies may also be dynamic, where priorities change over time, such as in earliest-deadline-first scheduling.

Scheduling policies can further be characterised by additional properties. A policy is called *work-conserving* if it never leaves a processor idle while there is a ready job that could execute on it. In other words, available processing capacity is always used whenever possible. A policy is called *deterministic* if the scheduler always makes the same decision whenever the system is in the same state. Thus, under a deterministic policy, a given system state uniquely determines the next scheduling decision.

A schedule describes the outcome of applying a scheduling policy on a concrete set of jobs. It specifies, for every point in time, which jobs are executing and how many processors are kept idle.

Scheduling decisions are made when the system state changes, such as when a job arrives or completes. At each such point, the scheduler selects one of the ready jobs according to the scheduling policy to execute on the processor.

2.2 The Analysis Problem

At this point, the system model is in place: tasks are defined, jobs are generated from those tasks, and a scheduling policy determines how the jobs are executed

on the processor. The next step is to make the analysis problem concrete and to see why it becomes difficult very quickly.

Consider the example in Figure 2.1, which illustrates two tasks, τ_1 and τ_2, together with their parameters. For simplicity, all values in this example are assumed to be integers rather than real numbers. Over a time span of 10 time units, these two tasks generate three jobs: one job from τ_1 and two jobs from τ_2.

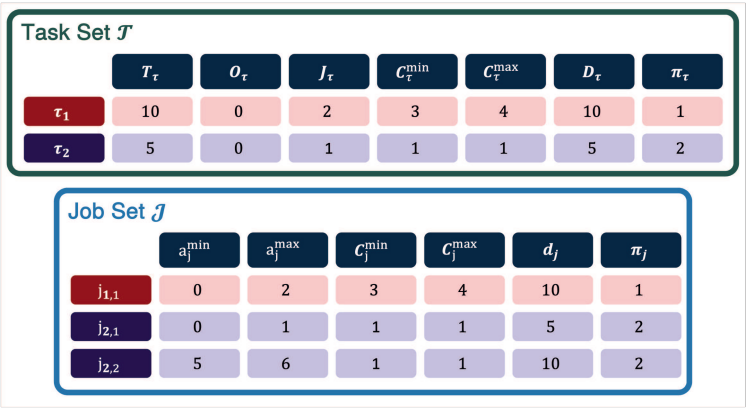


Figure 2.1: Example task set consisting of two periodic tasks, τ_1 and τ_2 and the jobs generated by them within a 10 time unit observation window

Even for this small system, many different schedules are possible. The reason is that job releases may vary within their arrival bounds, and execution times may vary within their execution bounds. Each combination of arrival times and execution times defines an *execution scenario*. For each job $j \in \mathcal{J}$, define

$$\gamma_j = (a_j, C_j),$$

where a_j denotes the arrival time of j and C_j denotes its execution time. An execution scenario for the entire job set $\mathcal{J}$ is then given by the collection of such assignments for all jobs,

$$\Gamma = \{\gamma_j \mid j \in \mathcal{J}\}.$$

The set Γ therefore represents all possible execution scenarios obtained from different combinations of arrival times and execution times of the jobs in $\mathcal{J}$.

Once the arrival and execution times of every job are fixed in this way, and assuming a deterministic scheduling policy, the resulting execution of the system is uniquely determined. Consequently, each execution scenario corresponds to exactly one concrete schedule.

The figure therefore illustrates all schedules that arise from all valid arrival-time and execution-time combinations within the 10-unit observation window.

Already in this simple setting with two tasks, three jobs, and a short time

window, the number of distinct feasible schedules can become large: in this example there are 24 possible schedules as shown in Figure 2.2.

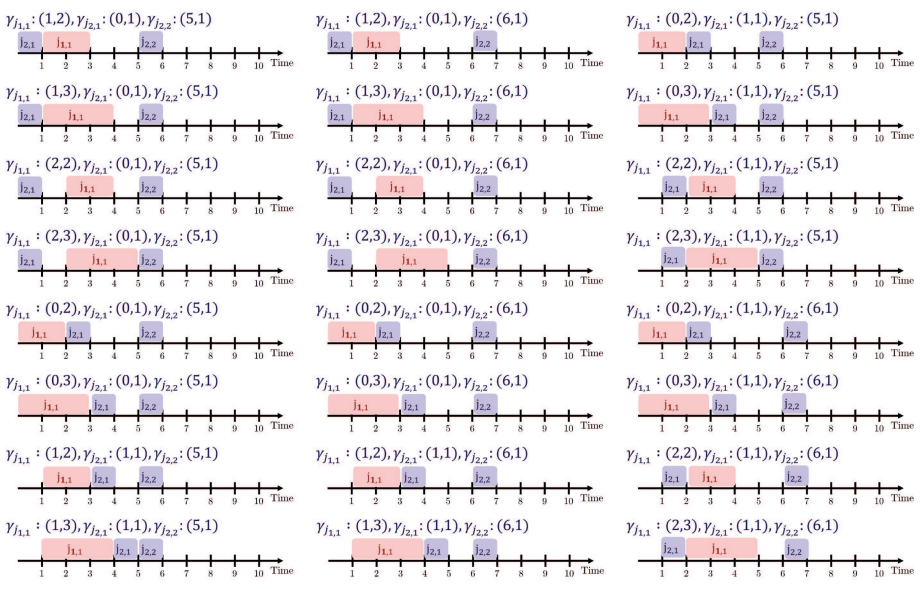


Figure 2.2: All feasible schedules arising from the possible arrival-time and execution-time combinations

The difficulty of this problem arises from the combinatorial growth in the number of possible schedules. Even for a fixed set of jobs, variations in execution order and processor availability may lead to distinct timing behaviours. As the number of tasks, jobs, and processors increases, the number of feasible scheduling scenarios grows rapidly.

For realistic systems, explicitly enumerating and analysing all such schedules becomes computationally infeasible. This observation motivates the need for structured analysis techniques capable of reasoning about large classes of schedules without exploring each one individually.

2.3 Existing Analysis Approaches

It has been seen earlier in this chapter why the scheduling problem is not an easy one, and in fact not something that can be trivially computed. This motivates the need for analysis techniques that can reason about all these scheduling scenarios.

Over the years, several approaches have been developed to address this problem. The state of the art can be divided into two broad categories, and the intuition behind how each category functions is discussed here.

The first category derives analytical bounds on the timing behaviour without explicitly exploring all possible schedules. This category uses (a set of) equations that model a worst-case scenario. Any solution to that set of equations returns bounds on the response time of each task. This class of analyses is usually referred to as *Response Time Analysis (RTA)* [5, 20]. On the other side, there are solutions that construct and explore the space of all possible schedules explicitly by reasoning over reachable system configurations [31, 65, 61]. These two categories represent the two ends of a spectrum with respect to scalability and accuracy.

2.3.1 Fixed-Point Iteration Based Analysis

One common class of schedulability analysis techniques derives timing bounds through analytical recurrence relations. These approaches are typically referred to as *fixed-point-iteration-based analysis*.

In such methods, the worst-case behaviour of a task is expressed as a recurrence equation that captures the interference it may experience from other tasks. Starting from an initial estimate, the equation is repeatedly evaluated until the value converges to a stable solution, known as a fixed point. This fixed point represents a safe upper bound on the response time of the task.

For example, classical response-time analysis for fixed-priority scheduling [5, 20, 7, 38, 63, 27, 28], the worst-case response time of a task is computed by iteratively evaluating an equation that accounts for the workload generated by higher-priority tasks during the task's execution window.

The primary advantage of fixed-point-iteration-based methods is their scalability. Since these techniques reason about aggregated interference rather than individual scheduling scenarios, they can analyse systems with a large number of tasks efficiently.

However, this efficiency comes at a cost. The analysis relies on conservative approximations of interference and release patterns. As task models become more expressive, or as additional system constraints are introduced, these approximations may become increasingly pessimistic. In complex systems, this pessimism may lead to declaring a system unschedulable even when a valid schedule exists. Furthermore, deriving safe recurrence relations for more complex task models may itself become difficult.

2.3.2 Reachability-Based Analysis

An alternative class of techniques analyses schedulability by explicitly exploring how the system evolves over time. These approaches are commonly referred to

as *reachability-based analysis*.

In reachability-based methods [31, 65, 61, 6, 15, 14], the system is represented as a collection of states, where each state captures the relevant information about the current execution of the system. Such information typically includes which jobs have already arrived, which jobs have completed, the remaining execution time of currently active jobs, and the current allocation of processors.

Transitions between states correspond to events such as job arrivals, job completions, or scheduling decisions. By systematically exploring all reachable states, it becomes possible to determine whether any execution leads to a missed deadline.

The primary advantage of reachability-based methods is their accuracy. Since all reachable behaviours are explored explicitly, the analysis captures the exact scheduling behaviour within the limits of the system model. The Schedule Abstraction Graph (SAG) [46, 48, 49] belongs to this class of techniques.

The main limitation is scalability. The number of reachable states grows rapidly with the number of tasks, processors, and timing parameters. As a result, the state space can become extremely large even for relatively small systems. This phenomenon, commonly referred to as *state-space explosion*, significantly restricts the applicability of naive reachability-based approaches.

2.3.3 The Trade-Off

Schedulability analysis techniques therefore exhibit a fundamental trade-off between scalability and accuracy. Fixed-point iteration based methods scale well because they reason about aggregated interference rather than individual schedules. However, this often requires conservative assumptions, which may lead to pessimistic results. Reachability-based methods, on the other hand, analyse the system by explicitly exploring its possible execution states. These techniques provide accurate results but often suffer from state-space explosion.

The Schedule Abstraction Graph (SAG) [46, 48, 49] belongs to the class of reachability-based analysis techniques. However, unlike traditional exhaustive exploration approaches, SAG specializes the state-space exploration engine for the timing analysis of real-time systems and introduces several abstraction mechanisms that significantly reduce the number of explored states. By removing irrelevant states, aggregating similar states, and merging compatible abstract states, SAG attempts to retain the accuracy of reachability-based analysis while improving scalability.

2.4 Schedule Abstraction Graph in the State of the Art

The Schedule Abstraction Graph (SAG) was first introduced by [46] as an exact reachability-based response-time analysis technique for non-pre-emptive scheduling. Since then, SAG has developed into a broader analysis framework that has been applied to a variety of task models and system settings.

While the central idea of SAG remains the same, namely to represent scheduling behaviour through abstract states and transitions, subsequent works have extended the framework in several directions. The approach was generalised to globally scheduled multiprocessor systems in [48], and later to parallel DAG task models in [49]. Further work incorporated additional system characteristics such as resource sharing with First-In First-Out (FIFO) spin locks [51], moldable gang tasks [50], and conditional DAG task models [30].

A recurring challenge for SAG-based analysis is state-space explosion. This issue was addressed in [54] through partial-order reduction techniques, which were later extended to DAG task models in [53]. More recent work has further broadened the applicability of the framework, for example by extending it to pre-emptive scheduling models [29].

Beyond schedulability analysis, SAG has also been used for diagnostic purposes. For example, [67] showed that the framework can generate counterexample schedules, making it possible to identify concrete deadline-miss scenarios when schedulability cannot be guaranteed.

This brief overview shows that SAG is not a single fixed analysis, but rather a general framework whose exact form depends on the task model and the timing information that must be retained. In this thesis, the framework is developed further in two directions. In Part I, it is extended to the Event-Driven Delay-Induced task model, which combines DAG-style dependencies with self-suspension behaviour. This allows to model and analyse key properties (e.g., precedence constraints, parallelism, access to co-processors and I/O and communication delays) found in most real-time systems. In Part II, the abstraction principles underlying SAG are transferred beyond processor scheduling to the communication domain of Time-Sensitive Networking (TSN), where the scheduled entities are packets rather than processor jobs.

2.5 Intuition Behind the Schedule Abstraction Graph

To explain the main ideas underlying the Schedule Abstraction Graph, this section presents a basic version of the framework for a simple task model scheduled non-

pre-emptively on a single processor. This restricted setting is chosen deliberately: it provides a clear starting point for understanding the abstraction principles of SAG before the richer task and communication models considered in later chapters are introduced.

Although the models studied later in this thesis are substantially more expressive, the same core idea remains: scheduling behaviour is explored in a structured way without explicitly enumerating every concrete scenario. Instead of analysing each possible combination of arrivals and execution times separately, scenarios that lead to the same scheduling behaviour are treated together.

This raises several questions. How does SAG determine which scenarios can be grouped together? What information must be retained to make subsequent scheduling decisions? How are different scheduling scenarios abstracted, and how are they represented within the graph? The remainder of this section addresses these questions.

2.5.1 Making a Scheduling Decision

Consider a particular point in time within a concrete schedule. In order to determine what the next event in the schedule will be, certain information about the jobs in the system must be known.

First, it must be known which jobs have already been dispatched and completed. The exact order in which they executed is not necessary, but identifying the jobs that have finished allows them to be excluded from further consideration.

Second, information about the jobs that have not yet been dispatched is required. In particular, the arrival times of these jobs must be known in order to determine when they become eligible for execution.

Third, the priorities of the jobs must be known. When multiple jobs are eligible to execute at the same time, the scheduler resolves the contention based on these priorities.

Finally, to decide when any of the eligible jobs can actually be run, we also need information about the processor. We must know whether the processor is currently idle and able to take on another job, or already busy executing one.

The information described above can be divided into two categories. Some information remains fixed regardless of where a particular execution is within the schedule. For example, job priorities and job arrival times are properties of the system itself and do not change during scheduling.

Other information depends on the current position within the schedule. In particular, the set of jobs that have already been dispatched and the status of the processor depend on the specific execution that has occurred up to that point.

To capture this information, the notion of a *concrete state* is introduced next.

2.5.2 Concrete States

A concrete state represents the relevant information about the system at a particular point during scheduling. From such a concrete state, it becomes possible to determine what the next scheduling decision may be and how the system may evolve.

In a concrete state two pieces of information are recorded. First, the set of jobs that have already been dispatched is stored. This set is denoted by $\mathcal{D}$. Second, the status of the processor is recorded, that is, whether the processor is currently executing a job or is free.

To understand how a schedule can be represented in terms of concrete states, consider one of the schedules shown in Figure 2.2. For this schedule, a state can be drawn for every time instance of the execution, as illustrated in Figure 2.3. Each state captures the current situation of the system at that point in time, including which jobs have already been dispatched and whether the processor is currently occupied.

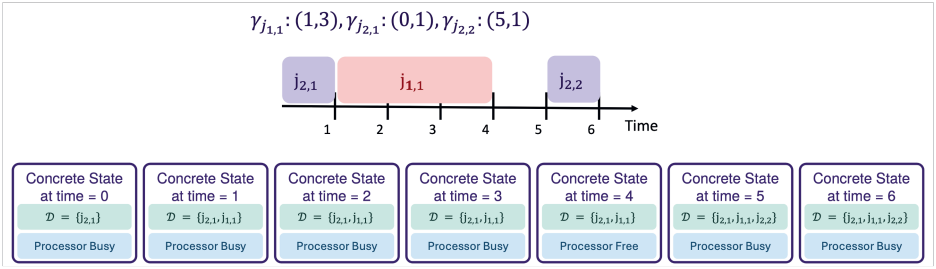


Figure 2.3: Concrete states corresponding to every time instance of a schedule

This representation corresponds to the most straightforward approach: storing a separate state for every point in time for every possible schedule. However, as seen earlier, the number of possible schedules can grow rapidly even for small systems. Representing schedules in this way would therefore lead to a rapid growth in number of states, resulting in a state-space explosion.

To address this issue, the Schedule Abstraction Graph applies two main abstraction strategies. The first reduces number of states by considering only those moments in time where scheduling decisions may occur. The second aggregates similar concrete states together into abstract states. These two ideas are introduced in the following subsections.

2.5.3 Relevant Scheduling Events

In the representation described above, a concrete state is stored for every time instance of a schedule. However, not every point in time leads to a change in the

scheduling behaviour.

In Figure 2.3, it can be observed that the concrete states at time instances 2 and 3 are identical. Storing separate states for these intermediate time instances does not provide any additional information about the scheduling behaviour.

For this reason, only those concrete states that correspond to potential scheduling decision points are retained. In the system model considered here, where jobs execute non-preemptively on a single processor, such decision points occur when a new job can be scheduled. Since each job is non-preemptive and therefore executes until completion once it starts, the next concrete state of interest occurs at the earliest time at which both the current job has completed its execution and another job is ready for dispatch.

Instead of storing all intermediate states, the execution can be represented by moving directly from one relevant concrete state to the next. The intermediate sequence of concrete states can be replaced by a transition between these two states, labelled by the job that is executing during this interval.

This idea is illustrated in Figure 2.4, where the original schedule containing seven concrete states is reduced to three relevant concrete states.

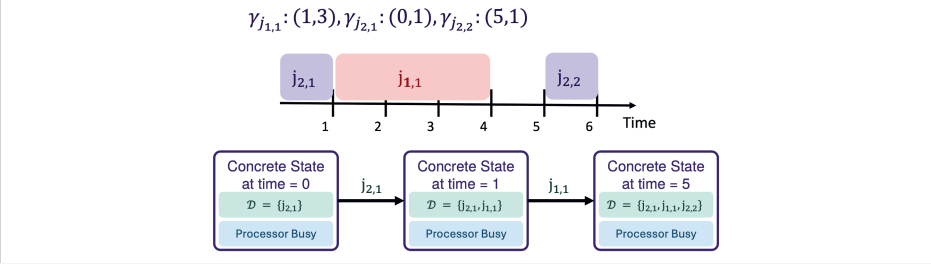


Figure 2.4: Reduction of a schedule to its relevant concrete states

By restricting attention to these relevant scheduling events, a large number of intermediate concrete states can be removed from consideration.

2.5.4 Abstract States

Even after removing intermediate states that do not affect scheduling behaviour, number of remaining concrete states may still be large because many different schedules must still be explored.

To illustrate this, consider the example in Figure 2.5, where a subset of ten different concrete schedules from Figure 2.2 is shown. Although these schedules are distinct, they all dispatch the jobs in the same order. In each of these schedules, the first transition corresponds to job $j_{2,1}$ being dispatched, and the second transition corresponds to job $j_{1,1}$ being dispatched. As a result, the relevant

concrete states in all these schedules can be reduced to three decision points.

Despite this similarity, the schedules themselves are not identical. The differences arise from the release times of the jobs, which may vary due to jitter. These variations lead to different completion times for the running jobs and therefore different times at which the processor becomes available.

However, if the first concrete state of each of these schedules is examined, a common pattern can be observed. In all cases, job $j_{2,1}$ has been dispatched, and the processor will become available within a certain time interval determined by the possible completion times of that job. Instead of storing each of these concrete states separately, their information can be combined.

This combined information is captured in what we call an *abstract state*. An abstract state records two components: the set of dispatched jobs, written as $\mathcal{D}$, and an interval $[A^{\min}, A^{\max}]$ that bounds, respectively, the earliest time at which a processor might become available and the earliest time by which a processor is guaranteed to be available.

Using this abstraction, the twelve schedules in Figure 2.5, which contain thirty-six relevant concrete states in total, can be represented by only three abstract states.

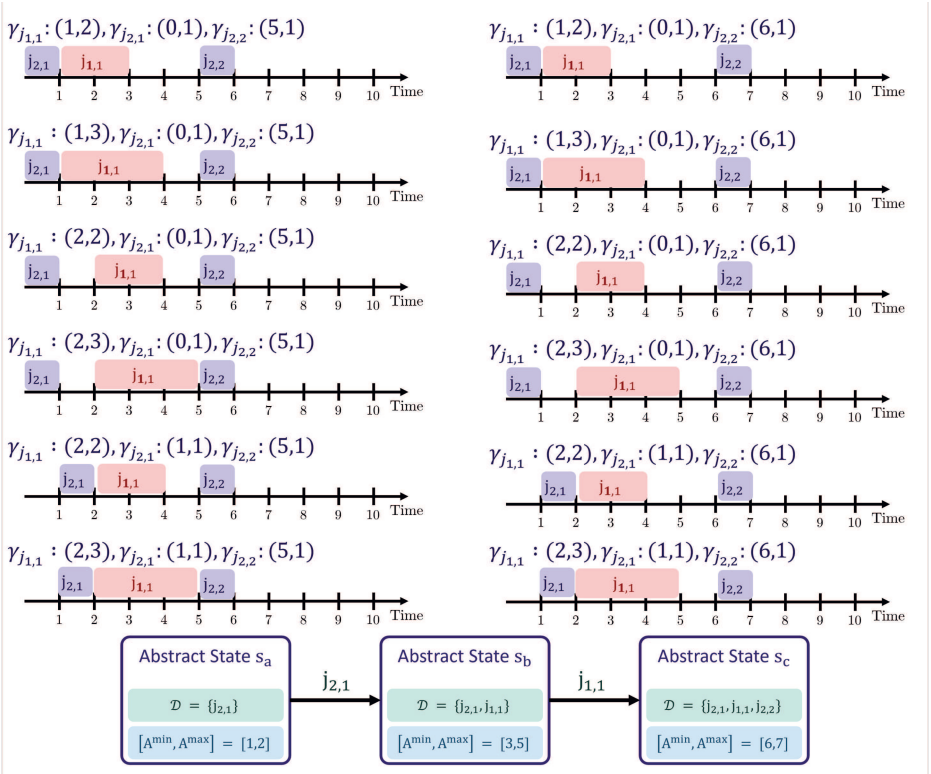


Figure 2.5: Reduction of multiple schedules to its abstract states.

More generally, similar concrete states are aggregated together into abstract states. An abstract state therefore represents a set of concrete states that share the same relevant scheduling information. Through this aggregation, many concrete scheduling situations can be represented by a single state, significantly reducing the number of states that must be explored.

2.5.5 Constructing the Schedule Abstraction Graph

The abstractions introduced above naturally lead to a graphical representation of the scheduling behaviour. Each abstract state represents a set of concrete scheduling situations that share the same relevant information, and transitions between abstract states represent the evolution of the system when a scheduling event occurs.

By connecting abstract states through such transitions, a directed graph structure is obtained. This graph can be represented as

$$G = (V, E),$$

where V is the set of abstract states and E is the set of transitions between abstract states. The vertices of this graph therefore correspond to abstract states, while the edges represent the possible scheduling decisions that transitions the system from one abstract state to another.

This structure is referred to as the *Schedule Abstraction Graph*. Starting from an initial abstract state that represents the system at time 0, before any job has been dispatched, the graph captures all possible evolutions of the system under the given task model and scheduling policy.

The process of constructing the graph begins from the initial abstract state, since this is the natural starting point for every possible schedule in the system. From a given abstract state, the possible jobs that may be scheduled next are determined.

For every possible next job to be scheduled, a successor abstract state is constructed. The corresponding scheduling decision labels the transition from the current abstract state to this successor state. Repeating this process for every newly generated abstract state gradually builds the full graph.

In this way, the Schedule Abstraction Graph provides a compact representation of the scheduling behaviour of the system. Rather than exploring every concrete schedule separately, many schedules are explored simultaneously through the abstract states and transitions of the graph.

For the example job set shown in Figure 2.1, applying this construction yields the graph shown in Figure 2.6 which abstracts all the schedules shown in Figure

2.2. It can be seen that this small graph, containing only 7 abstract states, is sufficient to represent all 24 possible schedules.

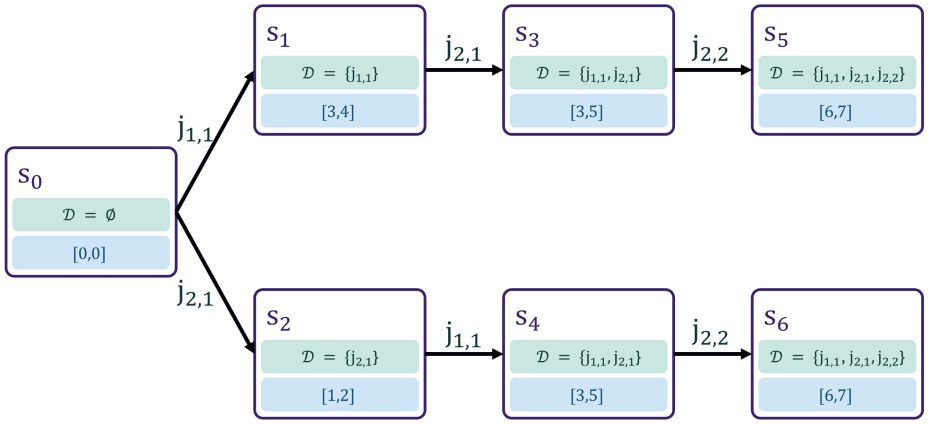


Figure 2.6: Schedule Abstraction Graph for jobset in Figure 2.1

2.5.6 Merging Abstract States

Although the abstraction steps described above significantly reduce number of states that must be explored, the size of the Schedule Abstraction Graph may still grow large. In particular, variations in job arrival times due to jitter and differences in execution times can lead to many abstract states being generated, especially when the number of jobs in the system increases.

To further reduce size of the explored state space, an additional step is introduced: merging similar abstract states.

Consider the abstract states s_3 and s_4 shown in Figure 2.6. Examining the contents of these states reveals that both contain the same set of dispatched jobs. Furthermore, the processor availability intervals associated with these states are identical. Since the abstraction does not preserve the order in which jobs were dispatched, the two paths leading to these states represent equivalent scheduling situations. These states can therefore be merged into a single abstract state.

This observation leads to the following merging rule: two abstract states can be merged if they contain the same set of dispatched jobs and their processor availability intervals overlap. When such states are merged, the processor availability interval of the resulting state is updated to be the union of the two original intervals.

Applying this merging rule to the graph in Figure 2.6 reduces number of abstract states from seven to five, as illustrated in Figure 2.7. Although the reduction in this small example may appear modest, the effect becomes much

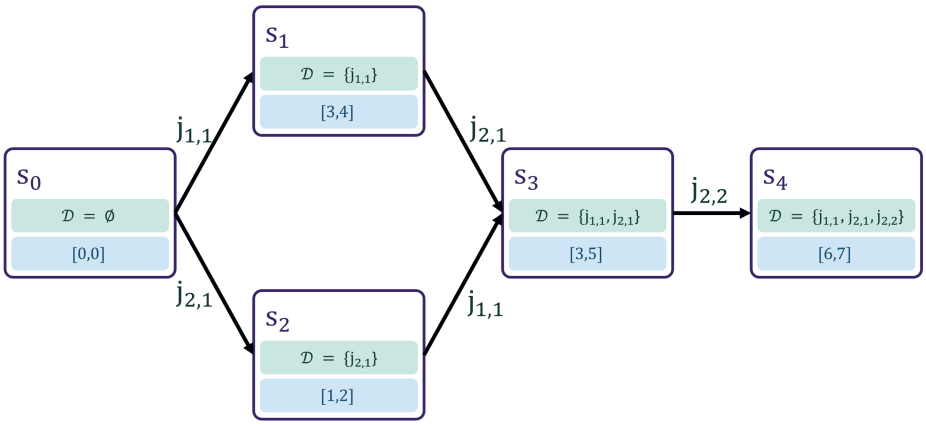


Figure 2.7: Schedule Abstraction Graph for jobset in Figure 2.1 with merging technique applied

more significant as the number of jobs and variability in their timing parameters increase. In larger systems, merging appropriate abstract states can substantially reduce the number of states that must be explored, thereby helping to curb state-space explosion.

Finally, since this graph is constructed to analyse the response time of the jobs and to determine whether they miss their deadlines or not, we can state that, for each job along every path, we consider both its best-case and worst-case response time behaviour. If, even in the worst case, the response time of every job explored in the graph still meets its deadline, then the corresponding job set is considered schedulable.

2.6 Formal Definition of the Schedule Abstraction Graph

Having established the intuition behind the Schedule Abstraction Graph, the approach can now be formalised. The formalisation given in this section intentionally focuses on the very basic version of SAG that we have seen so far, based on a simple task model scheduled on a single processor. The purpose of this restricted presentation is to provide a clear understanding of the fundamental working principles of the framework. It should therefore not be read as reflecting the full state of the art or the full range of task and system models that can currently be analysed using SAG. Rather, it serves as a conceptual starting point for the more advanced developments discussed later in the thesis.

This section presents the formal definition of an abstract state and the algorithm

used to construct the Schedule Abstraction Graph.

Formally, an abstract state s is defined as

$$s = (\mathcal{D}, [A^{\min}, A^{\max}]),$$

where $\mathcal{D}$ denotes the set of dispatched jobs, and the interval $[A^{\min}, A^{\max}]$ specifies, respectively, the earliest time at which the processor may become available and the earliest time at which it is guaranteed to be available.

Consider the problem of constructing a Schedule Abstraction Graph for a finite set of jobs $\mathcal{J}$. The construction begins with the initial abstract state

$$s_0 = (\emptyset, [0, 0]),$$

which represents the state of the system at time 0, before any job has been dispatched.

The graph is then explored layer by layer. Each layer contains abstract states at the same depth, where the depth corresponds to the number of dispatch decisions taken so far. The algorithm repeatedly expands the outermost layer of the partially constructed graph.

Two sets of states are maintained during the exploration:

- L^{cur}: the current layer being expanded, and
- L^{next}: the next layer produced by expanding L^{cur}.

Initially, $L^{\text{cur}} = \{s_0\}$ and $L^{\text{next}} = \emptyset$.

Each iteration of the algorithm consists of two phases.

Expansion phase. For every state $s \in L^{\text{cur}}$, the algorithm determines which jobs may be dispatched next from s. This step uses the information stored in s (the set of already dispatched jobs and the processor-availability interval) together with the job parameters (release bounds, execution bounds, priorities, and any model-specific readiness constraints).

For each job j that may be dispatched next:

- the bounds on its finish time, denoted by $[f_j^{\min}, f_j^{\max}]$, are determined,
- the global best-case and worst-case response-time bounds for j, denoted by BCRT_j and WCRT_j, are updated,
- and if a deadline miss is possible, the algorithm terminates and declares the system unschedulable.

If no deadline violation is detected, the successor abstract state obtained after dispatching j is constructed and inserted into L^{next}.

Merging phase. After all states in L^{cur} have been expanded, the algorithm merges compatible states in L^{next}. Merging replaces multiple similar abstract states by a single state that safely over-approximates them.

Once merging is complete, the algorithm proceeds to the next iteration by setting $L^{\text{cur}} \leftarrow L^{\text{next}}$ and clearing L^{next}.

Termination. If the exploration ends without encountering a deadline miss, then all relevant scheduling behaviour within the observation window has been analysed. In that case, the system is declared schedulable, and the computed global bounds on best-case and worst-case response times can be reported.

Algorithm 1 Schedule Abstraction Graph Construction

Input: Finite job set $\mathcal{J}$
for all $j \in \mathcal{J}$ **do**
    $\text{BCRT}_j \leftarrow \infty$
    $\text{WCRT}_j \leftarrow 0$
end for
$s_0 \leftarrow (\emptyset,\ [0,0])$
$L^{\text{cur}} \leftarrow \{s_0\}$
$L^{\text{next}} \leftarrow \emptyset$
while $L^{\text{cur}} \neq \emptyset$ **do**
    for all $s \in L^{\text{cur}}$ **do**
        Find the set of all eligible jobs
        for all j in the set of eligible jobs **do**
            Compute $[f_j^{\min}, f_j^{\max}]$
            $\text{BCRT}_j \leftarrow \min\{\text{BCRT}_j,\ f_j^{\min} - a_j^{\min}\}$
            $\text{WCRT}_j \leftarrow \max\{\text{WCRT}_j,\ f_j^{\max} - a_j^{\min}\}$
            if $f_j^{\max} > d_j$ **then**
                return unschedulable
            end if
            Create successor state s'
            $L^{\text{next}} \leftarrow L^{\text{next}} \cup \{s'\}$
        end for
    end for
    Merge all states in L^{next}
    $L^{\text{cur}} \leftarrow L^{\text{next}}$
    $L^{\text{next}} \leftarrow \emptyset$
end while
return schedulable and $\{\text{BCRT}_j, \text{WCRT}_j\}_{j \in \mathcal{J}}$

Algorithm 1 presents the Schedule Abstraction Graph construction procedure at the level required for this chapter. The expansion step (computing which jobs may be dispatched next and how successor states are formed) and the merging step (deciding which states can be merged and how their availability intervals are combined) are treated as subroutines, since their exact definitions depend on the task model and on the timing information that must be retained.

Algorithm 1 also highlights the overall structure of the analysis: the graph is constructed layer by layer, response-time bounds are updated during expansion, deadline violations are detected as soon as they become possible, and merging

is applied systematically to keep the representation compact. This way, the Schedule Abstraction Graph provides a systematic and finite exploration of all relevant scheduling behaviours within the observation window, while maintaining safe response-time bounds and controlling the growth of the explored state space.

2.6.1 Extension to Global Multiprocessor Scheduling

The intuition presented above considered the simplest setting of non-pre-emptive scheduling on a single processor. In that case, the timing information retained in an abstract state is captured by a single processor-availability interval, and an abstract state is defined as

$$s = (\mathcal{D}, [A^{\min}, A^{\max}]).$$

Here, $A^{\min}$ denotes the earliest time at which the processor may become available, and $A^{\max}$ denotes the earliest time at which the processor is guaranteed to be available.

For global multiprocessor scheduling, this representation must be generalised, since the scheduler must reason about the availability of several processors rather than just one. Accordingly, the single interval $[A^{\min}, A^{\max}]$ is replaced by a list of m intervals, where m is the number of processors:

$$s = \big(\mathcal{D},\ [A_1^{\min}, A_1^{\max}], [A_2^{\min}, A_2^{\max}], \ldots, [A_m^{\min}, A_m^{\max}]\big).$$

The meaning of these intervals is important. The interval $[A_1^{\min}, A_1^{\max}]$ characterises the availability of one processor: $A_1^{\min}$ denotes the earliest time at which at least one processor may possibly be available, while $A_1^{\max}$ denotes the earliest time at which at least one processor is guaranteed to be available. Importantly, this does not refer to any particular processor but only the existence of some available processor matters. More generally, for each $l \in \{1, \ldots, m\}$, $A_l^{\min}$ denotes the earliest time at which at least l processors may be available, and $A_l^{\max}$ denotes the earliest time at which at least l processors are guaranteed to be available. In particular, $A_m^{\min}$ and $A_m^{\max}$ characterise the earliest possible and guaranteed availability of all m processors.

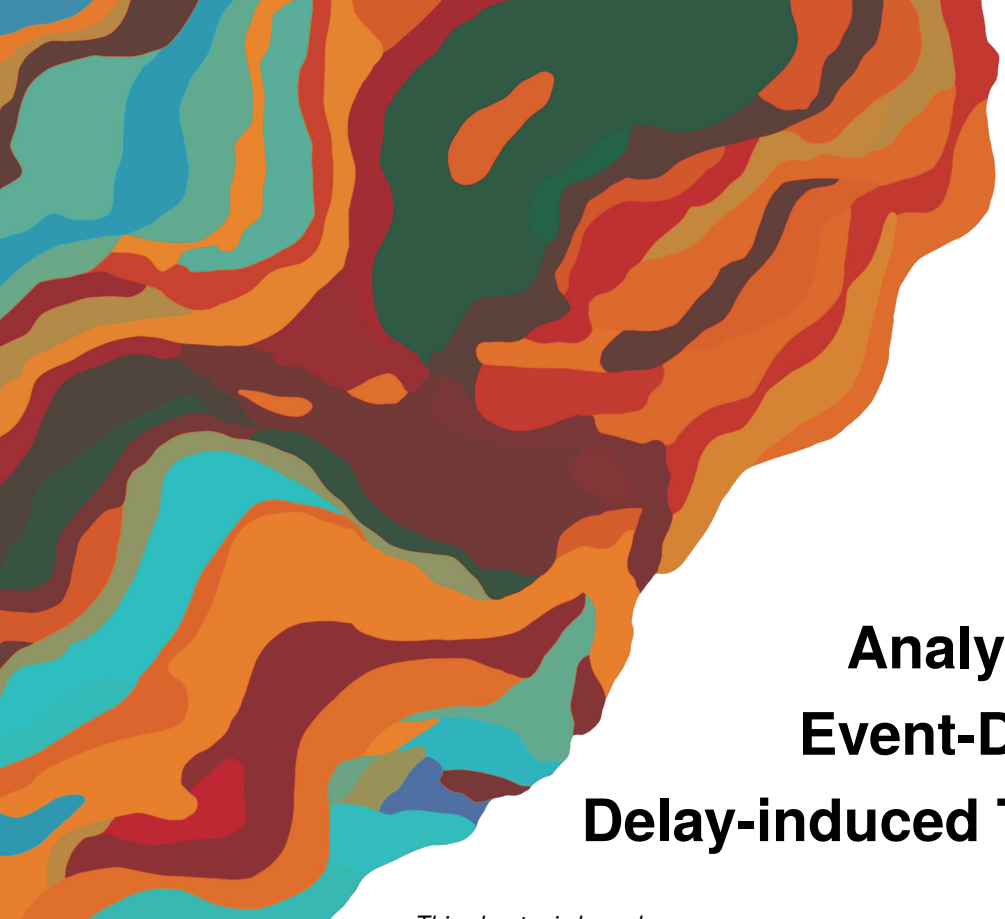
This extension of the Schedule Abstraction Graph to multiprocessor scheduling is presented in [48, 49]. In particular, the further extension to DAG task models in [49] forms an important basis for the extension of SAG to the Event-Driven Delay-Induced task model developed later in Chapter 3. By contrast, the basic single-processor intuition presented in this chapter serves as the conceptual starting point for the communication-domain developments in Part II, in particular in Chapters 5 and 6.

2.7 Scope of the Schedule Abstraction Graph in this Thesis

In this thesis, the framework is extended in two directions. In Part I, a further extension of the SAG framework is presented for analysing the Event-Driven Delay-Induced task model. This model combines concepts from DAG task models with self-suspending tasks, enabling the analysis of systems where task execution is driven by complex event dependencies and delays. The details of this extension are presented in Chapter 3.

It is important to note that all related work discussed so far focuses on analysing jobs or tasks scheduled on processors. In other words, the Schedule Abstraction Graph framework has primarily been applied to processor scheduling problems in real-time systems.

In Part II of this thesis, application of SAG methodology is extended beyond processor scheduling. The abstraction principles that form the basis of SAG are tailored to the context of Time-Sensitive Networking (TSN) in Chapters 5 and 6. In this setting, the entities being scheduled are network packets rather than processor jobs, and the scheduling decisions correspond to packet transmissions across network links. This demonstrates that the Schedule Abstraction Graph methodology is not limited to processor scheduling, but can also be applied to analyse scheduling behaviour in communication networks.



3

Analysis of Event-Driven Delay-induced Tasks

This chapter is based on:
Srinidhi Srinivasan, Mario Günzel, and Geoffrey Nelissen. "Response-Time Analysis for Limited-Preemptive Self-Suspending and Event-Driven Delay-Induced Tasks". In: 2024 IEEE Real-Time Systems Symposium (RTSS). *2024, pp. 29–42.* DOI: *10.1109/RTSS62706.2024.00013.*

TABLE OF CONTENTS

3.1 System Model
3.2 Analysis
3.3 Expansion Phase
3.4 Merging Phase
3.5 Correctness
3.6 Evaluation
3.7 Conclusion

In Chapter 2, the Schedule Abstraction Graph (SAG) was introduced as a structured method for analysing real-time task scheduling on a uniprocessor platform. The approach was formulated for periodic tasks with bounded execution times under deterministic and work-conserving scheduling policies. It was shown that the abstraction allows systematic exploration of scheduling decisions while deferring state-space explosion through carefully designed state aggregation.

However, many modern embedded real-time systems cannot be adequately modelled as independent periodic tasks consisting of a single execution segment. Heterogeneous computing platforms integrating multicore Central Processing Unit (CPU)s together with hardware accelerators such as Graphics Processing Unit (GPU)s, Tensor Processing Unit (TPU)s, or Field-Programmable Gate Array (FPGA)s are increasingly common in embedded and cyber-physical systems. Applications in assisted driving, smart manufacturing, robotics, and autonomous drones frequently execute data-intensive workloads across multiple computing resources.

Such applications typically:

- offload computation to accelerators,
- execute parallel processing stages,
- synchronise on data produced by other components,
- and experience bounded but non-negligible delays between computation segments.

These features imply that different execution segments of the same application are not temporally independent i.e., the activation or completion of one segment may depend on the outcome of another. Since classical independent task models assume tasks can be analysed in isolation, they are not well suited to capture the timing behaviour of such systems.

Two task models have traditionally been used to represent such behaviour.

Self-suspending tasks The self-suspending task model captures tasks that suspend their execution while waiting for external events, Input/Output (I/O) operations, or the completion of offloaded computation. The task is divided into computation segments separated by suspension intervals. This model is well-suited to describe blocking due to I/O device interactions or off-loaded computation on co-processors or remote servers. However, it still represents tasks as a sequence of operations and therefore does not capture parallel execution within a task.

Directed Acyclic Graph tasks The directed acyclic graph (DAG) task model represents each task as a graph whose vertices correspond to execution segments and whose edges encode precedence constraints. This model naturally captures parallelism and internal task dependencies. However, edges in classical DAG models only represent ordering constraints and do not capture bounded delays between the completion of one segment and the earliest start time of another, and therefore cannot model computation off-loading or I/O device interactions.

Recently the *Event-Driven Delay-Induced (EDD)* task model [3] was introduced to unify the DAG and self-suspending task models. An EDD task is represented as a directed acyclic graph in which:

- vertices correspond to execution segments,
- edges represent precedence constraints,
- and each edge is additionally labelled with minimum and maximum delay bounds.

The delay bounds model the time between the completion of a predecessor segment and the earliest possible start of its successor. These delays may represent waiting for sensor data, completion of accelerator computation, communication acknowledgements, or other external events.

The EDD task model strictly generalises both the segmented self-suspending task model and classical DAG tasks:

- A segmented self-suspending task corresponds to an EDD task whose graph has a single path.
- A classical DAG task corresponds to an EDD task in which all edge delays are equal to zero.

The EDD model therefore provides a unified abstraction capable of expressing parallelism, precedence constraints, and bounded suspension delays within a single framework.

State-of-the-art schedulability analyses for DAG, self-suspended, and EDD tasks are mostly focused on fully pre-emptive scheduling. However, the execution time of data-intensive applications is often susceptible to the availability of their data in cache and main memory. Their performance may thus be very negatively impacted by data evictions caused by preemptions by other tasks. Therefore, it was argued in [16] that limited-pre-emptive scheduling is a more suitable solution to meet the timing and performance requirements of such applications since it allows executing critical execution segments non-pre-emptively and restricts preemptions to occur only between segments with limited data dependency.

There exist only two schedulability analyses for non-pre-emptive self-suspending tasks [65, 18]. The analysis by Casini et al. [18] is limited to single core scheduling, or partitioned scheduling on multicore platforms. The analysis by Yalcinkaya et al. [65] uses UPPAAL to analyse segmented self-suspending tasks executed with global fixed-priority scheduling on a multicore platform. The analysis presented in [65] is an exact schedulability analysis but it does not return bounds on the worst-case response time (Worst-Case Response Time (WCRT)) of the tasks.

Foughali et al. [26] proposed another UPPAAL-based schedulability analysis for limited-pre-emptive periodic tasks. Their approach scales better than the analysis in [65] as the number of tasks and execution segments increases, and it also provides bounds on the WCRT of the tasks. However, unlike [65], their analysis is restricted to single-core platforms and considers sequential tasks without self-suspensions or release jitter. On the other hand, their model supports conditional execution branches, which are not supported by the analysis in [65].

The only existing analysis for the generalised EDD task model was proposed by Aromolo et al. [3]. It proposes a response time analysis for pre-emptive fixed-priority partitioned scheduling on multicore platforms. To date, there is no analysis for non-pre-emptive or limited pre-emptive scheduling, no analysis for global scheduling, and no analysis for job-level fixed priority scheduling (which covers task-level fixed priority and Earliest Deadline First (EDF) scheduling among others). We fill all these gaps in this work.

Limited pre-emptive DAG tasks under both partitioned and global scheduling have been studied in the past for several scheduling policies [64]. The schedule abstraction graph (SAG) analysis [49] was shown to be the most accurate response time analysis for non-pre-emptive global scheduling of DAG tasks on multicore platforms [64, 49]. For that reason, in this work, we develop a response time analysis for global scheduling of limited-pre-emptive EDD tasks following the SAG principles.

Table 3.1 compares the capabilities of our proposed analysis proposed to the state of the art. Besides the fact that this analysis is the first for limited-pre-emptive EDD tasks, we highlight the following: Yalcinkaya's test does not return response time bounds (it is only a schedulability test), and both Yalcinkaya's and Casini's tests are limited to task-level fixed priority scheduling, when the analysis presented in this chapter can analyse the larger class of job-level fixed priority scheduling. Foughali's and Casini's tests are limited to single core or partitioned scheduling. Additionally, Foughali's test cannot analyse self-suspensions nor DAGs.

Extending the Schedule Abstraction Graph In this chapter, we extend SAG framework to support limited-pre-emptive Event-Driven Delay-Induced (EDD)

Criterion	Casini [18]	Yalcinkaya [65]	SAG [49]	Foughali [26]	This work
WCRT	✓		✓	✓	✓
Self-Susp.	✓	✓			✓
DAG			✓		✓
EDD					✓
Global		✓	✓		✓
Task-level Fixed Priority (FP)	✓	✓	✓	✓	✓
Job-level FP			✓		✓

Table 3.1: State-of-the-art analyses comparison with respect to whether they provide response time bounds, the task models, and scheduling policies they support.

tasks under global *job-level* fixed-priority scheduling. While the overall exploration structure remains the same as in the original formulation, the abstract state representation, eligibility conditions, successor construction, and merging criteria are refined to capture precedence constraints, suspension delays, global scheduling decisions, and explicit pre-emption points.

The resulting analysis framework is flexible and supports different task models and scheduling policies (e.g., FP, EDF). It also allows execution segments (or jobs) of the same task to have identical or different priorities and deadlines, enabling the assignment of intermediate deadlines to subtasks and prioritization of more critical parts of an application.

Building on these extensions, the chapter makes the following contributions:

1. the first response-time analysis for limited-pre-emptive EDD tasks under global *job-level* fixed-priority scheduling on multicore platforms,

2. the first response-time analysis for limited-pre-emptive self-suspending tasks under global *job-level fixed-priority* scheduling,

3. improved response-time bounds for limited-pre-emptive DAG tasks under global job-level fixed-priority scheduling compared to the state of the art,

4. response-time analysis for limited-pre-emptive self-suspending tasks on single-core platforms that outperforms existing approaches.

3.1 System Model

We consider a finite set of n limited-pre-emptive Event-Driven Delay-Induced (EDD) tasks

$$\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$$

executing on a platform composed of m identical homogeneous processors. Tasks are scheduled according to a limited-pre-emptive, *global*, *work-conserving*, *job-level fixed-priority* (JLFP) scheduling policy.

The scheduler is work-conserving, meaning that no processor remains idle whenever a ready job exists. The scheduling policy is deterministic: whenever a processor becomes available, the scheduler always selects the same highest-priority job for any given set of ready jobs.

Pre-emptions are only allowed at explicitly defined *pre-emption points*, which coincide with segment boundaries (see the task model below). Each execution segment executes non-pre-emptively once started.

Each task $\tau \in \mathcal{T}$ is modelled as a directed acyclic graph (DAG)

$$\tau = (V_\tau, E_\tau),$$

where:

- V_τ is a finite set of vertices representing execution segments,
- $E_\tau \subseteq V_\tau \times V_\tau$ is a set of directed edges encoding precedence constraints.

Each vertex $v \in V_\tau$ is labelled with:

$$C^{\min}_{\tau,v}, \quad C^{\max}_{\tau,v}$$

representing the best-case and worst-case execution times (BCET and WCET) of the corresponding segment.

Each edge $e = (v, v') \in E_\tau$ is labelled with:

$$sus^{\min}_{\tau,v,v'}, \quad sus^{\max}_{\tau,v,v'},$$

representing the minimum and maximum suspension (delay) time between the completion of segment v and the earliest possible start of segment v'.

EXAMPLE 3.1

Let us consider an object detection task using radar and Light Detection And Ranging (LiDAR) data that offloads part of the computation on an FPGA to accelerate the treatment of the LiDAR points-cloud. Figure 3.1 shows how this task can be modelled as an EDD task. It starts with a node v_1 representing the execution of initialization code followed by two nodes v_2 and v_3 representing the treatment of the radar and LiDAR data in two parallel threads. The execution of v_2 and v_3 can start immediately after the completion of v_1. Therefore, the minimum and maximum delays on the edges leading

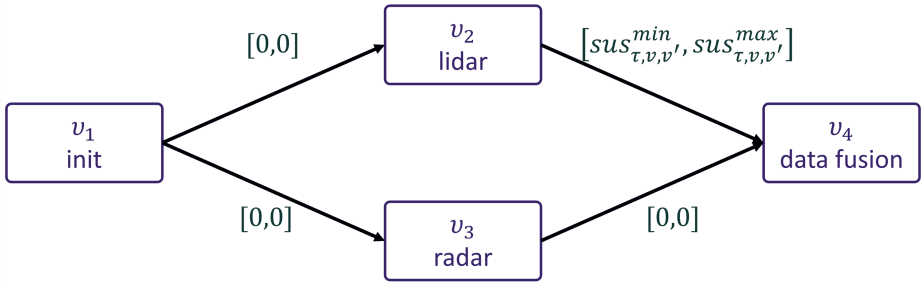


Figure 3.1: An example of an EDD task. Numbers on the edges model the minimum and maximum suspension time between nodes

to these nodes are both set to 0. The data fusion requires both parallel sections v_2 and v_3 to be completed first. Since the LiDAR processing is partly offloaded to an FPGA, there is a delay between the completion of v_2 and the earliest time the data fusion may start. That delay is bounded by $sus^{\min}_{\tau,v_2,v_3}$ and $sus^{\max}_{\tau,v_2,v_3}$ as shown on the edge from v_2 to v_4.

We consider applications that need their tasks to be scheduled according to a deterministic arrival pattern (e.g., typical task chains in Robot Operating System 2 (ROS2) [39] or Apollo Cyber RT [2] that are triggered by a timer, or more generally most tasks performing control, object recognition, obstacle detection, etc.,). On each of its arrivals, the task releases one job for each execution segment in its set of vertices V_τ.

Since the arrival pattern is assumed to be known beforehand, the problem reduces to the analysis of finitely many non-pre-emptive jobs in a limited observation window. For example, if the tasks are periodic without offset and have constrained deadlines, then it is enough to analyse the jobs released in an observation window equal to a single hyperperiod. The observation window length can be adapted to the specific arrival pattern being analysed. Therefore, our solution is not limited to the analysis of periodic tasks. Tasks with release bursts, multi-frame tasks, multi-period tasks or any other tasks with predictable arrival patterns can also be analysed.

The notion of a job in this chapter differs from that used in Chapter 2.

In Chapter 2, each task activation generated exactly one job corresponding to a single sequential execution unit. In the present model, however, each task τ is represented as a directed acyclic graph whose vertices correspond to execution segments. Consequently, each arrival of a task τ generates *one job for each vertex* $v \in V_\tau$.

Thus, the job set $\mathcal{J}$ consists of execution-segment jobs rather than whole-task

jobs. This refinement is necessary to capture:

- internal parallelism within a task,
- precedence constraints between segments,
- and bounded suspension delays between segments.

Formally, each job $j \in \mathcal{J}$ is described by the tuple

$$j = \big([a_j^{\min}, a_j^{\max}], [C_j^{\min}, C_j^{\max}], d_j, \pi_j, pred_j\big),$$

where:

- $[a_j^{\min}, a_j^{\max}]$ denotes the arrival-time interval,
- $[C_j^{\min}, C_j^{\max}]$ denotes the execution-time interval (bounded by the Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET) of the job),
- d_j is the absolute deadline,
- π_j is the priority,
- $pred_j \subseteq \mathcal{J}$ is the set of immediate predecessor jobs of j.

Note that our model allows jobs released by different segments of the same task to have the same or different deadlines and/or priorities. This highlights the versatility of our model. We define $succ_j = \{j' \in \mathcal{J} \mid j \in pred_{j'}\}$ as the set of successors of j.

Given a job j, for each predecessor $j' \in pred_j$, a minimal suspension time $sus^{\min}(j, j')$ and maximal suspension time $sus^{\max}(j, j')$ is defined. This suspension time is inherited from the minimal and maximal delay between two execution segments defined at the task level, i.e., the minimal and maximal delay labelled on each edge $e \in E_\tau$. Similar to the execution time, the actual suspension time value $sus(j, j') \in [sus^{\min}(j, j'), sus^{\max}(j, j')]$ between the execution of two jobs with precedence constraints is unknown a priori.

We say that a job is *ready* when:

1. it has been released,
2. all predecessor jobs have completed,
3. and the corresponding suspension times have expired.

The time point where a job becomes ready is denoted by

$$\rho_j = \max\{a_j, \max_{j' \in pred_j} \{f_{j'} + sus(j, j')\}\}$$

where $f_{j'}$ is the time at which the predecessor j' finished.

In Chapter 2, the notion of an execution scenario was introduced to capture the variability in job arrivals and execution times. In that context, each job $j \in \mathcal{J}$ was associated with a pair $\gamma_j = (a_j, C_j)$, where a_j denotes its arrival time and C_j its execution time, and an execution scenario was defined as the collection of such assignments for all jobs. Since the scheduling policy was assumed to be deterministic, fixing these parameters uniquely determined the resulting schedule.

In the model considered in this work, however, jobs may also be subject to suspension delays due to precedence constraints and external processing. Consequently, the definition of an execution scenario must be extended to account not only for release and execution times, but also for the suspension delays between dependent jobs. For this reason, for each job $j \in \mathcal{J}$, we define

$$\gamma_j = (a_j, C_j, (sus(j, j'))_{j' \in pred_j}),$$

where $a_j \in [a_j^{\min}, a_j^{\max}]$ denotes the concrete arrival time of j, $C_j \in [C_j^{\min}, C_j^{\max}]$ denotes its concrete execution time, and $sus(j, j') \in [sus^{\min}(j, j'), sus^{\max}(j, j')]$ denotes the concrete suspension delay between a predecessor job $j' \in pred_j$ and job. An execution scenario is then defined as the collection of such assignments for all jobs, i.e., a set $\{\gamma_j\}_{\forall j \in \mathcal{J}}$.

Given that the scheduler is deterministic, meaning that it always takes the same scheduling decision for the same set of ready jobs and core availability times, fixing these parameters uniquely determines the sequence of scheduling decisions and therefore the finishing time f_j of every job $j \in \mathcal{J}$.

If the finishing time f_j does not exceed the absolute deadline d_j for all jobs $j \in \mathcal{J}$ for all execution scenarios $\gamma \in \Gamma$, then we say that the job set $\mathcal{J}$ (and by extension, the task set $\mathcal{T}$) is *schedulable*.

For ease of presentation, we assume discrete time, i.e., all time values are natural numbers in this chapter. In this setting, one time unit is treated as the smallest distinguishable time increment. The analysis itself, however, is not restricted to discrete time and also holds for dense time. In implementations with dense time, these offsets can instead be interpreted using an infinitesimally small arbitrary number $\epsilon > 0$.

3.2 Analysis

We analyse the best-case and worst-case response time of each task in $\mathcal{T}$ by analyzing the best-case and worst-case response time of each job in $\mathcal{J}$. To this end, similarly to Chapter 2, we explore the space of all possible schedules under all possible execution scenarios.

As discussed in Chapter 2, the Schedule Abstraction Graph (SAG) provides an abstraction of the scheduling behaviour of a job set by representing reachable abstract system states and the scheduling decisions leading from one state to another. The SAG construction introduced there, however, is limited to task models without suspension [49, 50]. Extending it to self-suspending tasks, and more generally to the EDD task model, requires particular care, as self-suspension behaviour is known to be inherently unintuitive and has led to many erroneous results in the literature [19]. Therefore, the abstraction must be extended so as to correctly capture the timing dependencies induced by precedence constraints and suspension delays.

3.2.1 Schedule Abstraction Graph for EDD Tasks

As in Chapter 2, the SAG abstracts the set of all possible execution scenarios of a job set by representing scheduling decisions and their effect on the system state. This information is then used to derive bounds on the best-case and worst-case response times of jobs, and thus of their corresponding tasks. Each path in the SAG reflects a sequence of job scheduling decisions, and each node represents a possible abstract system state reachable under at least one execution scenario (see Figure 3.2 for an example).

Each abstract system state s in the SAG abstracts the possible system states after the scheduling decisions described by the paths ending at s. We define $\Gamma(s) \subseteq \Gamma$ as the subset of execution scenarios, i.e., the set of all concrete combinations of arrival times, execution times, and suspension times of jobs in $\mathcal{J}$, that are compatible with the scheduling decisions described by the paths ending at s. An edge connecting a state s to s' represents a scheduling decision in which the scheduler dispatches a job j, thereby making the abstract system state represented by s evolve into the abstract system state represented by s'. The edge is labelled by the dispatched job.

Compared to the abstraction introduced in Chapter 2, the EDD setting requires additional information to be stored in each abstract state. In particular, the abstraction presented earlier considered a single-processor platform and therefore only needed one processor-availability interval, denoted by $[A^{\min}, A^{\max}]$, to capture when the processor could next execute a job. In the present multicore

setting, this must be generalised to m availability intervals, one for each possible number of available processors. Moreover, readiness no longer depends only on arrival times and processor availability, but also on predecessor completion times and suspension delays. To capture this, an abstract state s must keep track of additional information. An abstract state s is defined by the following information:

1. the set of jobs $\mathcal{D}$ that have already been dispatched for execution on a processor until reaching a system state abstracted by s,
2. m intervals $[A_i^{\min}, A_i^{\max}]$ for $i = 1, \ldots, m$ that model processor availability in the abstract state s,
3. the finishing time interval $[f_j^{\min}, f_j^{\max}]$ for each job $j \in \mathcal{D}$ that has been dispatched on a processor and has at least one successor that still needs to be executed, and
4. a set of at most m jobs $\chi \subseteq \mathcal{D}$ that are certainly executing in the system state abstracted by s.

The set of dispatched jobs $\mathcal{D}$ is simply the aggregation of all the jobs labelling the edges of any path ending in s, since, by construction, the jobs that have been dispatched before reaching an abstract state are precisely those appearing on the corresponding path in the graph.

The interpretation of processor-availability intervals extends the single-processor notion introduced in Chapter 2. The use of multiple processor-availability intervals in the multiprocessor setting was first introduced in [48]. For each $i \in \{1, \ldots, m\}$, the interval $[A_i^{\min}, A_i^{\max}]$ characterises the availability of at least i processors. More precisely:

- $A_1^{\min}$ is the earliest time at which at least one processor may be available;
- $A_1^{\max}$ is the earliest time from which at least one processor is guaranteed to be available;
- more generally, $A_i^{\min}$ is the earliest time at which at least i processors may be available;
- and $A_i^{\max}$ is the earliest time from which at least i processors are guaranteed to be available.

The finishing-time information is new with respect to the basic SAG abstraction and is required to determine when successor jobs may become ready. It is sufficient to track the finishing time only for those dispatched jobs whose completion may still influence future scheduling decisions, namely jobs that still have at least one successor not yet dispatched.

Finally, the set $\chi \subseteq \mathcal{D}$ of certainly executing jobs contains at most m jobs, since there are only m cores in the platform. Let j denote the last job in $\mathcal{D}$ dispatched before reaching state s. Then χ contains the jobs that are still executing when j begins execution. In other words,

$$\chi = \{j' \in \mathcal{D} \mid f_{j'}^{\min} > \sigma_j\},$$

where σ_j denotes the time when the last dispatched job j began and $f_{j'}^{\min}$ is the earliest finishing time of job j'.

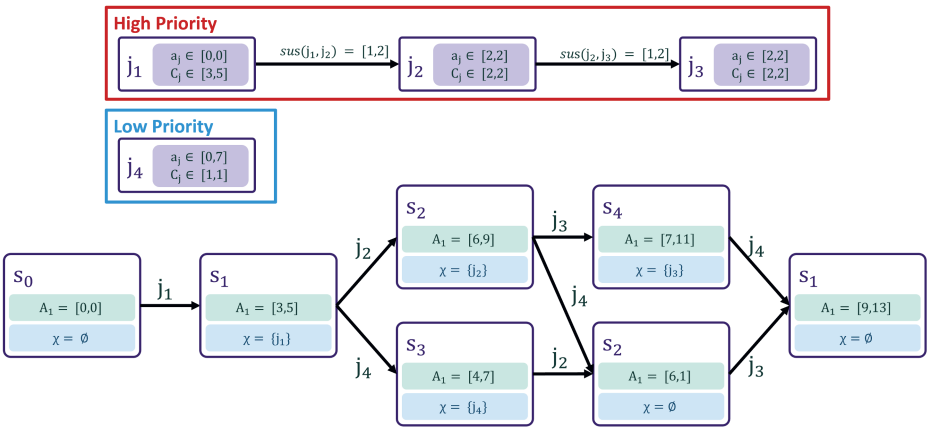


Figure 3.2: An example of generation of a schedule abstraction graph for EDD tasks

EXAMPLE 3.2

Consider the set of jobs shown in Figure 3.2 scheduled on a single-core platform. These jobs are released by two EDD tasks, more precisely by one sequential task and one self-suspending task, under fixed-priority scheduling. The self-suspending task consists of three execution segments and has the highest priority. The sequential task consists of a single execution segment and has the lowest priority. As shown in Figure 3.2, job j_1 has execution-time variation, with a BCET of 3 and a WCET of 5, the two suspension intervals may each vary between 1 and 2 time units, and j_4 experiences arrival jitter and may be released at any time in the interval $[0, 7]$. Due to these uncertainties in execution, suspension, and arrival times, even this small set of four jobs gives rise to many execution scenarios. More precisely, there are 96 different concrete assignments of values to the arrival, execution, and suspension parameters of the jobs. The corresponding SAG is shown in Figure 3.2. To improve readability, Figure 3.2 illustrates only the state

information relevant to the discussion. In particular, attributes such as job finishing intervals are omitted, although they remain part of the underlying abstract-state representation. As can be seen, the graph captures all 96 execution scenarios and abstracts all reachable system states using only 7 abstract states.

The SAG starts from an initial abstract state s_0, which represents the system state before any job has been dispatched on the platform. Hence, the availability interval $[A_1^{\min}, A_1^{\max}]$ is $[0, 0]$, meaning that the core is certainly available from time 0 onward. Since no job has been dispatched yet, both the set of dispatched jobs $\mathcal{D}$ and the set of certainly executing jobs χ are empty (Figure 3.2 shows only χ, as the content of $\mathcal{D}$ is given by the labels of the jobs appearing on the edges of paths ending at a state s).

Since, in this example, j_1 has the highest priority and certainly arrives at time 0, the scheduler always dispatches j_1 first in every execution scenario. Thus, j_1 is the only eligible job in the abstract state s_0, and the SAG contains a single successor abstract state s_1. Because j_1 executes for between 3 and 5 time units, the availability interval of the single core becomes $A_1 = [3, 5]$, signifying that the core may become available at time 3 and is certainly available at time 5 after the execution of j_1. Both the set of dispatched jobs $\mathcal{D}_1$ and the set of certainly executing jobs χ_1 in s_1 contain j_1.

Observe that all execution scenarios $\gamma \in \Gamma$ pass through the abstract states s_0 and s_1. Therefore, both abstract states account for all execution scenarios, and we write $\Gamma(s_0) = \Gamma(s_1) = \Gamma$.

Once the system reaches the state abstracted by s_1, the scheduler may make two different scheduling decisions depending on when j_4 is released and when j_2 becomes ready, i.e., when the suspension following the completion of j_1 expires. If j_4 is released before j_2 becomes ready, then j_4 is the next job dispatched; otherwise, j_2 is dispatched, since it has higher priority. Accordingly, the SAG branches after s_1 into two outgoing edges, one labelled j_4 and one labelled j_2. The core-availability interval and the set of certainly executing jobs are then updated in the corresponding successor states. Note that the sets of execution scenarios accounted for by the two successor abstract states are disjoint, i.e., $\Gamma(s_2) \cap \Gamma(s_3) = \emptyset$. However, all possible scheduling decisions from s_1 are represented, and therefore

$$\Gamma(s_2) \cup \Gamma(s_3) = \Gamma(s_1) = \Gamma.$$

Hence, all execution scenarios remain accounted for at each layer of the SAG.

The SAG is constructed using Algorithm 2. It takes the job set $\mathcal{J}$ as input and starts by initializing bounds on the best-case and worst-case response time of each job in $\mathcal{J}$ (Lines 2 to 4). It also creates the initial abstract state s_0 that represents the system state in which no job has yet been dispatched on the platform (Line 5). Then, as illustrated in Example 3.2.1, the graph is constructed layer by layer (Lines 6 to 32). The construction of each layer consists of two phases: an expansion phase and a merging phase.

Expansion Phase: For each abstract state in the current layer of the graph, denoted by L^{cur} in Algorithm 2, all possible next scheduling decisions are explored, i.e., for each abstract state $s_i \in L^{\mathrm{cur}}$, the analysis determines which jobs may be dispatched next in any concrete system state abstracted by s_i (Lines 9 to 12). For every such possible scheduling decision, that is, for every job j that may be dispatched by the scheduler in a concrete system state represented by s_i, a new abstract state is generated (Line 19) and added to the next layer of the SAG, denoted by L^{next}. At this stage, the earliest and latest finishing times of job j over all execution scenarios compatible with dispatching j from state s_i are also computed. These values are then used to update the best-case and worst-case response-time bounds of j and to check whether its deadline can be violated. The details of the expansion phase are provided in Section 3.3.

Merging Phase: To limit the growth of the graph, similar abstract states in the same layer are merged into a single abstract state (Lines 20 to 25). This preserves correctness while reducing the number of abstract states that must be expanded during the next expansion phase. The details of the merging phase are provided in Section 3.4.

The algorithm stops when there is no job left that can be dispatched in any concrete system state abstracted by the abstract states in the current layer. As already noted in Chapter 2, Algorithm 2 does not need to store the full graph explicitly. It is sufficient to keep only the current layer and the next layer under construction.

In this chapter, the algorithm is extended to account for precedence constraints, suspension delays, and limited-pre-emptive execution.

In the following sections, we therefore provide the concrete definitions of the expansion and merging procedures for EDD tasks, building upon the general SAG framework introduced in Chapter 2.

3.3 Expansion Phase

Assuming that the SAG is generated up to depth ℓ, this section details how we expand the graph to depth $\ell + 1$. To that end, we iterate over the abstract states

Algorithm 2 Schedule Abstraction Graph Construction for EDD Tasks

Input: EDD Jobset $\mathcal{J}$
for all $j \in \mathcal{J}$ **do**
  $\mathrm{BCRT}_j \leftarrow \infty$; $\mathrm{WCRT}_j \leftarrow 0$
    ▷ Initialize the bounds on the BCRT and WCRT of each job
end for
$s_0 = \{\mathcal{D} = \emptyset, A = ([0,0],...[0,0]), F = \emptyset, \chi = \emptyset\}$
    ▷ Build the initial system state
$L^{\mathrm{cur}} \leftarrow \{s_0\}$ ▷ Initialize the current layer of the SAG
$L^{\mathrm{next}} \leftarrow \emptyset$ ▷ Initialize the next layer of the SAG
while $L^{\mathrm{cur}} \neq \emptyset$ **do**
  for all $s_i \in L^{\mathrm{cur}}$ **do**
    Find the set of potentially ready jobs $\mathcal{R}$ using Equation (3.1)
    for each $j \in \mathcal{R}$ **do**
      if Corollary 3.2 is respected for j **then**
        $f_j^{\min}$ and $f_j^{\max} \leftarrow$ Equation (3.9) and (3.10)
          ▷ Earliest and latest finishing time of j
        $\mathrm{BCRT}_j \leftarrow \min\{\mathrm{BCRT}_j, f_j^{\min} - a_j^{\min}\}$
        $\mathrm{WCRT}_j \leftarrow \max\{\mathrm{WCRT}_j, f_j^{\max} - a_j^{\min}\}$
          ▷ Update the BCRT and WCRT bounds
        if $f_j^{\max} > d_j$ **then**
          return unschedulable
        end if
        Create new abstract state $\tilde{s}_i$ ▷ See Section 3.3.3
        if $\exists s_k \in L^{\mathrm{next}}$ that can merge with $\tilde{s}_i$ **then**
          ▷ See Section 3.4
          Merge s_k and $\tilde{s}_i$ in a state s' using Equation (3.22) to (3.27)
          $L^{\mathrm{next}} \leftarrow L^{\mathrm{next}} \cup \{s'\}$
          $L^{\mathrm{next}} \leftarrow L^{\mathrm{next}} \setminus \{s_k\}$
        else
          $L^{\mathrm{next}} \leftarrow L^{\mathrm{next}} \cup \{\tilde{s}_i\}$
        end if
      end if
    end for
  end for
  $L^{\mathrm{cur}} \leftarrow L^{\mathrm{next}}$
  $L^{\mathrm{next}} \leftarrow \emptyset$
end while
return schedulable

on the ℓ-th layer and generate new abstract states for the $(\ell+1)$-th layer. Given an abstract state s on the ℓ-th layer of the SAG, the generation of the abstract states that result from dispatching all jobs that may be scheduled by the scheduler in any concrete system state abstracted by s can be divided into 3 steps:

1. Determine all potentially ready jobs in a system state abstracted by s and calculate the interval $[\rho_j^{min}, \rho_j^{max}]$ in which they may become ready to execute.

2. Compute bounds σ_j^{min} and σ_j^{max} on the start time of each potentially ready job j.

3. For every job j that may be started in the system state abstracted by s, generate a new abstract state after scheduling j, and add it to the $(\ell+1)$-th

layer of the SAG.

3.3.1 Potentially Ready Jobs

First, we have to decide which scheduling decisions are possible in the system states abstracted by the abstract state s. That is, we have to determine which jobs may be started by the scheduler in the system state abstracted by s for any possible execution scenario in $\Gamma(s)$.

We know that only jobs $j \in \mathcal{J} \setminus \mathcal{D}$ that have not been dispatched yet can be started next. Moreover, the job j that is started next must be *ready* when it is started. That is, all its predecessors $pred_j$ must have been dispatched, i.e., $pred_j \subseteq \mathcal{D}$. Therefore, we define the set $\mathcal{R}$ of *potentially ready* jobs as

$$\mathcal{R} := \left\{ j \in \mathcal{J} \setminus \mathcal{D} \,\middle|\, pred_j \subseteq \mathcal{D} \right\}. \tag{3.1}$$

Each potentially ready job $j \in \mathcal{R}$ becomes ready as soon as (i) it has arrived, (ii) all its predecessors finish their execution, and (iii) the suspension times between all its predecessors and j expire, Hence, the ready time ρ_j of j under any execution scenario $\gamma \in \Gamma(s)$ is given by

$$\rho_j = \max(a_j, \max\left\{ f_{j'} + sus(j, j') \,\middle|\, j' \in pred_j \right\}). \tag{3.2}$$

Note that Equation (3.2) uses concrete values, which will depend on the specific execution scenario considered. Since s abstracts a set of execution scenarios, we must account for all possible values a_j, $sus(j, j')$ and $f_{j'}$ may assume in the execution scenarios $\gamma \in \Gamma(s)$.

According to the system model, the arrival time a_j and suspension time $sus(j, j')$ are within the interval $[a_j^{\min}, a_j^{\max}]$ and $[sus^{\min}(j, j'), sus^{\max}(j, j')]$, respectively. For $f_{j'}$, as discussed in Section 3.2.1, a lower and upper bound $f_{j'}^{\min}$ and $f_{j'}^{\max}$ are stored in the abstract state s. Thus, a lower bound ρ_j^{min} and upper bound ρ_j^{max} on ρ_j are given by:

$$\rho_j^{min} := \quad \max\left(a_j^{\min}, \max\left\{ f_{j'}^{\min} + sus^{\min}(j, j') \mid j' \in pred_j \right\}\right) \tag{3.3}$$

$$\rho_j^{max} := \quad \max\left(a_j^{\max}, \max\left\{ f_{j'}^{\max} + sus^{\max}(j, j') \mid j' \in pred_j \right\}\right) \tag{3.4}$$

3.3.2 Start Times

Consider a *concrete* execution scenario $\gamma \in \Gamma(s)$. A single job j is selected by the scheduler to be started next after the abstract state s. There are two cases

that must be considered:

Case 1: j is already ready when a processor becomes available. In that case, j has to be the ready job with the highest priority when the processor becomes available.

Case 2: j is *not* ready when a processor becomes available. In that case, no other jobs must have become ready before j becomes ready (otherwise, those other jobs would be executed first due to the work-conserving property of the scheduling policy). Moreover, when j becomes ready, no higher-priority jobs become ready at the same time.

In the following, we consider those two cases to derive bounds for the starting time σ_j of the job j in the execution scenario γ.

Lemma 3.1. *If j starts as described in Case 1, then the following bounds hold for the starting time σ_j of j:*

1. $\rho_j^{min} \leq \sigma_j$
2. $A_1^{\min} \leq \sigma_j \leq A_1^{\max}$
3. $\sigma_j < \rho_{j'}^{max}$ *for all* $j' \in \mathcal{R}$ *with* $\pi_{j'} < \pi_j$.

Proof. We prove the three properties in order.

1. By definition, a job cannot start to execute before it is ready. Therefore, $\rho_j \leq \sigma_j$, and thus $\rho_j^{min} \leq \sigma_j$ since ρ_j^{min} is a lower bound on ρ_j.
2. According to the definition of Case 1, j is ready when the processor becomes available. Therefore, j is dispatched as soon as a processor becomes available. Since $A_1^{\min}$ and $A_1^{\max}$ are a lower and upper bound on when a processor becomes available, we must have $\sigma_j \in [A_1^{\min}, A_1^{\max}]$.
3. There may be no higher priority job ready when j starts, otherwise the higher priority job would be dispatched instead of j. Hence, $\sigma_j < \rho_{j'} \leq \rho_{j'}^{max}$ for all $j' \in \mathcal{R}$ with $\pi_{j'} < \pi_j$. □

Let $\bar{\rho}_j^{hp}$ be an upper bound on the time at which any higher priority job than j certainly becomes ready:

$$\bar{\rho}_j^{hp} := \min \left\{ \rho_{j'}^{max} \,\middle|\, j' \in \mathcal{R} \wedge \pi_{j'} < \pi_j \right\}. \tag{3.5}$$

We have the following corollary from Lemma 3.1

Corollary 3.2. *If j starts executing as described in Case 1, then* $\sigma_j \in \left[\max \left\{ \rho_j^{min}, A_1^{\min} \right\}, \min \left\{ A_1^{\max}, \bar{\rho}_j^{hp} - 1 \right\} \right]$.

Now, we examine Case 2.

Lemma 3.3. *If j starts as described in Case 2, then the following bounds hold for the starting time σ_j of j:*

1. $A_1^{\min} < \sigma_j$
2. $\rho_j^{min} \leq \sigma_j \leq \rho_j^{max}$
3. $\sigma_j \leq \rho_{j'}^{max}$ *for all* $j' \in \mathcal{R}$ *such that* $j' \neq j$
4. $\sigma_j < \rho_{j'}^{max}$ *for all* $j' \in \mathcal{R}$ *with* $\pi_{j'} < \pi_j$.

Proof. We prove the four properties in order.

1. According to the definition of Case 2, j is not ready when a processor becomes available. Therefore, j is started *after* the earliest time a processor may become available, that is, $\sigma_j > A_1^{\min}$.

2. Since, according to the definition of Case 2, a processor is available when j becomes ready and no higher priority job is ready, j is started as soon as it becomes ready. Therefore, $\sigma_j = \rho_j$ implying $\rho_j^{min} \leq \sigma_j \leq \rho_j^{max}$.

3. Because a processor is available before j starts, if another job $j' \in \mathcal{R}$ was ready before j starts, then, by the work-conserving property, j' would be executed on the free processor instead of j. Therefore, $\sigma_j \leq \rho_{j'} \leq \rho_{j'}^{max}$ must hold for all $j' \in \mathcal{R}$ such that $j \neq j'$.

4. If another job $j' \in \mathcal{R}$ with higher priority than j was ready at or before j starts, then that other job would be executed instead of j. Therefore, $\sigma_j < \rho_{j'} \leq \rho_{j'}^{max}$ must hold for all $j' \in \mathcal{R}$ with $\pi_{j'} < \pi_j$. □

Let $\bar{\rho}_j^{any}$ be an upper bound on the time at which any job certainly becomes ready:

$$\bar{\rho}_j^{any} := \min \left\{ \rho_{j'}^{max} \,\middle|\, j' \in \mathcal{R} \right\}. \tag{3.6}$$

We have the following corollary from Lemma 3.3

Corollary 3.4. *If j starts executing as described in Case 2, then*

$$\sigma_j \in \left[\max \left\{ \rho_j^{min}, A_1^{\min} + 1 \right\}, \min \left\{ \bar{\rho}_j^{any}, \bar{\rho}_j^{hp} - 1 \right\} \right].$$

Combining Corollaries 3.2 and 3.4, we derive bounds on the start time of j in Lemma 3.5.

Lemma 3.5. *If j is the next job started in any execution scenario abstracted by s, then the start time σ_j of j is in the interval $[\sigma_j^{min}, \sigma_j^{max}]$ with*

$$\sigma_j^{min} := \max\left\{\rho_j^{min}, A_1^{\min}\right\} \tag{3.7}$$

$$\sigma_j^{max} := \min\left\{\bar{\rho}_j^{hp} - 1, \max\left\{A_1^{\max}, \bar{\rho}_j^{any}\right\}\right\} \tag{3.8}$$

Proof. It is a direct combination of the bounds on σ_j shown in Corollaries 3.2 and 3.4. □

Corollary 3.6. *A potentially ready job $j \in \mathcal{R}$ may potentially be the next job dispatched by the scheduler in the system state abstracted by s only if $\sigma_j^{min} \leq \sigma_j^{max}$.*

Proof. Applying Lemma 3.5, if $\sigma_j^{min} > \sigma_j^{max}$ for a potentially ready job $j \in \mathcal{R}$, then the interval in which it may start is empty. Therefore, there is no execution scenario $\gamma \in \Gamma(s)$ such that j is the next job scheduled by the scheduler after the system state abstracted by s. □

3.3.3 Generating New Abstract States

Two cases must be considered for every potentially ready job $j \in \mathcal{R}$. Either $\sigma_j^{min} > \sigma_j^{max}$ or $\sigma_j^{min} \leq \sigma_j^{max}$. We discuss both in this section.

Case $\sigma_j^{min} > \sigma_j^{max}$. By Corollary 3.6, if $\sigma_j^{min} > \sigma_j^{max}$, then j cannot be the next job scheduled by the scheduler in the system state abstracted by s. Thus, there is no edge labelled with j outgoing from s that must be added to the SAG.

Case $\sigma_j^{min} \leq \sigma_j^{max}$. If $\sigma_j^{min} \leq \sigma_j^{max}$, then there may be an execution scenario $\gamma \in \Gamma(s)$ such that j is the next job scheduled by the scheduler. In such case, we can calculate bounds on the finishing time f_j of j in γ as follows.

The finishing time of j is $f_j \in [f_j^{\min}, f_j^{\max}]$, where:

$$f_j^{\min} := \sigma_j^{min} + C_j^{\min} \tag{3.9}$$

$$f_j^{\max} := \sigma_j^{max} + C_j^{\max} \tag{3.10}$$

That is, j finishes at the earliest when it starts the earliest and executes for its BCET, and it finishes the latest when it starts the latest and executes for its WCET (remember that jobs are non-pre-emptive and therefore run to completion once they start).

Since there may be an execution scenario $\gamma \in \Gamma(s)$ such that j is the next job scheduled, we generate a new abstract state $\tilde{s}$ abstracting the system state when j is scheduled after the system state abstracted by s. The new abstract state $\tilde{s}$ is then added to the SAG with an edge labelled with j and connecting s to $\tilde{s}$.

For convenience, we refer to the parameters of $\tilde{s}$ by $\tilde{\mathcal{D}}$, $\tilde{\chi}$, $\tilde{A} = ([\tilde{A}_i^{min}, \tilde{A}_i^{max}])_i$, and $\tilde{F}$. The rest of this section explains how to build $\tilde{\mathcal{D}}$, $\tilde{\chi}$, $\tilde{A} = ([\tilde{A}_i^{min}, \tilde{A}_i^{max}])_i$, and $\tilde{F}$.

Set of scheduled jobs Given that $\tilde{s}$ results from executing j after s, the set of dispatched jobs $\tilde{\mathcal{D}}$ is straightforwardly obtained by adding j to the set of scheduled jobs in s:

$$\tilde{\mathcal{D}} := \mathcal{D} \cup \{j\} \tag{3.11}$$

Set of finishing intervals The set of finishing intervals $\tilde{F}$ for jobs with a successor that is still to be scheduled in $\tilde{s}$ is obtained from the set of finishing intervals F in s as follows:

$$\begin{aligned} \tilde{F} := &\left\{ [f_{j'}^{\min}, f_{j'}^{\max}] \in F \mid succ_(j') \not\subseteq \tilde{\mathcal{D}} \right\} \\ &\cup \left\{ [f_j^{\min}, f_j^{\max}] \text{ if } succ_(j) \neq \emptyset \right\} \end{aligned} \tag{3.12}$$

Note that we remove the finishing time intervals of all jobs that have all their successors in $\tilde{\mathcal{D}}$ from the set of finishing time intervals $\tilde{F}$ since an abstract state $\tilde{s}$ should only remember finishing time intervals of finished jobs that have successors that have not been executed yet.

Set of certainly running jobs The set $\tilde{\chi}$ of certainly currently running jobs in $\tilde{s}$ is built from the set of certainly running jobs in s by:

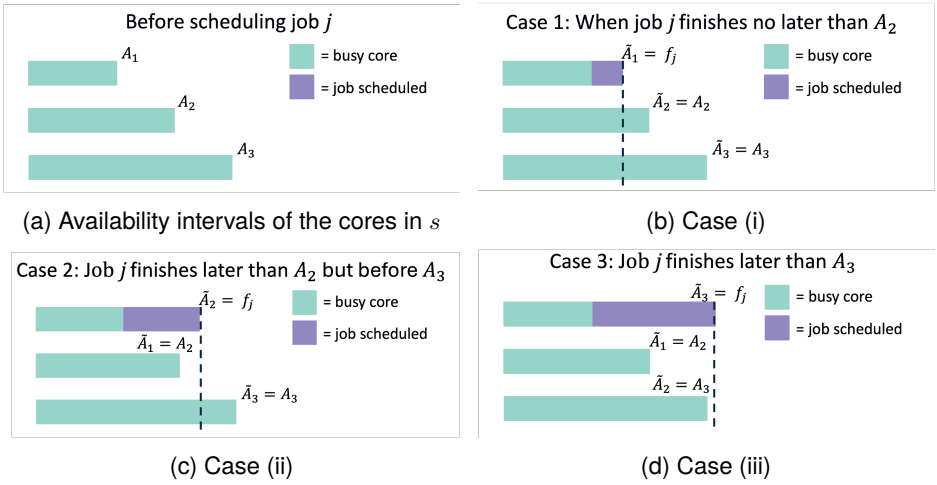
1. adding j to the set of certainly running jobs since it was just dispatched on a core when transitioning from s to $\tilde{s}$;

2. removing all predecessors of j from the set of certainly running jobs since predecessors of j must have finished their execution for j to start executing, and

3. removing all jobs that might be finished when j starts executing, but keeping those that are certainly not finished yet.

Hence, we get

$$\tilde{\chi} := \{j\} \cup \left\{ j' \in \chi \,\middle|\, j' \notin pred_j \wedge f_{j'}^{\min} > \sigma_j^{max} \right\}. \tag{3.13}$$



(a) Availability intervals of the cores in s

(b) Case (i)

(c) Case (ii)

(d) Case (iii)

Figure 3.3: Deriving the availability intervals of the cores in $\tilde{s}$.

Cores availability intervals To derive the availability intervals of the cores in the new system state abstracted by $\tilde{s}$, we start by making the following observations using the examples shown in Figure 3.3.

Observation. 1: Consider an execution scenario where i cores were available at time A_i and $i+1$ cores were available at time A_{i+1} before executing job j. Then, as shown in Figure 3.3, after dispatching j on a core, three cases may happen: (i) the job j finishes no later than A_i, (ii) j finishes later than A_i but no later than A_{i+1} or (iii) j finishes after A_{i+1}. If j finishes no later than A_i, then we will still have i cores becoming available at time A_i after j was dispatched (see the case of A_2 in Figure 3.3b). If j finishes later than A_i but no later than A_{i+1}, then we have i cores becoming available when j finishes its execution (see $\tilde{A}_2$ in Figure 3.3c). If j finishes later than A_{i+1}, then i cores become available at A_{i+1}, i.e., when $i+1$ cores were becoming available before dispatching job j (see $\tilde{A}_2$ in Figure 3.3d). Hence, the new availability time of i cores after dispatching j is given by $\tilde{A}_i$ computed as

$$\tilde{A}_i = \min\left\{\max\left\{f_j, A_i\right\}, A_{i+1}\right\}. \tag{3.14}$$

Observation. 2: Let A_m be the time at which m cores were becoming available before scheduling the job j. After dispatching job j on a core, the m cores will again all be available at j's finishing time if $f_j > A_m$ (see $\tilde{A}_3$ in Figure 3.3d) or at A_m otherwise (see $\tilde{A}_3$ in Figure 3.3c). Hence, $\tilde{A}_m$ is computed as

$$\tilde{A}_m = \max\left\{f_j, A_m\right\}. \tag{3.15}$$

These observations lead to the following lemma.

Lemma 3.7. *For all* $i = 1, \ldots, m-1$*,*

$$\tilde{A}_i \geq \min\left\{\max\left\{f_j^{\min}, A_i^{\min}\right\}, A_{i+1}^{\min}\right\} \tag{3.16}$$

$$\tilde{A}_i \leq \min\left\{\max\left\{f_j^{\max}, A_i^{\max}\right\}, A_{i+1}^{\max}\right\} \tag{3.17}$$

and

$$\tilde{A}_m \geq \max\left\{f_j^{\min}, A_m^{\min}\right\} \tag{3.18}$$

$$\tilde{A}_m \leq \max\left\{f_j^{\max}, A_m^{\max}\right\}. \tag{3.19}$$

Proof. The lower bound on $\tilde{A}_i$ is obtained from Equation (3.14) and (3.15) injecting the lower bound on f_j computed in Equation (3.9) and the lower bound on A_i known from the previous system state before scheduling job j as abstracted in s.

Similarly, the upper bound on $\tilde{A}_i$ is obtained from Equation (3.14) and (3.15) injecting the upper bound on f_j computed in Equation (3.10) and the upper bound on A_i known from the system state before scheduling job j as abstracted in s. □

The bounds of Lemma 3.7 would be already sufficient to define the availability intervals for abstract state $\tilde{s}$ abstracting the system state after scheduling job j. However, in the following two lemmas, we introduce two additional constraints on the core availability intervals to tighten the results of the analysis.

Lemma 3.8. *All availability intervals in* $\tilde{s}$ *are lower bounded by the start time of* j*. That is, for all* $i = 1, \ldots, m$*,* $\sigma_j^{min} \leq \tilde{A}_i$*.*

Proof. Let j' be the next job dispatched by the scheduler after j. The job j' can only start at or after j started to execute (otherwise, j' was dispatched before j, leading to a contradiction). Thus, the start time σ_j of j is a lower bound on the time at which any core becomes available to execute any job after j. □

Second, we make use of the set χ of certainly executing jobs in the system state before j starts executing to tighten the availability intervals after j's dispatch.

Lemma 3.9. *Let* $p := |\chi \cap pred_j|$ *be the number of predecessors of job* j *that were certainly running in the system state abstracted by* s*. At least* $p-1$ *cores must be available at time* σ_j^{max} *in* $\tilde{s}$*. That is, for all* $i < p$*,* $\tilde{A}_i \leq \sigma_j^{max}$*.*

Proof. By definition of χ, all jobs in χ are certainly running before j starts. Therefore, all the predecessors of j in χ, i.e., $\chi \cap pred_j$, are occupying a core before j starts. Since all predecessors of j must have finished their execution for j to start, all jobs in $\chi \cap pred_j$ must be finished when j starts, and the $p = |\chi \cap pred_j|$ cores occupied by predecessors of j must have become available when j starts. Since

j may start executing on at most one of those p cores, at least $p-1$ cores are certainly available when j starts executing. Since j starts at the latest at σ_j^{max}, it yields $\tilde{A}_i^{max} \leq \sigma_j^{max}$ for all $i < p$. □

Combining the bounds in Lemmas 3.7, 3.8 and 3.9, we can compute the new core availability intervals in $\tilde{s}$ as defined in Lemma 3.10 below.

Lemma 3.10. *Let $p := |\chi \cap pred_j|$. In any execution scenario $\gamma \in \Gamma(\tilde{s})$, we have $\tilde{A}_i \in [\tilde{A}_i^{min}, \tilde{A}_i^{max}]$ where*

$$\tilde{A}_i^{min} = \begin{cases} \max\left\{\sigma_j^{min}, \min\left\{\max\left\{f_j^{\min}, A_i^{\min}\right\}, A_{i+1}^{\min}\right\}\right\} & \text{if } i < m \\ \max\left\{f_j^{\min}, A_m^{\min}\right\} & \text{if } i = m \end{cases} \tag{3.20}$$

$$\tilde{A}_i^{max} = \begin{cases} \min\left\{\sigma_j^{max}, \max\left\{f_j^{\max}, A_i^{\max}\right\}, A_{i+1}^{\max}\right\} & \text{if } i < p \\ \min\left\{\max\left\{f_j^{\max}, A_i^{\max}\right\}, A_{i+1}^{\max}\right\} & \text{if } p \leq i < m \\ \max\left\{f_j^{\max}, A_m^{\max}\right\} & \text{if } i = m \end{cases} \tag{3.21}$$

Proof. It is a simple application of Lemmas 3.7, 3.8 and 3.9. □

3.4 Merging Phase

The goal of the merging phase is to control the growth of the SAG. The idea is that two abstract states in the SAG can be merged into a single abstract state if the system states they abstract are "similar enough". That is, given two abstract states $s_1 = (\mathcal{D}(s_1), \chi(s_1), A(s_1), F(s_1))$ and $s_2 = (\mathcal{D}(s_2), \chi(s_2), A(s_2), F(s_2))$ at the $(\ell+1)$-th layer of the SAG such that s_1 abstracts the execution scenarios $\Gamma(s_1)$ and s_2 abstracts $\Gamma(s_2)$, we can construct a merged abstract state s' abstracting $\Gamma(s') := \Gamma(s_1) \cup \Gamma(s_2)$. To that end, we define *merging criteria* and a merging procedure that replaces the two initial abstract states by one merged abstract state.

Merging criteria: The abstract states s_1 and s_2 are merged if all the following merging criteria are fulfilled:

1. The set of dispatched jobs is identical, i.e., we have $\mathcal{D}(s_1) = \mathcal{D}(s_2)$.
2. The availability intervals in both abstract states overlap or are contiguous. Specifically, this means that, $[A_i^{\min}(s_1), A_i^{\max}(s_1) + 1] \cap [A_i^{\min}(s_2), A_i^{\max}(s_2) + 1] \neq \emptyset$ for all $i = 1, \ldots, m$.
3. The finishing intervals of s_1 and s_2 of jobs with unfinished successors are contiguous or overlap, i.e.,

$[f_j^{\min}(s_1), f_j^{\max}(s_1)+1] \cap [f_j^{\min}(s_2), f_j^{\max}(s_2)+1] \neq \emptyset$ for every job $j \in \mathcal{D}$ with a successor that was not dispatched yet.

Merging procedure: Given two abstract states s_1 and s_2 fulfilling the merging criteria, s_1 and s_2 are replaced by a single abstract state s' abstracting the information of both s_1 and s_2.

More specifically, the merged abstract state s' is defined as follows:

- The set of finished jobs in s' is equal to the set of finished jobs in s_1 and s_2, i.e.,
$$\mathcal{D}(s') := \mathcal{D}(s_1) = \mathcal{D}(s_2). \tag{3.22}$$
- The set of certainly currently executing jobs in s' is made of the jobs that are certainly executing in both s_1 and s_2, i.e.,
$$\chi(s') := \chi(s_1) \cap \chi(s_2). \tag{3.23}$$
- The availability intervals of s' over-approximate the availability intervals of s_1 and s_2. That is, for all $i = 1, \ldots, m$,
$$A_i^{\min}(s') = \min\left\{A_i^{\min}(s_1), A_i^{\min}(s_2)\right\} \tag{3.24}$$
$$A_i^{\max}(s') = \max\left\{A_i^{\max}(s_1), A_i^{\max}(s_2)\right\}. \tag{3.25}$$
- Similarly, the finishing intervals are defined to over-approximate the finishing intervals of both abstract states. That is, for every $j \in \mathcal{D}$ that has at least one successor that has not yet been dispatched,
$$f_j^{\min}(s') = \min\left\{f_j^{\min}(s_1), f_j^{\min}(s_2)\right\} \tag{3.26}$$
$$f_j^{\max}(s') = \max\left\{f_j^{\max}(s_1), f_j^{\max}(s_2)\right\}. \tag{3.27}$$

The merged abstract state s' replaces the two original abstract states s_1 and s_2, and all vertices to s_1 or s_2 are replaced by vertices to s'.

EXAMPLE 3.3

In the SAG shown in Figure 3.2, we see that two abstract states, created by dispatching job j_4 after abstract state s_2 and job j_2 after abstract state s_3, have been merged into a single abstract state s_5. To clarify how the merge occurred, let us consider the non-merged versions of these abstract states. We have s_2 that becomes $\tilde{s}_2$ after dispatching j_4, and s_3 that becomes $\tilde{s}_3$ after dispatching j_2.

For $\tilde{s_2}$, the availability interval $A_1(\tilde{s_1})$ of the single core is $[7, 10]$ and for $\tilde{s_3}$,

the availability interval $\tilde{A}_1(\tilde{s_3})$ is $[6, 9]$. Furthermore, for both states, $\tilde{F}(\tilde{s_2})$ and $\tilde{F}(\tilde{s_3})$ contains only the finish time interval of j_2 (i.e., the only scheduled job with a successor that was not dispatched yet). The finish time interval of j_2 is $[6, 9]$ in $\tilde{s_2}$ and is $[7, 10]$ in $\tilde{s_3}$. Finally, the set χ of certainly running job contains only j_4 in $\tilde{s_2}$ and only j_2 in $\tilde{s_3}$.
The abstract states $\tilde{s_2}$ and $\tilde{s_3}$ meet all the merging criteria. Hence, the abstract states are merged to form the abstract state s_5 with an availability interval $A_1 = [6, 10]$ that spans both the availability interval $\tilde{A}_1(\tilde{s_2})$ and $\tilde{A}_1(\tilde{s_3})$, a finish time interval of j_2 equal to $[6, 10]$, and with an empty set of certainly running jobs since the initial abstract states $\tilde{s_2}$ and $\tilde{s_3}$ disagree on which jobs are certainly running in their respective execution scenarios.

3.5 Correctness

In this section, we prove that the SAG analyses all possible execution scenarios for $\mathcal{J}$ and that it derives safe bounds on the response time of every job in each of those execution scenarios.

Theorem 3.11. *The SAG analyses all reachable system states in all possible execution scenarios in* Γ.

Proof. We make a proof by induction that all system states reachable in an execution scenarios in Γ is in the SAG. That is, we prove $\bigcup_{s \in L_\ell} \Gamma(s) = \Gamma$ for all $\ell = 0, 1, \ldots$.

Start case ($\ell = 0$). In all execution scenarios the initial system state must be that all cores are free and no jobs has been dispatched yet. That states is abstracted by the initial abstract state s_0 in the SAG. Furthermore, it yields $\bigcup_{s \in L_0} \Gamma(s) = \Gamma$ where L_0 is the set of abstract states in layer 0 of the SAG.

Induction step ($\ell \mapsto \ell+1$). If $\bigcup_{s \in L_\ell} \Gamma(s) = \Gamma$ at the ℓ-th layer of the SAG, then we prove that layer $\ell+1$ contains all system states reachable from the system states recorded in the ℓ-th layer, and $\bigcup_{s \in L_{\ell+1}} \Gamma(s) = \Gamma$. Since, for every abstract state s in the ℓ-th layer, Algorithm 2 adds a new system state in layer $\ell + 1$ for every possible scheduling decision in all execution scenarios covered by s (Lines 10 to 27), we have that layer $\ell + 1$ contains all system states reachable from s, and we have that $\bigcup_{\tilde{s} \in L_{\ell+1}} \Gamma(\tilde{s}) \supseteq \Gamma(s)$ for all $s \in L_\ell$, which yields $\bigcup_{\tilde{s} \in L_{\ell+1}} \Gamma(\tilde{s}) \supseteq \bigcup_{s \in L_\ell} \Gamma(s) = \Gamma$ by the induction hypothesis. This proves our induction claim.

Conclusion. Since, by the induction step, every layer contains all system states reachable from the previous layer in all execution scenarios, all system states of all execution scenarios are eventually analysed in the SAG. □

Corollary 3.12. *Algorithm 2 returns safe bounds on the* BCRT *and* WCRT *of every job in* $\mathcal{J}$*.*

Proof. By Theorem 3.11, all reachable system states are part of the SAG. Since Algorithm 2 computes a lower and upper bound on the finish time of each job scheduled in every system state in the SAG (Line 13), and because it returns the lowest and largest relative finish time it calculated in any system state for every job (Lines 14 and 15), the relative finish times returned for every job are real lower and upper bounds on the BCRT and WCRT of every job. □

3.6 Evaluation

Since our analysis is the first for limited-pre-emptive EDD tasks, there is no prior work we can compare our analysis against, as we are now setting the benchmark. Therefore, instead, we evaluate our analysis against the state of the art for self-suspending [49, 18] and DAG tasks [49]. In the following, we refer to [65] as **Yalcinkaya**, [18] as **Casini**, and [49] as **Nasri**.

We answer the following questions. How does our analysis compare against the state of the art response time analysis for limited-pre-emptive self-suspending tasks[1] and DAG tasks? What is the accuracy of our analysis in terms of schedulability? How well does it scale with the number of jobs and cores?

Experimental setup. We implemented our analysis in C++ using [55] as a basis but implementing a completely new analysis core that optimizes runtime. For each experiment, we generate n periodic tasks with implicit-deadline and without offset ($n \in \{4, 5, 8, 10, 12\}$) executed on m processors ($m \in \{1, 2, 4, 6\}$). Tasks can be of two types: DAG or segmented self-suspending. To generate segmented self-suspending tasks, we randomly generate n "artificial" utilization values U'_τ that sum to a target total utilization U' for the n tasks $\tau \in \mathcal{T}$ using UUnifast [9]. We then randomly draw a period T_τ for each task from a logarithmic distribution over the interval $[10, 10000]$ as suggested in [40]. We set the total WCET and the total worst-case suspension time of each task τ to $C_\tau = \beta \times U'_\tau \times T_\tau$ and $susp_\tau = (1 - \beta) \times U'_\tau \times T_\tau$, respectively, where β is a parameter that defines the ratio of execution versus suspension. We then divide the total WCET between S execution segments, and the worst-case suspension time between $S - 1$ suspension intervals using the randfixedsum algorithm [24], where S is a parameter of the experiment. The best-case execution time and best-case suspension time of each segment is set to 50% of their worst-case counterpart.

[1] We compare on single core only since no analysis for multicore existed until our work.

Tasks are assigned priorities according to deadline monotonic. To increase the chance that the generated task sets are schedulable, we also limit the WCET of any execution segment to be at most half the smallest task period.

Each experiment is run on a single core of an Intel Xeon w7-3465x processor with peak frequency of 3.2GHz and equipped with 251GB of RAM.

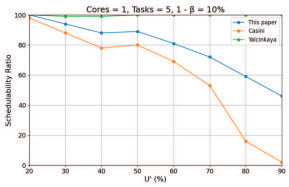
Single-core scheduling. Since Casini can only analyse single core scheduling we start with an experiment for self-suspending task scheduling on single core. Figure 3.4a shows the number of task sets that were deemed schedulable by Yalcinkaya, Casini and our analysis out of 100 randomly generated task set per utilization value U'. Yalcinkaya is an exact schedulability analysis and it deems almost all task sets as being schedulable. Yalcinkaya, however, is not a response time analysis. Therefore, contrary to Casini and our analysis, it cannot provide response time bounds. However, Yalcinkaya's exactness allows us to check the real accuracy of Casini and our analysis in terms of schedulability.

Our analysis is able to detect around 90% of the schedulable task sets up to $U' = 50\%$ but the accuracy steadily decreases until around 50% at $U' = 90\%$. It is still a significant gain in comparison to Casini, which can detect almost no schedulable task set at $U' = 90\%$.

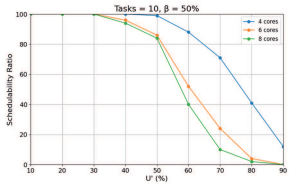
Runtime. Besides the fact that Yalcinkaya does not return response time bounds, the test [65] is known to suffer from scalability issues. Therefore, we test how our new analysis fares in terms of runtime against Yalcinkaya in a still rather simple setting by considering 5 tasks comprising 5 execution segments executed on two cores. Figure 3.4f shows the average runtime and the number of experiments that timed out after 1 hour for each value of U'. None of the tests timed out with our analysis. In comparison, between 23 and 50% of the tests timed out with Yalcinkaya for almost every utilization value. The runtime of Yalcinkaya is also 4-6 orders of magnitudes larger than that of our analysis in average. It shows that our analysis scales much better and is thus more practical than the analysis proposed in [65] to analyse multicore systems. We also remind one more time that our analysis calculates more information than [65] as it also calculates best-case and worst-case response time bounds for every job where [65] only returns a boolean value for the schedulability.

To be noted that Foughali et al. [26] recently published an UPPAAL-based test that scales better than Yalcinkaya. Yet, the test in [26] can only analyse single core platforms and does not support self-suspensions. Still, to benchmark our analysis, we ran Foughali et al. [26] on the AUTOSAR use case[2] available on

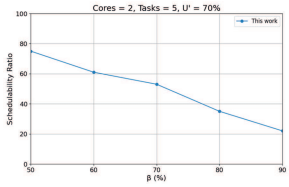
[2]The use case is based on the WATERS 2017 industrial challenge [32].



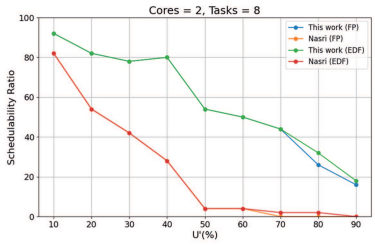
(a) Schedulability ratio on single core.



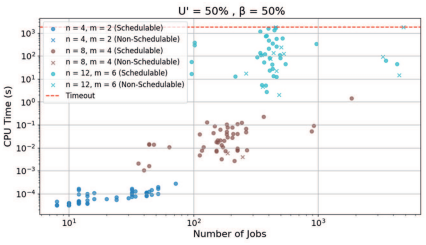
(b) Schedulability ratio on 4, 6 and 8 cores.



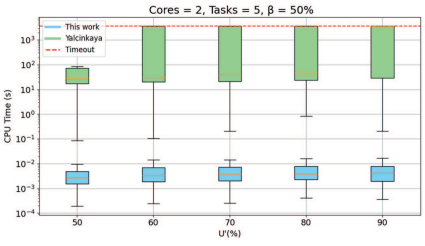
(c) Influence of the parameter β.



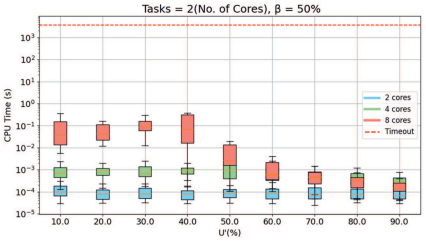
(d) Schedulability ratio for DAG tasks.



(e) Runtime of our analysis.



(f) Average runtime and timeouts.



(g) Runtime of our analysis for systems with 2, 4 and 8 cores.

Figure 3.4: Experimental results

their gitlab[3]. Foughali's test requires 125 seconds to check the schedulability of seven limited-pre-emptive tasks made of 710 segments executed on a single core. According to our tests, it also requires between 125 and 165 seconds to calculate the worst-case response time bound for a task. In comparison, the test of Nasri available on GitHub [55] takes 244 seconds to calculate the exact worst-case but also the best-case response times for every task as well as every job released in a hyper-period. Our new analysis solves the same problem and provides exact response time bound *in only 4 seconds*, thus showing almost two orders of magnitude improvement against Nasri and a 30X speed-up in comparison to [4].

Thanks to those performances, our analysis could easily analyse systems of 8 cores running 10 tasks comprising 5 execution segments each with no time

[3]We contacted the authors of [26] who provided us an optimized version of the `water_2.xta` file generated with the `parse_sol.py` script available on [4]. We ran the test using UPPAAL5.0 on an i7-8750H processor @2.20GHz with 32GB of RAM. Following the recommendation of the authors, we activate the Difference Bound Matrix (DBM) option for the state space representation to accelerate the analysis.

outs. The result of that experiment is shown in Figure 3.4b. Again, we see that our analysis can identify a large number of schedulable task sets even at high utilization values.

In Figure 3.4e, we provide the runtime of our analysis for various configurations of cores and tasks. This figure illustrates the impact of the number of jobs and cores on the runtime of the analysis. We observe that our approach can analyse the large majority of systems with up to 12 tasks on 6 cores under 2 minutes and some larger systems take up to an hour. This demonstrates the scalability of our analysis, especially as the number of cores and tasks increases.

In Figure 3.4g, we illustrate the impact of the number of cores and number of tasks on the runtime of our analysis for varying utilisation values. The number of tasks was set to be twice the number of cores. We observe that the runtime of our analysis scales with the number of cores and tasks, but decreases as utilization values increase. This is because, with higher utilization, fewer schedulable task sets are found, meaning that the analysis does not need to explore the entire state space. Importantly, these results demonstrate that our analysis scales well with increasing number of cores and tasks, effectively handling the analysis of larger systems within the timeout limit.

Impact of β. We tested how the parameter β impacts schedulability. Figure 3.4c shows that, unsurprisingly, schedulability decreases when the ratio β that controls the length of the execution segments versus the suspension ones increases. Since execution segments become longer, the blocking they may generate due to their non-pre-emptivity increases, making the schedulability drop.

Improvement of DAG analysis. We evaluated the improvement offered by our new version of the SAG analysis over the previous SAG analysis for DAG tasks, as presented in [49]. The DAG tasks were generated using the method described in [18, 44], with the probability of a node forking set to 0.6, a maximum of 2 nested forks, and up to 3 parallel segments originating from a fork. For this experiment, we generated 50 random task sets per utilization, for both Fixed Priority (FP) and Earliest Deadline First (EDF) scheduling policies, which was sufficient to demonstrate the improvement. The results, shown in Figure 3.4d, indicate that the new system state abstraction used in our work significantly enhances analysis accuracy in comparison to Nasri, which was the best existing test for limited-pre-emptive DAG tasks until now. Our analysis was able to detect 2.5 times more schedulable task sets for both FP and EDF scheduling policies than Nasri. Notably, this significant improvement did not come at the cost of increased runtime, on the contrary. Our analysis takes between 0.0001 and 2.22 seconds to analyse a task set with an average runtime of 0.36 seconds. In comparison,

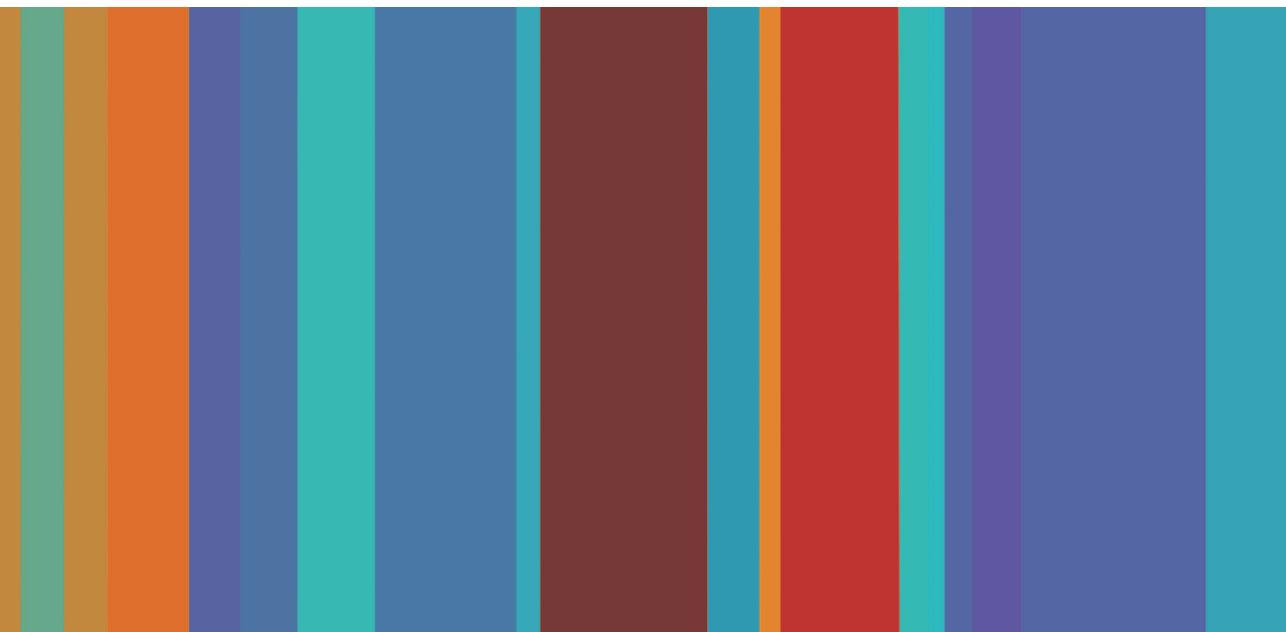
Nasri takes between 0.0013 and 139.68 seconds with an average of 4.42 seconds. Therefore, our analysis demonstrates a runtime improvement over Nasri of one order of magnitude in average and up to two orders of magnitudes in some cases. This is thanks to our new implementation of the SAG analysis core, which was rethought to optimize runtime.

3.7 Conclusion

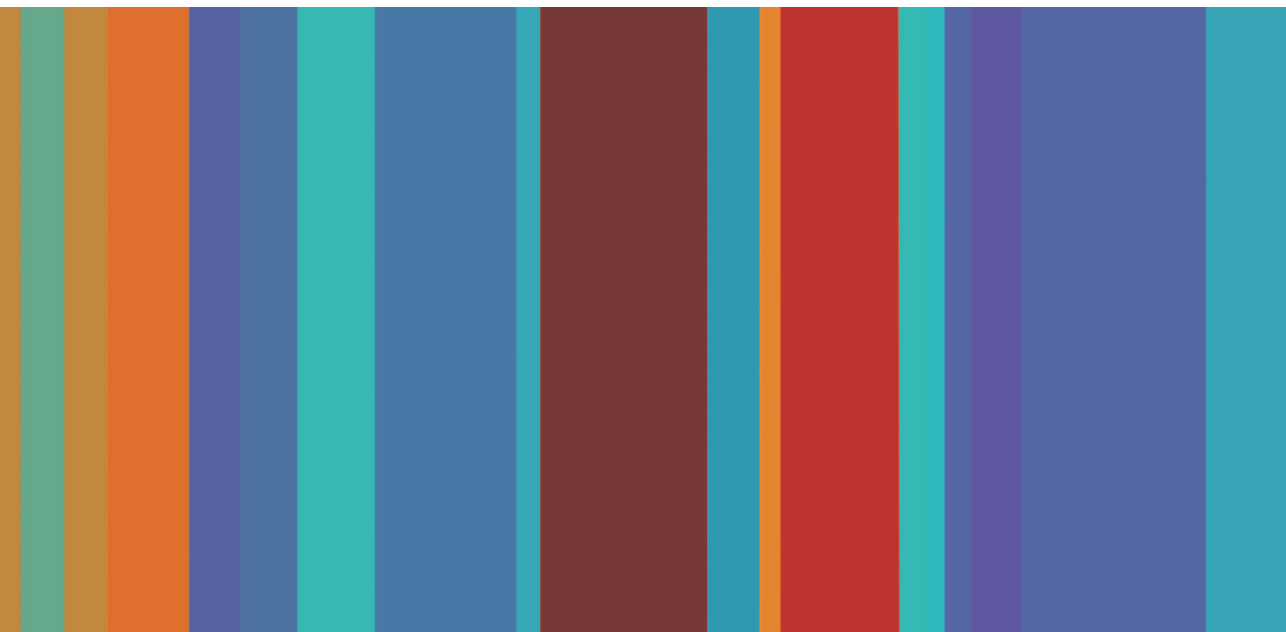
In this chapter, we proposed the first response time analysis for limited-pre-emptive EDD tasks executed by a job-level fixed priority global scheduling algorithm on a single or multicore platform. The analysis extends the SAG approach to account for suspension times, proposing an updated SAG generation algorithm and a new system state abstraction.

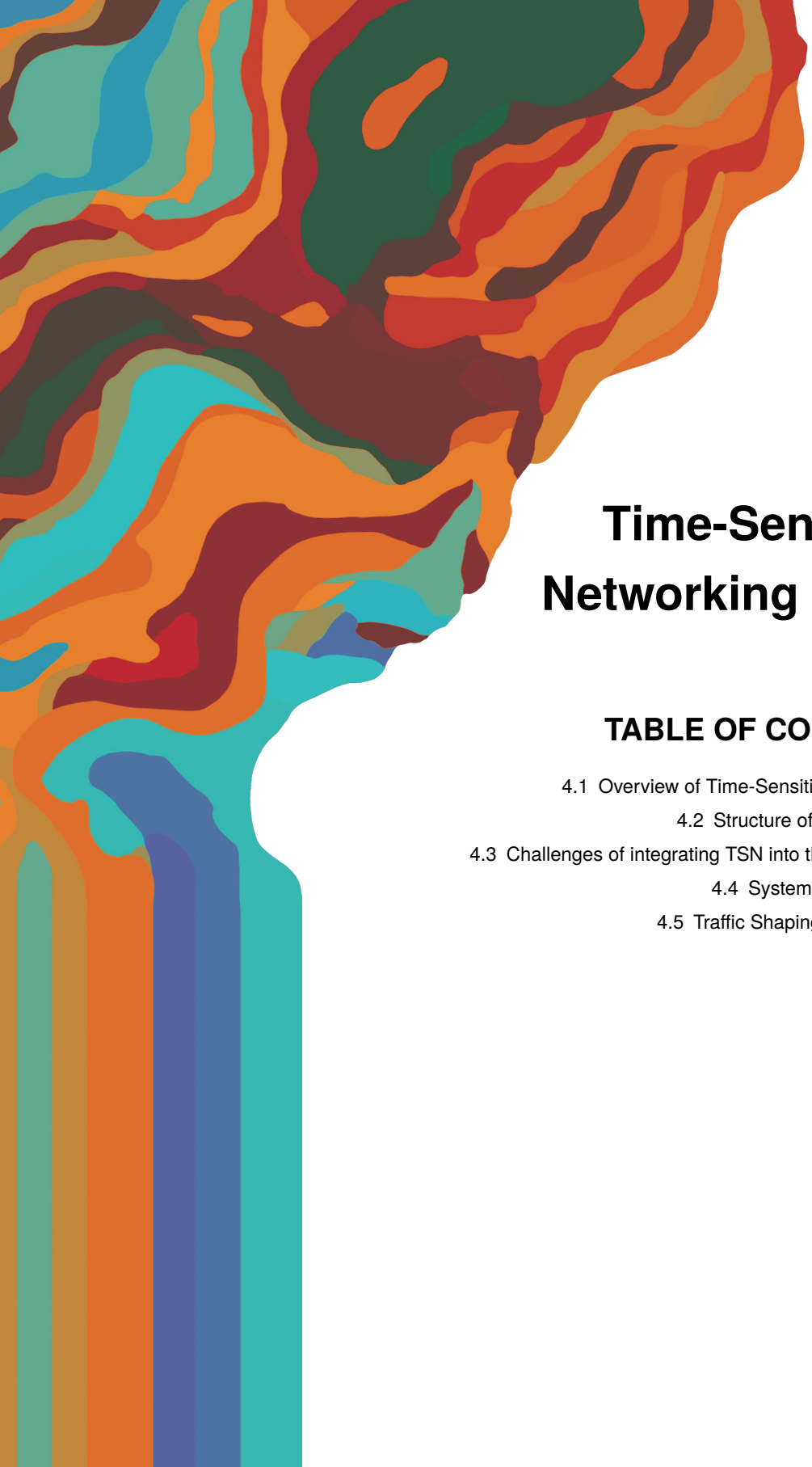
The experimental results show that, on randomly generated task sets, our analysis outperforms the only other existing response time analysis for limited-pre-emptive self-suspending tasks. It outperforms the state of the art analysis for limited-pre-emptive DAG tasks globally scheduled on a multicore platform. It also scales to the analysis of much larger systems than the exact schedulability test for self-suspending tasks by [65] or limited-pre-emptive tasks by [26] as shown on a realistic automotive case study derived from the WATERS17 industrial challenge [32].

PART II



Extending the Schedule Abstraction Graph to Time-Sensitive Networking





4

Time-Sensitive Networking (TSN)

TABLE OF CONTENTS

4.1 Overview of Time-Sensitive Networking

4.2 Structure of a TSN Switch

4.3 Challenges of integrating TSN into the SAG model

4.4 System Model of TSN

4.5 Traffic Shaping Mechanisms

Modern real-time systems increasingly consist of multiple distributed components that must exchange information over a communication network. In many industrial applications, such as automotive systems, robotics platforms, and factory automation environments, the sensors, controllers, and actuators operate as separate computational units connected through a network infrastructure. While each component performs local computations, the overall system behaviour depends on the timely exchange of data between these components.

In such distributed real-time systems, correctness is therefore determined not only by the functional behaviour of individual components but also by their timing behaviour, which was the focus of Chapters 2 and 3, as well as by the timing of the communication between them. Messages carrying sensor readings, control commands, or synchronization signals must arrive within well-defined timing bounds. If these messages are delivered too late, the receiving component may operate on outdated information, potentially leading to degraded performance or unsafe system behaviour.

Conventional Ethernet networks were originally designed to maximise throughput and flexibility rather than to provide predictable timing guarantees. As a result, packet transmission delays in conventional Ethernet networks can vary significantly depending on network load, contention between packets, and buffering within switches. While such variability may be acceptable in general-purpose communication networks, it is problematic for real-time systems that rely on strict timing guarantees.

To address these limitations, a set of standards collectively known as Time-Sensitive Networking (TSN) [37] has been developed by the Institute of Electrical and Electronics Engineers (IEEE) 802.1 working group. TSN extends standard Ethernet by introducing mechanisms that enable deterministic communication over switched Ethernet networks. These mechanisms provide stronger guarantees on packet transmission timing by regulating when packets may be transmitted and how they compete for network resources.

TSN introduces several techniques that help control packet delays and reduce timing variability. These include precise network-wide time synchronization, traffic prioritization, and traffic shaping mechanisms that regulate packet transmissions in order to prevent uncontrolled bursts and interference between traffic flows. By combining these mechanisms, TSN networks are able to provide bounded communication delays while maintaining compatibility with standard Ethernet infrastructure.

The introduction of such mechanisms fundamentally changes how communication networks behave in real-time systems. Packet transmissions are no longer governed solely by best-effort forwarding rules, but instead follow structured scheduling and shaping policies that determine when packets are eligible to be

transmitted on a link. In this sense, packet transmissions in TSN networks can be viewed as a scheduling problem that shares several similarities with the processor scheduling problems discussed earlier in this thesis.

Understanding the behaviour of these mechanisms is therefore essential when analysing the timing properties of distributed real-time systems that rely on TSN communication. In the remainder of this chapter, the key principles of Time-Sensitive Networking are introduced. The discussion begins with an overview of the TSN standardisation effort and its core mechanisms. This is followed by an examination of the internal structure of TSN switches and the traffic shaping mechanisms that regulate packet transmissions. These concepts form the foundation for the analysis techniques presented in the following chapters.

4.1 Overview of Time-Sensitive Networking

Time-Sensitive Networking (TSN) is a collection of standards developed by the IEEE 802.1 working group to enable deterministic communication over Ethernet networks. Rather than introducing a completely new networking technology, TSN extends standard Ethernet by adding mechanisms that control how packets are transmitted through the network. The goal of these extensions is to provide predictable communication delays while preserving compatibility with existing Ethernet infrastructure.

The development of TSN was motivated by the increasing use of Ethernet in cyber-physical systems. Traditionally, specialized field bus technologies were used in industrial environments to provide reliable and predictable communication between controllers, sensors, and actuators. However, these technologies often lacked interoperability and scalability. Ethernet, on the other hand, offers high bandwidth, flexibility, and widespread adoption, making it an attractive candidate for industrial networking. The challenge lies in adapting Ethernet so that it can satisfy the strict timing requirements of real-time systems.

To achieve deterministic behaviour, TSN introduces several mechanisms that regulate how packets are transmitted and how network resources are shared between traffic flows. These mechanisms address different aspects of communication timing, including synchronization of network devices, prioritization of traffic, and shaping of packet transmissions.

One fundamental component of TSN is precise time synchronization across the network. The IEEE 802.1AS [34] standard defines a mechanism that allows all network devices to share a common notion of time with sub-microsecond accuracy. This global time reference enables coordinated scheduling of packet transmissions across multiple switches and links, which is essential for certain

TSN mechanisms such as time-triggered transmission.

Another key aspect of TSN is the ability to differentiate between multiple classes of network traffic. Packets may belong to different traffic classes depending on their priority and timing requirements. Each class is typically associated with a separate transmission queue in the network switches. Higher-priority traffic can be given preferential access to the network in order to ensure that time-critical packets experience minimal delay.

In addition to prioritization, TSN introduces traffic shaping mechanisms that regulate the rate and timing of packet transmissions. Traffic shaping prevents uncontrolled bursts of traffic from overwhelming the network and ensures that packets are transmitted in a controlled and predictable manner.

Two shaping mechanisms are particularly relevant in TSN. The first is the Time-Aware Shaper (TAS), standardized in IEEE 802.1Qbv [36]. TAS controls packet transmissions through time-triggered gate schedules, allowing selected traffic classes to transmit only during predefined time windows. The second is the Credit-Based Shaper (CBS), standardized in IEEE 802.1Qav [35]. CBS regulates the transmission opportunities of traffic classes through a credit mechanism, thereby smoothing bursts and controlling bandwidth sharing. Both mechanisms play an important role in determining packet eligibility for transmission and are discussed in detail later in this chapter.

In addition to these shaping mechanisms, TSN includes other standards that support resource reservation and configuration of traffic streams. For example, IEEE 802.1Qcc extends earlier stream reservation protocols by enabling centralised configuration of network resources. This allows network controllers to allocate bandwidth and configure scheduling parameters for specific traffic flows.

Together, these standards form a comprehensive framework for building deterministic Ethernet networks. By combining time synchronisation, traffic prioritisation, and shaping mechanisms, TSN networks can provide bounded latency and reduced delay variability for time-critical communication flows.

4.2 Structure of a TSN Switch

In order to understand how packet transmissions are scheduled in a Time-Sensitive Networking (TSN) system, it is necessary to examine the internal structure of a network switch. The behaviour of packets inside a switch directly determines how contention between packets is resolved and how transmission delays arise within the network.

A network switch connects multiple incoming and outgoing communication links and forwards packets from their source towards their intended destination.

When a packet arrives at a switch, it first passes through an ingress processing stage, where the switch determines the appropriate output port based on the packet's destination address and forwarding tables maintained by the switch.

After the output port has been determined, the packet is placed into one of several transmission queues associated with that output port. Modern Ethernet switches typically maintain multiple queues per output port, each corresponding to a different traffic class or priority level. Packets are classified and assigned to these queues according to their priority or traffic type. We can see this illustrated in the Figure 4.1 which shows the architecture of the switch.

In TSN-enabled switches, each queue is associated with a gate that determines whether packets in that queue are allowed to participate in transmission decisions. At any given time, a gate can be in one of two states:

- Open, allowing packets in the queue to be considered for transmission.
- Closed, preventing packets in the queue from being transmitted even if packets are waiting.

These gates introduce an additional level of control over packet transmissions. Instead of selecting packets directly from the highest-priority queue with pending packets, the switch first evaluates the state of the gates. Only packets belonging to queues whose gates are currently open are eligible to compete for transmission on the output link.

The state of the gates is controlled by the traffic shaping mechanisms implemented in the switch. These mechanisms determine when gates should open or close in order to regulate the flow of packets belonging to different traffic classes.

Once the eligibility of packets has been determined by the gate states, the switch applies its priority-based transmission logic. Among the packets that belong to queues with open gates, the switch selects the packet from the highest-priority queue for transmission on the output link.

After a packet is selected for transmission, it occupies the output link for the duration of its transmission time. During this time, the link cannot transmit another packet. Once the transmission completes, the switch again evaluates the queues, gate states, and priorities in order to determine the next packet to transmit.

The combination of queueing, gate control, and priority-based selection determines the order in which packets are transmitted from the switch.

Structural Correspondence to Task Scheduling In Part-I, the focus was on scheduling behaviour in computational systems, where jobs compete for access to processing resources. In communication networks, an analogous form of resource contention arises, where packets compete for access to a network link.

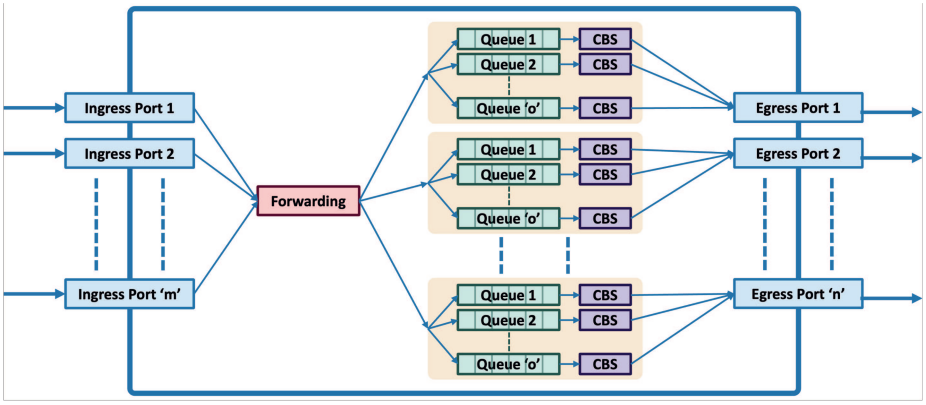


Figure 4.1: Switch architecture with m ingress ports, n egress ports, and o queues per egress port (one per traffic class).

This makes it possible to model each egress port as a single processor on which non-pre-emptive tasks are scheduled. In this view, a communication flow corresponds to a task that generates packets, and the execution-time bounds of the task correspond to transmission-time bounds derived from the variation in packet lengths within the flow. This modelling correspondence makes it possible to interpret packet transmissions using scheduling concepts developed for computation. However, this does not make the problem trivial, since packet transmissions in TSN networks are also affected by additional mechanisms that must be modelled explicitly, including FIFO queues and traffic shaping mechanisms that regulate when packets become eligible for transmission. Table 4.1 summarises this modelling correspondence.

Processor Scheduling	**TSN Communication**
Tasks	Communication Flows
Jobs	Packets
Processor	Egress Port
Scheduling decision	Packet selection for transmission
Job Execution time	Packet Transmission time
Scheduling policy	Switch transmission policy

Table 4.1: Structural correspondence between processor scheduling and packet transmission in TSN networks

4.3 Challenges of integrating TSN into the SAG model

The structural correspondence between processor scheduling and packet transmission provides a useful starting point for extending the scheduling abstraction developed in Chapter 2 to TSN communication. However, this integration is not trivial. Although packets may be modelled as jobs and network links as processors, TSN adds further mechanisms that have not been considered in the SAG analysis conducted so far.

A first challenge comes from the internal structure of a switch. In the processor scheduling approaches used for the SAG up to now, jobs usually contend directly with one another for processor access. In contrast, packets arriving at a TSN switch are first placed into multiple queues, each associated with a traffic class. As a result, transmission decisions depend not only on arrival times and priorities, but also on queue membership and FIFO ordering within each queue. A packet can only be considered for transmission when it reaches the head of its queue, which introduces an additional constraint compared to standard scheduling models.

A second challenge is the presence of traffic shaping mechanisms. In the computational setting, job eligibility is usually determined by release times and precedence constraints. In TSN, packet eligibility may also depend on the state of the shaping mechanism. Under TAS, a packet can only be transmitted when the gate of its queue is open according to a predefined schedule. Under CBS, transmission depends on the value of a credit variable that evolves over time. This means that eligibility is no longer determined only by arrival of a packet and availability of the link, but also by additional state variables that must be modelled explicitly.

Another challenge is that modelling an egress port as a single processor and packet transmissions as non-pre-emptive task executions applies most naturally at the level of a single switch. In practice, however, packets travel through multiple switches before reaching their destination. The delay experienced at one switch affects the arrival behaviour at the next. Therefore, extending this scheduling abstraction to TSN requires reasoning not only about a single scheduling point, but about a sequence of dependent scheduling decisions across the network.

Therefore, the SAG analysis must be extended to capture queue structures, shaping-based eligibility conditions, and the interaction between multiple switches along a packet's path. These aspects make the integration of TSN into the SAG framework more challenging.

4.4 System Model of TSN

The model in this Chapter captures packet transmission at a single switch egress port, providing the foundation for analysing end-to-end timing in TSN networks. It formally describes the packets generated by communication flows and the rules that govern their transmission at the switch.

4.4.1 Communication Flows and Packet Model

A TSN-based communication system carries a number of recurring communication activities over time. Each such activity represents a stream of data that must be transmitted between communicating devices in the network. In general, communication may occur irregularly or sporadically. However, in this thesis, the focus is on communication activities that repeat in a regular manner.

These recurring communication activities are modelled as *flows*. For example, a sensor may periodically transmit measurement data to a controller, or a controller may periodically send commands to an actuator. Each such recurring communication stream is represented as a flow in the system.

The communication system is therefore described as a finite set of flows,

$$\mathcal{F} = \{F_1, F_2, \ldots, F_{|\mathcal{F}|}\}.$$

Each flow has a number of timing and transmission parameters associated with it. When analysing the timing behaviour of network communication, it is necessary to know how regularly a flow injects packets, when its first packet may be released, how much variation there may be in packet arrivals at a switch, how long its packets may take to transmit, what timing requirement applies to the flow, and what priority is assigned to its packets. To capture this information precisely, a flow F is defined by the tuple

$$F = \left(T_F,\ O_F,\ J_F,\ l_F^{\min},\ l_F^{\max},\ D_F,\ \pi_F\right)$$

where:

- $T_F \in \mathbb{R}_{>0}$ denotes the period of the flow.
- $O_F \in \mathbb{R}_{\geq 0}$ denotes the offset of the flow, defining the earliest possible release time of its first packet.
- $J_F \in \mathbb{R}_{\geq 0}$ denotes the arrival jitter of the flow, capturing variability in the periodicity of packet arrivals.

- $l_F^{\min}, l_F^{\max} \in \mathbb{R}_{>0}$ denote the minimum and maximum length of packets generated by the flow (in bytes).
- $D_F \in \mathbb{R}_{>0}$ denotes the relative end-to-end deadline of the flow.
- $\pi_F \in \mathbb{Z}_{\geq 0}$, when defined, denotes the priority of the flow, where lower numeric values correspond to higher priorities. This parameter is only required under TLFP scheduling, and is omitted for JLFP scheduling.

A flow describes a recurring communication activity. However, the flow itself is not the entity that is directly transmitted on a link. Each individual occurrence of a flow gives rise to a concrete communication unit. Such an occurrence is called a *packet*.

In other words, every time a flow releases data, that release corresponds to one packet. If a flow repeats periodically, it generates a sequence of packets over time. Each of these packets represents one specific transmission instance of the flow.

Over a given observation window, the flows in $\mathcal{F}$ generate a collection of packets. The set of all such packets is denoted by $\mathcal{P}$.

Each packet $p_{i,j} \in \mathcal{P}$ is an indivisible unit of transmission and is associated with exactly one flow. While a flow describes the general communication pattern, a packet represents one concrete transmission that must be scheduled on the network links along its route.

For a flow $F_i \in \mathcal{F}$, the j-th packet generated by F_i, denoted $p_{i,j}$, has its arrival time at the switch bounded by

$$a_{p_{i,j}}^{\min} = O_{F_i} + j \cdot T_{F_i},$$

$$a_{p_{i,j}}^{\max} = O_{F_i} + j \cdot T_{F_i} + J_{F_i}.$$

These bounds define the interval within which the packet may arrive at the considered switch. They capture possible variability caused, for example, by forwarding and queueing delays in upstream switches.

Let R^{tx} denote the transmission rate of the considered egress port. Since $[l_{F_i}^{\min}, l_{F_i}^{\max}]$ denotes the bounds on the packet lengths of flow F_i, the transmission duration bounds of a packet $p_{i,j}$ of that flow are given by

$$C_{p_{i,j}}^{\min} = \frac{l_{F_i}^{\min}}{R^{\text{tx}}}, \qquad C_{p_{i,j}}^{\max} = \frac{l_{F_i}^{\max}}{R^{\text{tx}}}.$$

Each packet has an absolute deadline defined by

$$d_{p_{i,j}} = O_{F_i} + j \cdot T_{F_i} + D_{F_i}.$$

This deadline is determined by the periodic release pattern of the flow and the relative end-to-end deadline assigned to that flow.

Finally, each packet inherits its priority from the corresponding flow:

$$\pi_{p_{i,j}} = \pi_{F_i}.$$

Packets are the fundamental transmission entities considered in our analysis. At any given time, an egress port can transmit at most one packet on its outgoing link. Similarly, a packet can be transmitted on at most one link at a time.

4.4.2 Queueing Model and Packet Eligibility

Packets arriving at a switch are first classified and directed to the appropriate egress port according to the forwarding logic of the switch. Each egress port maintains multiple transmission queues, typically corresponding to different traffic classes or priority levels. Packets are assigned to these queues according to their priority $\pi_{p_{i,j}}$.

Within each queue, packets are ordered according to a first-in-first-out (FIFO) discipline. As a result, packets can only be transmitted in the order in which they arrive at the queue. In particular, a packet can only be considered for transmission when it reaches the head of its queue.

As already mentioned, in addition to the queueing order, each queue is associated with a *gate* that determines whether packets from that queue are currently allowed to participate in transmission decisions. At any given time, a gate may be open or closed. When the gate of a queue is open, packets at the head of that queue may become eligible for transmission. When the gate is closed, packets remain blocked in the queue even if they have already arrived.

A packet $p_{i,j}$ is therefore considered *eligible* for transmission at an egress port if the following conditions are satisfied:

- the packet has arrived at the queue,
- the packet is at the head of its queue,
- the gate associated with that queue is open.

Whenever the outgoing link becomes available, the switch examines all queues whose gates are open and selects the packet with the highest priority among the eligible packets for transmission.

The duration of the transmission of packet $p_{i,j}$ corresponds to its transmission time $C_{p_{i,j}}$. During this interval, the link is occupied and no other packet may be

transmitted. Once the transmission completes, the switch again evaluates the queues and selects the next eligible packet for transmission.

The state of the gates associated with the queues is controlled by traffic shaping mechanisms implemented in the switch. These mechanisms regulate when packets of different traffic classes may be transmitted and therefore influence the waiting time experienced by packets in the queues. The operation of these shaping mechanisms is discussed in the following section.

4.5 Traffic Shaping Mechanisms

As already discussed , packets arriving at a TSN switch are stored in priority queues associated with each egress port. While priority-based selection determines which packet is transmitted among eligible queues, additional mechanisms are required to regulate when packets become eligible for transmission. Without such mechanisms, bursts of traffic from high-priority flows could monopolise the transmission link and introduce unpredictable delays for other traffic classes.

To address this problem, TSN introduces traffic shaping mechanisms that control the eligibility of packets within the transmission queues. Two shaping mechanisms are particularly relevant in the context of this thesis: the Time-Aware Shaper (TAS) and the Credit-Based Shaper (CBS). Both mechanisms regulate packet transmissions at the level of the output queues, but they operate according to different principles.

4.5.1 Time-Aware Shaper (TAS)

The Time-Aware Shaper (TAS), standardized in IEEE 802.1Qbv, is a traffic shaping mechanism designed to provide highly deterministic packet transmission behaviour. TAS is particularly well suited for communication flows with strict timing requirements, such as those found in industrial control systems, where packets must be delivered with minimal delay and low jitter.

The key idea behind TAS is to regulate packet transmissions according to a predefined time schedule. As discussed earlier, each transmission queue at an egress port is associated with a gate that determines whether packets from that queue may participate in transmission decisions. In TAS-enabled systems, the state of these gates follows a periodic schedule known as a *Gate Control List* (Gate Control List (GCL)). The GCL specifies a sequence of time intervals during which the gates of individual queues are either open or closed. Since TSN networks rely on precise network-wide clock synchronization, these schedules can be applied consistently across multiple switches in the network.

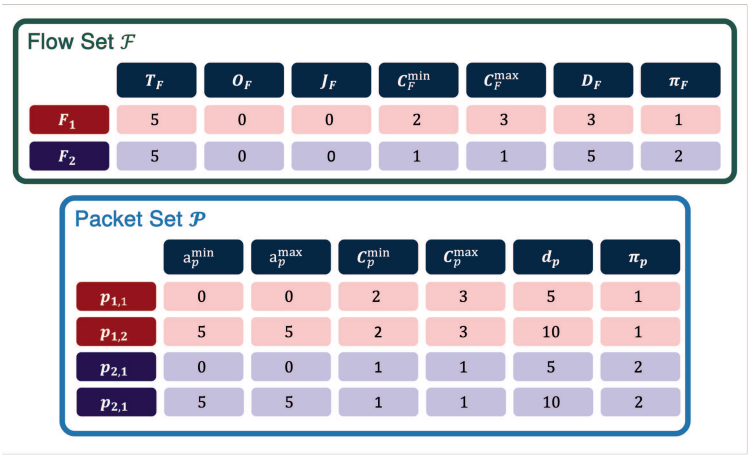


Figure 4.2: Flow set and the corresponding set of packets released during the first 10 time units

At any given time, only packets belonging to queues whose gates are open are eligible for transmission. Among the eligible packets, the switch selects the packet with the highest priority according to the priority-based scheduling rule described earlier. By carefully designing the gate schedule, TAS allows certain traffic classes to obtain exclusive transmission opportunities, effectively isolating them from interference caused by lower-priority traffic.

EXAMPLE 4.1

To illustrate the behaviour of TAS, consider the flow set shown in Figure 4.2. The figure presents two communication flows F_1 and F_2 together with the packets they generate for 10 time units and the gate schedule controlling their transmission. Both flows inject packets periodically with period $T = 5$ and without offsets or jitter. Flow F_1 has a higher priority ($\pi_{F_1} = 1$) than flow F_2 ($\pi_{F_2} = 2$). Packets generated by F_1 require a transmission time of 2 to 3 time units, whereas packets generated by F_2 require a transmission time of 1 time unit.

The resulting packet set within the first 10 time units is shown in Figure 4.2. Flow F_1 generates packets $p_{1,1}$ and $p_{1,2}$ arriving at times 0 and 5, respectively. Similarly, flow F_2 generates packets $p_{2,1}$ and $p_{2,2}$ arriving at times 0 and 5. The deadlines of the packets correspond to the end of their respective periods.

In the schedule illustrated in Figure 4.3, we consider one of the possible transmission scenarios for the packet set of Figure 4.2. Packet $p_{1,1}$ is assumed to require the maximum transmission time of 3 time units, while packet

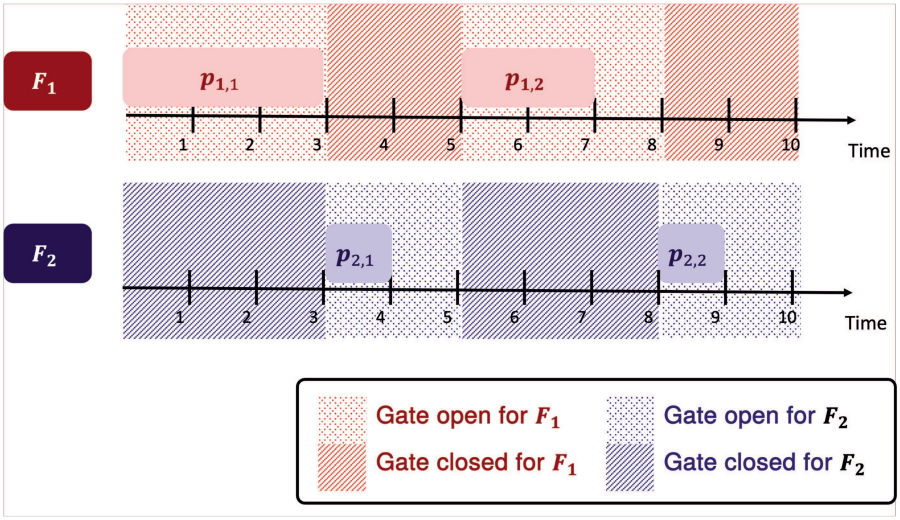


Figure 4.3: Example illustrating TAS scheduling

$p_{1,2}$ is assumed to require only 2 time units.

Figure 4.3 shows the gate schedule applied to the queues associated with these two flows. The gate of the queue corresponding to flow F_1 periodically opens during intervals $[0, 3)$ and $[5, 7)$, while the gate associated with flow F_2 opens during the complementary intervals. When the gate of a queue is closed, packets waiting in that queue cannot participate in the transmission decision.

Consider first packet $p_{1,1}$ generated by the high-priority flow F_1. Since the gate associated with F_1 is open during the interval $[0, 3)$, packet $p_{1,1}$ can be transmitted immediately upon arrival and completes its transmission at time 3. During this interval, packets belonging to F_2 cannot interfere because the gate associated with the F_2 queue remains closed.

Packet $p_{2,1}$ of flow F_2 arrives at time 0 but cannot be transmitted immediately since its gate is closed while the transmission window of F_1 is active. The packet must therefore wait until time 3, when the gate associated with F_2 opens. At this point, $p_{2,1}$ is transmitted and completes its transmission at time 4.

A similar behaviour occurs in the second period. Packet $p_{1,2}$ arrives at time 5 and is transmitted immediately during the transmission window allocated to F_1. The transmission of $p_{1,2}$ completes at time 7. However, the gate associated with flow F_2 only opens at time 8 according to the configured Gate Control List. Consequently, even though packets belonging to F_2 are waiting in the queue and the link becomes available at time 7, the switch does not start

transmitting them immediately. Instead, the egress port remains idle during the interval $[7, 8)$ in order to strictly follow the gate schedule defined by the GCL. Once the gate of F_2 opens at time 8, packet $p_{2,2}$ becomes eligible for transmission and is subsequently transmitted.
This example illustrates how TAS isolates the transmission opportunities of different traffic classes. By allocating dedicated transmission windows to the high-priority flow, TAS ensures that packets of F_1 experience deterministic and bounded delays. Lower-priority traffic, represented by flow F_2, may experience additional waiting time but cannot interfere with the transmission of higher-priority packets.

By allocating dedicated transmission windows to critical traffic classes, TAS enables deterministic communication patterns in which the transmission times of packets can be precisely controlled. This property makes TAS particularly suitable for applications requiring strict timing guarantees, such as periodic control loops and time-triggered communication in industrial networks.

4.5.2 Credit-Based Shaper (CBS)

While the Time-Aware Shaper provides deterministic transmission windows for critical traffic, some communication flows require more flexible bandwidth guarantees rather than strict time-triggered transmission schedules. In particular, applications such as audio and video streaming generate traffic that must be transmitted at a controlled rate but do not necessarily require exclusive transmission windows on the link. For such traffic classes, Time-Sensitive Networking introduces the Credit-Based Shaper (CBS), defined in IEEE 802.1Qav.

The Credit-Based Shaper regulates packet transmissions by controlling the rate at which packets from a queue may be transmitted. Instead of relying on predetermined gate schedules, CBS associates each shaped queue with a *credit* value that evolves over time. This credit determines when packets from the corresponding queue become eligible for transmission.

When packets are waiting in the queue but the queue is not currently transmitting, the credit increases over time according to a parameter known as the *idle slope*. When a packet from the queue is transmitted, the credit decreases according to the *send slope*. These parameters determine how quickly credit accumulates when the queue is idle (idle slope) and how quickly credit decreases while packets are transmitted (send slope). As a result, the credit reflects the recent transmission behaviour of the queue and controls the long-term transmission rate of the queue.

A packet at the head of a CBS queue becomes eligible for transmission only

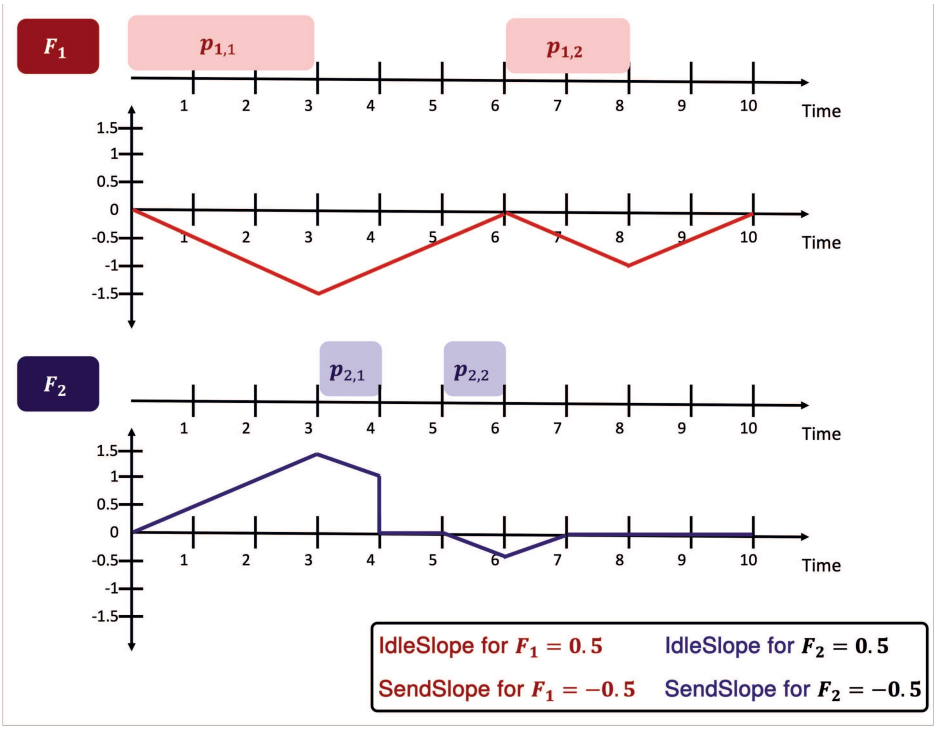


Figure 4.4: Example illustrating CBS scheduling

when the credit associated with that queue is non-negative. If the credit is negative, the queue must wait until the credit increases sufficiently before transmitting the next packet. This mechanism prevents bursts of traffic from occupying the link for extended periods and ensures that bandwidth is shared fairly between multiple traffic classes.

Once the packet begins transmission, the credit decreases according to the send slope and may become negative. In that case, the queue must accumulate credit again before another packet can be transmitted.

In contrast to TAS, where transmission opportunities are determined by predefined time windows, CBS regulates packet transmissions based on the evolution of the credit value. This results in smoother traffic patterns and allows flows to transmit packets more flexibly while still respecting their allocated bandwidth.

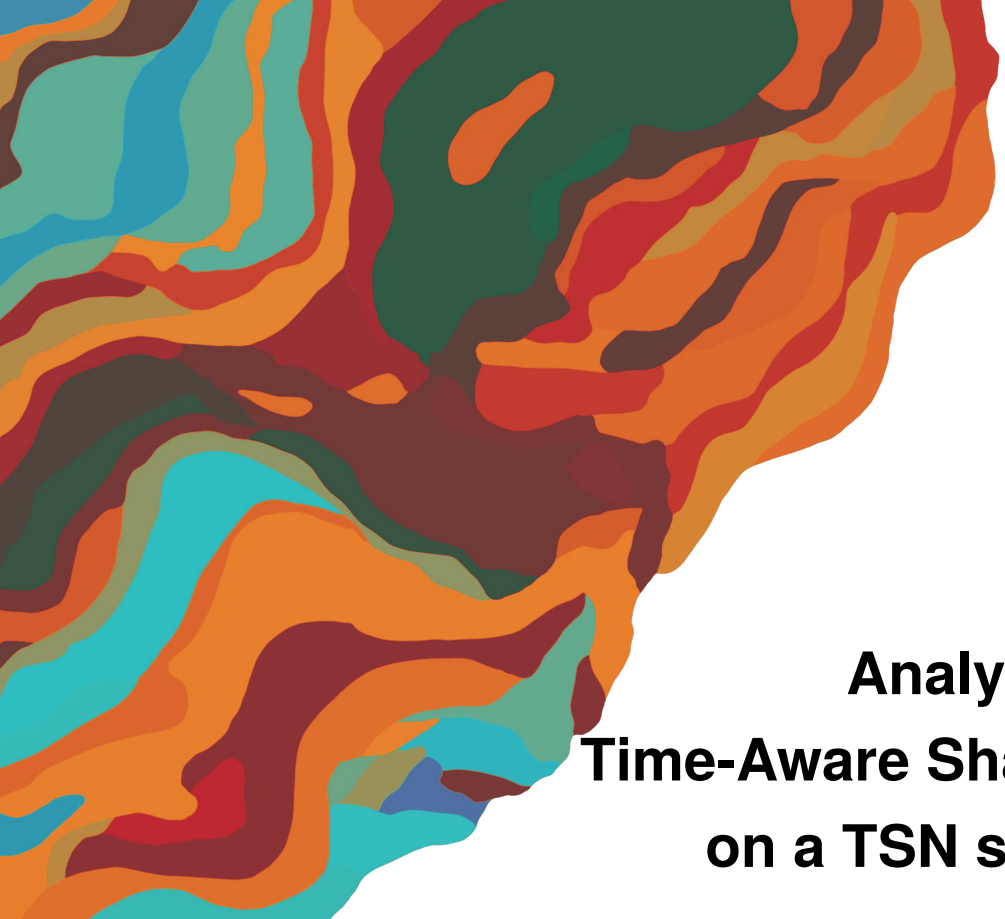
EXAMPLE 4.2

To illustrate the behaviour of CBS, consider the example shown in Figure 4.4. We use the same flow set and resulting packet set as introduced in Figure 4.2. Figure 4.4 illustrates the resulting packet transmissions together with the evolution of the credit values of the queues corresponding to F_1 and F_2. In

this example, both flows are shaped with identical parameters, where the idle slope is 0.5 and the send slope is -0.5. When a queue has packets waiting but is not transmitting, its credit increases over time according to the idle slope. When a packet from the queue is transmitted, the credit decreases according to the send slope.
At time 0, packet $p_{1,1}$ arrives and is transmitted immediately. During its transmission, the credit of the queue corresponding to F_1 decreases, resulting in a negative credit value by the time the transmission completes. Because the credit is negative, the queue of F_1 is temporarily prevented from transmitting another packet until the credit increases again during idle periods.
While F_1 is transmitting, the credit associated with the queue of F_2 increases since packet $p_{2,1}$ is waiting in the queue. Once the credit becomes non-negative, packet $p_{2,1}$ becomes eligible for transmission and is transmitted. During this transmission, the credit of F_2 decreases accordingly.
After the transmission of $p_{2,1}$ is finished, the credit is still non-negative, but there are no additional packets waiting in the queue for F_2, so the credit is reduced to 0. When $p_{2,2}$ later arrives in the system, its class credit is 0 (i.e., non-negative), while the credit of the class associated with F_1 is still negative. Therefore, $p_{2,2}$ is the highest-priority packet that is ready for service and is transmitted immediately.
Subsequently, $p_{1,2}$ is transmitted once its class credit becomes non-negative. In this way, the credit mechanism determines when packets from each queue are eligible for transmission, guaranteeing that each flow's transmission rate stays bounded while still allowing packets to be sent whenever adequate credit has accumulated.

The mechanisms introduced in this chapter regulate how packets compete for transmission in TSN networks. While TAS enforces deterministic transmission windows through time-based gate schedules, CBS controls packet transmissions through a rate-based credit mechanism. Both approaches introduce structured constraints on packet scheduling at network switches.

In the following chapters, we extend the SAG analysis framework to reason about the timing behaviour of TSN networks employing these shaping mechanisms.



5

Analysis of Time-Aware Shapers on a TSN switch

This chapter is based on:
Srinidhi Srinivasan et al. "Analysis of TSN Time-Aware Shapers Using Schedule Abstraction Graphs". In: 36th Euromicro Conference on Real-Time Systems (ECRTS 2024). *Vol. 298. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 16:1–16:24.* DOI: 10.4230/LIPIcs.ECRTS.2024.16.

TABLE OF CONTENTS

5.1 System Model
5.2 Latency Analysis Algorithm for TAS
5.3 Possibly Eligible Packets and Eligible Transmission Intervals
5.4 Expansion Phase
5.5 Merge Phase
5.6 Proof of Correctness
5.7 Evaluation
5.8 Conclusion

In Chapter 4, we introduced the TSN switch model and examined the mechanisms that determine packet transmission behaviour, including per-class FIFO queues, gate control, and priority-based selection. This chapter focuses on analysing the Time-Aware Shaper (TAS).

As discussed in Section 4.5.1, TAS regulates transmission opportunities through time-triggered gate control. At specific time instants, gates open or close, thereby determining which traffic classes may compete for transmission. Due to its explicit time-triggered structure, TAS is typically used for traffic classes with the most stringent timing requirements.

To guarantee that time-sensitive flows satisfy their latency constraints, it is necessary to derive safe lower and upper bounds on packet delay at each switch. Existing approaches for TAS analysis are predominantly based on Network Calculus and Compositional Performance Analysis (Compositional Performance Analysis (CPA)). While these methods provide safe upper bounds, they are generally sufficient analyses and may therefore be pessimistic. In addition, many existing analyses impose structural restrictions on the gate control list, such as assuming non-overlapping transmission windows or limiting the number of gate events per cycle.

Such restrictions simplify the analysis but may reduce flexibility and lead to under-utilisation of the bandwidth. In practice, TSN standards allow arbitrary periodic gate schedules, including overlapping gate openings across traffic classes. Supporting such general configurations requires an analysis technique capable of reasoning about concrete transmission interactions without relying on overly conservative bounds.

In this chapter, we present a new latency analysis for TAS in a single switch setting that allows arbitrary periodic gate schedules without restricting the number, structure, or overlap of gate events. The analysis is inspired by the Schedule Abstraction Graph framework developed for task scheduling and presented in Part I. We adapt the SAG methodology to capture time-triggered gate behaviour and packet transmission decisions in a TSN switch.

In the remainder of this chapter, we formalise the TAS model and introduce the corresponding extension of the Schedule Abstraction Graph.

5.1 System Model

This section defines the system model used to analyse the latency of Time-Aware Shaper (TAS) traffic at a single egress port of a TSN switch. The model follows the structure introduced in Chapter 4, and focuses on periodic flows and periodic gate schedules.

5.1.1 Flow Model

Let $\mathcal{F}$ be a finite set of communication flows transmitted through the considered egress port. Each flow $F_i \in \mathcal{F}$ generates packets periodically with period T_i. Each packet of F_i has an end-to-end deadline D_i such that

$$D_i \leq T_i.$$

This implies that, if all deadlines are met, at most one packet of F_i arrives at a switch in each interval $[k \times T_i, (k+1) \times T_i)$, where $k \in \mathbb{N}$.

We assume that all flows start at the same time, i.e., there are no offsets.

Since the end-to-end analysis is performed per switch, we analyse packet transmission latency independently at each switch along the route.

Let $\mathcal{P}$ be the finite set of packets transmitted through the considered egress port. Each packet $p_{i,j} \in \mathcal{P}$ is described by:

$$p_{i,j} = \left(a^{\min}_{p_{i,j}},\ a^{\max}_{p_{i,j}},\ C^{\min}_{p_{i,j}},\ C^{\max}_{p_{i,j}},\ d_{p_{i,j}},\ \pi_{p_{i,j}} \right),$$

as defined in Section 4.4.1.

Transmission Scenario We introduce the notion of a *transmission scenario*, which is adapted from the concept of execution scenario defined in Section 2.2.

A *transmission scenario* is a concrete choice of arrival times and transmission durations for all packets in $\mathcal{P}$, within their specified bounds. Since TAS operates deterministically, each scenario results in a unique packets transmission order.

Definition 5.1 (Transmission Scenario)**.** A transmission scenario γ specifies a concrete arrival time $a_{p_{i,j}}$ and transmission duration $C_{p_{i,j}}$ for each packet $p_{i,j} \in \mathcal{P}$:

$$\gamma = \left\{ \left(p_{i,j}, a_{p_{i,j}}, C_{p_{i,j}} \right) \,\middle|\, p_{i,j} \in \mathcal{P} \right\}, \tag{5.1}$$

where $a_{p_{i,j}} \in [a^{\min}_{p_{i,j}}, a^{\max}_{p_{i,j}}]$ and $C_{p_{i,j}} \in [C^{\min}_{p_{i,j}}, C^{\max}_{p_{i,j}}]$.

If multiple packets arrive simultaneously at a queue, their order is determined by a deterministic but potentially unknown tie-breaking rule. Since this rule may not be known a priori, all possible tie-breaking outcomes are considered implicitly as part of the set of transmission scenarios.

Latency Definition For a given transmission scenario, let $f_{p_{i,j}}$ denote the finish time of packet $p_{i,j}$, i.e., the time at which its transmission ends.

The latency (or response time) of $p_{i,j}$, denoted by $R_{p_{i,j}}$, is defined as:

$$R_{p_{i,j}} = f_{p_{i,j}} - a^{\min}_{p_{i,j}}.$$

The best-case and worst-case latency of $p_{i,j}$, denoted by $\mathrm{BCRT}p_{i,j}$ and $\mathrm{WCRT}p_{i,j}$, are the minimum and maximum possible values of $R_{p_{i,j}}$ across all transmission scenarios.

The system is considered *schedulable* if, for all $p_{i,j} \in \mathcal{P}$,

$$\mathrm{WCRT}p_{i,j} \leq d_{p_{i,j}} - a^{\min}_{p_{i,j}}.$$

Otherwise, if there exists at least one packet $p_{i,j}$ such that

$$\mathrm{WCRT}p_{i,j} > d_{p_{i,j}} - a^{\min}_{p_{i,j}},$$

the system is considered *unschedulable*.

5.1.2 Observation Window

Let H be the least common multiple of all flow periods T_i and all gate schedule periods. Since both flows and gate schedules are periodic and start at time 0, the system repeats every H time units. We call H the length of a hyper-period.

Because each flow satisfies $D_i \leq T_i$, every packet released before time H must complete transmission by time H in any schedulable system. Therefore, the response time of packets released in one hyper-period is not influenced by packets released in a different hyper-period.

Therefore, it is sufficient to analyse packets released within the interval $[0, H)$ to check the system schedulability. We call this interval the *observation window*. The packet set $\mathcal{P}$ contains all packets released during this window.

Lemma 5.2. *For worst-case latency analysis, it is sufficient to consider packets released within a time interval of length H.*

Proof. Flows and gate schedules are periodic and aligned at time 0. Thus, at time H, all flows and schedules start a new period, and the system returns to the same abstract state as at time 0.

Since H is a common multiple of the periods of all flows and all flows have zero offset, the last packet of any flow F_i released in the interval $[0, H)$ arrives at time

$$a_{i,\mathrm{last}} = H - T_i.$$

Because $D_i \leq T_i$, its deadline is

$$d_{i,\mathrm{last}} = a_{i,\mathrm{last}} + D_i = H - T_i + D_i \leq H.$$

Hence, every packet released in $[0, H)$ has its deadline at or before time H. Therefore, in a schedulable system, no packet released before H can remain in

the system after time H.

Because at time H all flows and gate schedules return to the same configuration as at time 0, and the TSN scheduling policy is deterministic, the behaviour after time H is identical to that in $[0, H)$. Therefore, packets released after H cannot experience worse latency than those released within $[0, H)$. □

Hence, it is sufficient to analyse the finite set of packets $\mathcal{P}$ released in the interval $[0, H)$, which we refer to as the observation window.

5.1.3 Switch model

As introduced in Chapter 4, TSN groups packets into traffic classes based on their priority. Let $\mathcal{C}$ denote the set of all traffic classes. Each class $\mathcal{C}_a \in \mathcal{C}$ has a unique priority $\pi_{\mathcal{C}_a}$. A packet belongs to class $\mathcal{C}_a$ if it has the same priority. Hence, the set of packets in $\mathcal{C}_a$ is

$$\mathcal{P}_{\mathcal{C}_a} = \{p_{i,j} \mid \pi_{p_{i,j}} = \pi_{\mathcal{C}_a}\}.$$

Each class has its own FIFO queue. When a packet arrives at the switch, it is placed in the queue of its class. Each queue is controlled by a time-aware shaper gate. A packet can only be transmitted when the gate of its class is open.

The opening and closing of each gate is defined by a gate-control list (GCL), which specifies a periodic schedule. For a class $\mathcal{C}_a$, we denote by $\mathcal{G}_{\mathcal{C}_a}$ the set of time intervals during which its gate is open:

$$\mathcal{G}_{\mathcal{C}_a} = \{[go_1^{\mathcal{C}_a}, gc_1^{\mathcal{C}_a}),\ [go_2^{\mathcal{C}_a}, gc_2^{\mathcal{C}_a}),\ \ldots,\ [go_{n_{\mathcal{C}_a}}^{\mathcal{C}_a}, gc_{n_{\mathcal{C}_a}}^{\mathcal{C}_a})\}.$$

Here, $[go_l^{\mathcal{C}_a}, gc_l^{\mathcal{C}_a})$ is the l-th open interval, and $n_{\mathcal{C}_a}$ is the number of such intervals within the observation window of length H. The intervals are non-overlapping, i.e., $gc_l^{\mathcal{C}_a} < go_{l+1}^{\mathcal{C}_a}$ for all l. The gate is closed outside these intervals.

To prevent incomplete transmissions, TSN uses a guard band: a packet is not allowed to start transmission if it cannot finish before the gate closes.

Ready condition A packet $p_{i,j}$ is *ready* for transmission at time t if all of the following hold:

R1 $p_{i,j}$ is at the top of its class queue at time t.

R2 The gate of its class is open at time t.

R3 The gate remains open until at least time $t + C_{p_{i,j}}$.

Transmission policy When the egress port is free, the switch transmits the highest-priority ready packet. If no packet is ready, the port remains idle. Packet transmissions are non-pre-emptive. If a packet starts transmission at time t, it completes at time $t + C_{p_{i,j}}$.

After each transmission, the Ethernet protocol enforces an *inter-packet gap* (Inter-Packet Gap (IPG)), denoted by C_{IPG}. During this time, no packet can be transmitted. Hence, the egress port becomes free again at time $t + C_{p_{i,j}} + C_{\mathrm{IPG}}$.

Operator definitions For a set $\mathcal{X}$, we define

$$\min_{\infty}(\mathcal{X}) = \min(\mathcal{X} \cup \{\infty\}).$$

For two sets of time intervals $\mathcal{I}_A$ and $\mathcal{I}_B$, we define their difference as

$$\mathcal{I}_A \setminus \mathcal{I}_B = \{t \mid t \in \mathcal{I}_A \ \wedge \ t \notin \mathcal{I}_B\}.$$

5.2 Latency Analysis Algorithm for TAS

This section presents the algorithm used to compute latency bounds for TAS-shaped traffic. The algorithm explores all feasible transmission orders within the observation window and derives safe bounds on both the best- and worst-case latency of each packet in $\mathcal{P}$.

5.2.1 Overview and Challenges

The algorithm is based on the general concept of the Schedule Abstraction Graph (SAG) introduced in Chapter 2, in which all feasible transmission orders and the corresponding abstract states are systematically explored. However, constructing a SAG for analysing the worst-case latency of packets transmitted by a TSN switch calls for an entirely different approach from that presented in Chapter 2 and Chapter 3, and any published work on the schedule-abstraction framework. The primary reason is that TSN switch characteristics necessitate substantial extensions of the SAG.

- **FIFO queues:** Packets of the same traffic class share the same priority and are served in FIFO order. As a result, the transmission order depends on their arrival times. Since arrival times may vary across transmissions, the relative order between packets of the same class is not fixed. This breaks the key SAG assumption of a consistent tie-breaking rule between equal-priority jobs and makes the transmission order scenario-dependent. We discuss how we address this challenge in Section 5.3.1.

- **Time-Aware Shaper (TAS):** Packets may only transmit when their gate is open. This makes the scheduling policy non-work-conserving, as the egress port may remain idle even when packets are waiting. In addition, the periodic opening and closing of gates restricts when packets can start transmission, which leads to multiple disjoint feasible transmission intervals and complicates the analysis of interference from higher-priority traffic. We discuss the solution in Section 5.3.3.

These aspects invalidate the assumptions of the basic SAG model. The algorithm therefore extends the SAG construction to account for arrival-dependent ordering and gate-controlled transmission.

5.2.2 Graph Construction Algorithm

An abstract state s is defined by:

- the set of packets that have already been transmitted, denoted by $\mathcal{D}$,
- an interval $[A^{\min}, A^{\max}]$ describing when the egress port may become available.

The exact availability time is not known due to uncertainty in arrivals and transmission durations. Instead:

- the egress port is certainly unavailable before $A^{\min}$,
- the egress port is guaranteed to be available at or after $A^{\max}$,
- within $[A^{\min}, A^{\max})$, the egress port may or may not be available.

Edges in the graph represent packet transmissions. An edge labelled with packet $p_{i,j}$ connects the abstract state s to a successor state s'. This is illustrated in the representation of the schedule abstraction graph in Figure 5.2, that is built using the packet set defined in Figure 5.1, where traffic classes are ordered by priority, with lower numerical class identifiers corresponding to higher-priority traffic. The figure only depicts the availability intervals of each abstract state, and not the set of transmitted packets, in order to keep the illustration concise.

Initialization and Exploration. For each packet $p_{i,j} \in \mathcal{P}$, the algorithm maintains (lines 3 and 4):

- $\mathrm{BCRT}_{p_{i,j}}$: best-case latency (initially ∞),
- $\mathrm{WCRT}_{p_{i,j}}$: worst-case latency (initially 0).

Packet Set $\mathcal{P}$

	$\mathcal{C}_a$	a_p^{min}	a_p^{max}	C_p^{max}	C_p^{max}	$\mathcal{G}_{\mathcal{C}_a}$
$p_{1,1}$	$\mathcal{C}_1$	1	8	11	11	{[0,4),[23,60)}
$p_{2,1}$	$\mathcal{C}_2$	11	13	4	4	{[0,10),[17,60)}
$p_{3,1}$	$\mathcal{C}_3$	3	10	4	4	{[0,7),[14,60)}
$p_{4,1}$	$\mathcal{C}_4$	0	0	1	21	{[0,60)}

Figure 5.1: Example packet set.

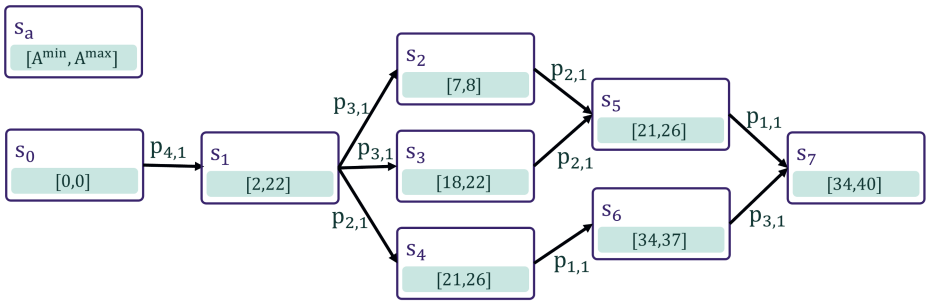


Figure 5.2: SAG built using the packet set in Figure. 5.1

Exploration starts from the initial state $s_0 = (\emptyset, [0, 0])$ (line 6). The state space is explored in a breadth-first manner until all possibilities are covered or a deadline violation is detected.

Eligible Packets. In each reachable abstract state, the algorithm determines the set of *possibly eligible* packets. A packet is possibly eligible if it can become ready according to rules R1–R3 in that abstract state.

Eligible Transmission Intervals. For each eligible packet $p_{i,j}$, the algorithm computes its eligible transmission intervals (line 12), i.e., the time intervals in which $p_{i,j}$ may start transmission while respecting:

- its arrival bounds,
- the egress port availability,
- the gate schedule,
- the higher-priority packets interference.

Due to the periodic gate schedule, a packet may have multiple disjoint start intervals. Each interval leads to a distinct successor state.

Algorithm 3 Schedule Abstraction Graph generation algorithm with FIFO Queues and TAS

1: **Input:** Finite Packet Set $\mathcal{P}$
2: **for all** $p_{i,j} \in \mathcal{P}$ **do**
3: $\quad \mathrm{BCRT}p_{i,j} \leftarrow \infty$
4: $\quad \mathrm{WCRT}p_{i,j} \leftarrow 0$
5: **end for**
6: $s_0 \leftarrow (\emptyset,\ [0,0])$
7: $L^{\mathrm{cur}} \leftarrow \{s_0\}$
8: $L^{\mathrm{next}} \leftarrow \emptyset$
9: **while** $L^{\mathrm{cur}} \neq \emptyset$ **do**
10: $\quad$ **for all** $s \in L^{\mathrm{cur}}$ **do**
11: $\quad\quad$ **for** each eligible packet $p_{i,j}$ according to Section 5.3.2 **do**
12: $\quad\quad\quad \mathcal{E}_{p_{i,j}} \leftarrow$ Equation (5.13)
13: $\quad\quad\quad$ **for** each interval $\mathcal{E}^{l}_{p_{i,j}}$ in $\mathcal{E}_{p_{i,j}}$ **do**
14: $\quad\quad\quad\quad EFT_{p_{i,j}} \leftarrow$ Equation (5.15)
15: $\quad\quad\quad\quad LFT_{p_{i,j}} \leftarrow$ Equation (5.14)
16: $\quad\quad\quad\quad \mathrm{BCRT}p_{i,j} \leftarrow \min\{EFT_{p_{i,j}} - a^{\min}_{p_{i,j}},\ \mathrm{BCRT}p_{i,j}\}$
17: $\quad\quad\quad\quad \mathrm{WCRT}p_{i,j} \leftarrow \max\{LFT_{p_{i,j}} - a^{\min}_{p_{i,j}},\ \mathrm{WCRT}p_{i,j}\}$
18: $\quad\quad\quad\quad$ Create a new abstract state s'
19: $\quad\quad\quad\quad A^{\min} \leftarrow$ Equation (5.16)
20: $\quad\quad\quad\quad A^{\max} \leftarrow$ Equation (5.17)
21: $\quad\quad\quad\quad$ Connect s to s' with an edge labelled $p_{i,j}$
22: $\quad\quad\quad\quad L^{\mathrm{next}} \leftarrow L^{\mathrm{next}} \cup \{s'\}$
23: $\quad\quad\quad$ **end for**
24: $\quad\quad$ **end for**
25: $\quad$ **end for**
26: $\quad$ Merge all states in L^{next}
27: $\quad L^{\mathrm{cur}} \leftarrow L^{\mathrm{next}}$
28: $\quad L^{\mathrm{next}} \leftarrow \emptyset$
29: **end while**

Finish-Time Bounds and State Update. For each interval, the algorithm computes the earliest finish time $f^{\min}_{p_{i,j}}$ (line 14) and the latest finish time $f^{\max}_{p_{i,j}}$ (line 15).

Latency bounds (lines 16 and 17) are updated as:

$$\mathrm{BCRT}_{p_{i,j}} \leftarrow \min\{\mathrm{BCRT}_{p_{i,j}}, f^{\min}_{p_{i,j}} - a^{\min}_{p_{i,j}}\},$$

$$\mathrm{WCRT}_{p_{i,j}} \leftarrow \max\{\mathrm{WCRT}_{p_{i,j}}, f^{\max}_{p_{i,j}} - a^{\min}_{p_{i,j}}\}.$$

If

$$\mathrm{WCRT}_{p_{i,j}} > d_{p_{i,j}} - a^{\min}_{p_{i,j}},$$

the algorithm terminates and declares the system unschedulable.

Otherwise, a successor state is created per interval, updating:

- the set of transmitted packets,
- the egress port-availability interval (accounting for the IPG).

State Merging. As with the SAG analyses presented in previous chapters, to control state-space growth, states are merged when:

- they contain the same transmitted packets,
- their availability intervals overlap or are contiguous.

Merging is performed so that correctness is preserved, in the sense that every concrete state remains covered by the merged state.

5.2.3 Key Differences with the generic SAG construction algorithm

Compared to the generic SAG construction of Chapter 2, the main differences are that:

- the eligibility of a packet depends on FIFO position and gate state,
- multiple eligible start intervals may exist for a single packet,
- successor states are generated per interval rather than per packet,
- guard-band and IPG constraints influence finish times,
- gate schedules introduce additional timing constraints.

If the exploration completes without detecting a deadline violation, the algorithm returns the computed best- and worst-case latency bounds for all packets in $\mathcal{P}$ and concludes that the system is schedulable.

5.3 Possibly Eligible Packets and Eligible Transmission Intervals

According to the SAG generation algorithm discussed in Section 5.2, in an abstract state s, a packet $p_{i,j}$ may be transmitted next through the egress if it is the highest-priority packet that is ready in that abstract state. Ready means that $p_{i,j}$ is at the top of its FIFO queue, and the gate of its class is open and will not close until $C_{p_{i,j}}$ time units later. We discuss the impact of FIFO ordering in Section 5.3.1, and how to obtain the intervals in which a packet is the highest-priority ready packet in Sections 5.3.2 and 5.3.3.

5.3.1 Accounting for FIFO Queues

Each traffic class has a FIFO queue that stores the packets arriving into the switch. Packets must be at the top of their FIFO queue to possibly be transmitted next on the switch's egress (by Rule R1 in Section 5.1.3). In the rest of this section, we derive the set of packets that may be at the top of the FIFO queue when the system reached an abstract state s. To do so, we first derive the earliest time when a packet will certainly be in the queue of each traffic class $\mathcal{C}_a \in \mathcal{C}$ (Lemma 5.3). We denote that time instant by $tr_{\mathcal{C}_a}^{\max}$. We then use that time to get the set $top_{\mathcal{C}_a}$ of all packets that may have entered $\mathcal{C}_a$'s queue before $tr_{\mathcal{C}_a}^{\max}$ (Lemma 5.4), and we prove that a packet $p_{i,j}$ may possibly be at the top of its queue if and only if it is in $top_{\mathcal{C}_a}$ (Theorem 5.7).

Lemma 5.3. *If the system is in abstract state s, then for any concrete state represented by s, the earliest time at which a not yet transmitted packet of class $\mathcal{C}_a$ will* certainly *enter its FIFO queue is upper bounded by*

$$tr_{\mathcal{C}_a}^{\max} = \min_{\infty}\{a_{p_{i,j}}^{\max} | p_{i,j} \in \mathcal{C}_a \setminus \mathcal{P}\} \tag{5.2}$$

Proof. The earliest time when a packet is certainly in $\mathcal{C}_a$'s queue is the earliest time a packet of class $\mathcal{C}_a$ certainly arrives in the switch in any concrete state in the abstract state s. That time is the earliest maximum arrival time of any packet of class $\mathcal{C}_a$ that was not transmitted yet in state s (i.e., any packet $p_{i,j} \in \mathcal{C}_a \setminus \mathcal{P}$). Thus, $tr_{\mathcal{C}_a}^{\max}$ is given by $\min_{\infty}\{a_{p_{i,j}}^{\max} | p_{i,j} \in \mathcal{C}_a \setminus \mathcal{P}\}$. □

We use $tr_{\mathcal{C}_a}^{\max}$ in Lemma 5.4 to derive the set of packets that may possibly be in $\mathcal{C}_a$'s FIFO queue at $tr_{\mathcal{C}_a}^{\max}$.

Lemma 5.4. *If the system is in abstract state s, then for any concrete state represented by s, the packet at the top of the FIFO queue of class $\mathcal{C}_a$ must be in the set*

$$top_{\mathcal{C}_a} = \{p_{i,j} | p_{i,j} \in \mathcal{C}_a \setminus \mathcal{P} \wedge a_{p_{i,j}}^{\min} \leq tr_{\mathcal{C}_a}^{\max}\}. \tag{5.3}$$

Proof. Only packets of class $\mathcal{C}_a$ that have not yet been transmitted (i.e., all packets in $\mathcal{C}_a \setminus \mathcal{P}$) may be in $\mathcal{C}_a$'s FIFO queue at $tr_{\mathcal{C}_a}^{\max}$. Moreover, only packets that may potentially arrive in the switch at or before $tr_{\mathcal{C}_a}^{\max}$ may be in the queue by $tr_{\mathcal{C}_a}^{\max}$. Thus, only packets in $\mathcal{C}_a \setminus \mathcal{P}$ with an earliest arrival time $a_{p_{i,j}}^{\min}$ such that $a_{p_{i,j}}^{\min} \leq tr_{\mathcal{C}_a}^{\max}$ may be in $\mathcal{C}_a$'s FIFO queue at $tr_{\mathcal{C}_a}^{\max}$. □

We now prove with the next two lemmas that all packets in $top_{\mathcal{C}_a}$ may be at the top of $\mathcal{C}_a$'s FIFO queue and that no packet outside of $top_{\mathcal{C}_a}$ may be at the top of $\mathcal{C}_a$'s FIFO queue when in abstract state s.

Lemma 5.5. *If packet $p_{i,j}$ is in $top_{\mathcal{C}_a}$, then there is at least one transmission scenario where $p_{i,j}$ is at the top of $\mathcal{C}_a$'s FIFO queue in abstract state s.*

Proof. A packet $p_{i,j}$ is at the top of $\mathcal{C}_a$'s FIFO queue if it is the first not-yet-transmitted packet of $\mathcal{C}_a$ that arrives in the switch. We know that if packet $p_{i,j}$ is in $top_{\mathcal{C}_a}$, then its earliest arrival time $a^{\min}_{p_{i,j}}$ is such that $a^{\min}_{p_{i,j}} \leq tr^{\max}_{\mathcal{C}_a}$ (by Equation (5.3)). Moreover, by Equation (5.2), we also know that for all packets $p_{i,j} \in \mathcal{C}_a \setminus \mathcal{P}$ that have not yet been transmitted, their latest arrival time is larger than or equal to $tr^{\max}_{\mathcal{C}_a}$, i.e., $\forall p_{i,j} \in \mathcal{C}_a \setminus \mathcal{P}, a^{\max}_{p_{i,j}} \geq tr^{\max}_{\mathcal{C}_a}$. Therefore, there is at least one transmission scenario where $p_{i,j}$ arrives at $a_{p_{i,j}}$ such that $a^{\min}_{p_{i,j}} \leq a_{p_{i,j}} \leq tr^{\max}_{\mathcal{C}_a}$ and all other packets of class $\mathcal{C}_a$ that have not yet been transmitted arrive at or after $tr^{\max}_{\mathcal{C}_a}$. In such transmission scenario, $p_{i,j}$ is the first to enter $\mathcal{C}_a$'s queue and is thus at the top of the queue. □

Lemma 5.6. *If $p_{i,j}$ is not in $top_{\mathcal{C}_a}$, then $p_{i,j}$ cannot be at the top of $\mathcal{C}_a$'s FIFO queue in abstract state s.*

Proof. From Lemma 5.3, we know that there is certainly a packet in $\mathcal{C}_a$'s FIFO queue at time $tr^{\max}_{\mathcal{C}_a}$ in abstract state s. Thus, any packet at the top of $\mathcal{C}_a$'s FIFO queue in abstract state s must (1) not have been transmitted yet in abstract state s (i.e., it must be in $\mathcal{C}_a \setminus \mathcal{P}$), and (2) must have arrived in the switch at or before $tr^{\max}_{\mathcal{C}_a}$ (i.e., $a_{p_{i,j}} \leq tr^{\max}_{\mathcal{C}_a}$). However, if $p_{i,j}$ is not in $top_{\mathcal{C}_a}$, then, by Equation (5.3), either $p_{i,j} \notin \mathcal{C}_a \setminus \mathcal{P}$ or $a^{\min}_{p_{i,j}} > tr^{\max}_{\mathcal{C}_a}$ and thus $a_{p_{i,j}} > tr^{\max}_{\mathcal{C}_a}$ (since $a^{\min}_{p_{i,j}}$ is a lower bound on $p_{i,j}$'s arrival time). Therefore, if $p_{i,j}$ is not in $top_{\mathcal{C}_a}$, then $p_{i,j}$ cannot be at the top of $\mathcal{C}_a$'s FIFO queue in abstract state s. □

We use the two lemmas above to derive our main theorem.

Theorem 5.7. *There is at least one transmission scenario such that packet $p_{i,j}$ will be at the top of $\mathcal{C}_a$'s FIFO queue in abstract state s if and only if $p_{i,j} \in top_{\mathcal{C}_a}$.*

Proof. The "if" is proven by Lemma 5.5, and the "only if" is the contrapositive of Lemma 5.6. □

5.3.2 Potentially eligible packets

After computing all packets that can possibly be on the top of their respective queue, we find which of these packets can potentially be the next packet to be transmitted on the egress in a concrete state represented by abstract state s. To decide whether a packet $p_{i,j}$ is potentially eligible to be the next transmitted packet, we first derive a lower bound, called $t^{\text{lb}}_{p_{i,j}}$, on the earliest time $p_{i,j}$ may start its transmission in the abstract state s, and an upper bound, called $t^{\text{ub}}_{p_{i,j}}$, on the latest time by which $p_{i,j}$ must start to transmit to be the next packet transmitted

on the egress. If $t^{\text{lb}}_{p_{i,j}} > t^{\text{ub}}_{p_{i,j}}$, then $p_{i,j}$ cannot be the next packet transmitted in abstract state s as it would otherwise lead to a contradiction on the definitions of $t^{\text{lb}}_{p_{i,j}}$ and $t^{\text{ub}}_{p_{i,j}}$. On the other hand, if $t^{\text{lb}}_{p_{i,j}} \leq t^{\text{ub}}_{p_{i,j}}$, then $p_{i,j}$ is potentially eligible to be transmitted next in abstract state s.

Lower-bound on $p_{i,j}$'s start time

We first prove a lower bound on packet $p_{i,j}$'s start of transmission in an abstract state s.

Lemma 5.8. *A packet $p_{i,j}$ may not start being transmitted earlier than*

$$t^{\text{lb}}_{p_{i,j}} = \max\{a^{\min}_{p_{i,j}}, A^{\min}\}. \tag{5.4}$$

Proof. $p_{i,j}$ may not start being transmitted before the time it arrives in the switch, which is lower bounded by $a^{\min}_{p_{i,j}}$. Furthermore, it cannot be transmitted before the earliest time the egress port is available in abstract state s, which is lower bounded by $A^{\min}$. Thus, $p_{i,j}$ cannot be transmitted earlier than $\max\{a^{\min}_{p_{i,j}}, A^{\min}\}$. □

Upper-bound on $p_{i,j}$'s start time

According to the TSN scheduling policy described in Section 5.1, a packet $p_{i,j}$ will certainly be transmitted when the egress port is available, $p_{i,j}$ is ready, and it is the highest priority packet that is ready for transmission. Therefore, as soon as there is a ready packet and the egress port is available, a packet will certainly be transmitted. In the following, we compute an upper bound on the time at which the egress port is available and a packet is certainly ready.

According to Rule R1, see Section 5.1.3, to be ready, a packet must be at the top of its queue. The following lemma directly follows.

Lemma 5.9. *The earliest time when the egress is certainly available and there is certainly a packet at the top of $\mathcal{C}_a$'s FIFO queue in abstract state s is*

$$\hat{t}_{\mathcal{C}_a}(s) = \max\{tr^{\max}_{\mathcal{C}_a}, A^{\max}\} \tag{5.5}$$

Proof. Only packets in the set $top_{\mathcal{C}_a}$ of each class $\mathcal{C}_a$ may be ready in abstract state s. According to Lemma 5.3, a packet may reach the top of $\mathcal{C}_a$'s FIFO queue at the latest at $tr^{\max}_{\mathcal{C}_a}$. Furthermore, by definition of $A^{\max}$, the egress becomes available at the latest at $A^{\max}$ in abstract state s. Thus, the earliest time when the egress is *certainly* available and there is *certainly* a packet at the top of $\mathcal{C}_a$'s FIFO queue is $\max\{tr^{\max}_{\mathcal{C}_a}, A^{\max}\}$. □

Then, according to Rules R2 and R3, for a packet $p_{i,j}$ to be ready, the gate of its class must be open and must not close until $C_{p_{i,j}}$ time units later. To find the first time instant when that condition is certainly true, Lemma 5.10 derives the set of intervals in which $p_{i,j}$ may not be able to start to be transmitted because either R2 and/or R3 is not respected.

Lemma 5.10. *A packet $p_{i,j}$ of class $\mathcal{C}_a$ may* not *be able to start transmitting in any time interval in the set $\mathcal{G}^{\text{pos}}_{p_{i,j}}$ defined below due to not respecting Rule R2 and/or Rule R3.*

$$\mathcal{G}^{\text{pos}}_{p_{i,j}} = \left\{ [0, go_1^{\mathcal{C}_a}) \cup \left\{ \bigcup_{1 \leq l < n_{\mathcal{C}_a}} (gc_l^{\mathcal{C}_a} - C^{\max}_{p_{i,j}}, go_{l+1}^{\mathcal{C}_a}) \right\} \cup (gc_{n_{\mathcal{C}_a}}^{\mathcal{C}_a} - C^{\max}_{p_{i,j}}, \infty) \right\} \tag{5.6}$$

Proof. According to Rule R2 packet $p_{i,j}$ may not start being transmitted in any interval where the gate of its class is closed. Thus, $p_{i,j}$ cannot be transmitted in any of the following intervals

$$\left\{ [0, go_1^{\mathcal{C}_a}) \cup \left\{ \bigcup_{1 \leq l < n_{\mathcal{C}_a}} [gc_l^{\mathcal{C}_a}, go_{l+1}^{\mathcal{C}_a}) \right\} \cup [gc_{n_{\mathcal{C}_a}}^{\mathcal{C}_a}, \infty) \right\}.$$

Furthermore, according to Rule R3, $p_{i,j}$ may not start being transmitted during the $C_{p_{i,j}} - \epsilon$ time units before the gate closes (where ϵ is an arbitrarily small value). Therefore, $p_{i,j}$ may not start being transmitted in the time intervals in the set $\bigcup_{1 \leq l \leq n_{\mathcal{C}_a}} (gc_l^{\mathcal{C}_a} - C_{p_{i,j}}, gc_l^{\mathcal{C}_a})$. The lengths of those intervals are maximized when $C_{p_{i,j}} = C^{\max}_{p_{i,j}}$. Therefore, $p_{i,j}$ may not be able to start being transmitted in any time interval in the set

$$\left\{ \begin{array}{c} [0, go_1^{\mathcal{C}_a}) \\ \cup \\ \bigcup\limits_{1 \leq l < n_{\mathcal{C}_a}} [gc_l^{\mathcal{C}_a}, go_{l+1}^{\mathcal{C}_a}) \\ \cup \\ [gc_{n_{\mathcal{C}_a}}^{\mathcal{C}_a}, \infty) \\ \cup \\ \bigcup\limits_{1 \leq l \leq n_{\mathcal{C}_a}} (gc_l^{\mathcal{C}_a} - C^{\max}_{p_{i,j}}, gc_l^{\mathcal{C}_a}) \end{array} \right\} = \left\{ \begin{array}{c} [0, go_1^{\mathcal{C}_a}) \\ \cup \\ \bigcup\limits_{1 \leq l < n_{\mathcal{C}_a}} (gc_l^{\mathcal{C}_a} - C^{\max}_{p_{i,j}}, go_{l+1}^{\mathcal{C}_a}) \\ \cup \\ (gc_{n_{\mathcal{C}_a}}^{\mathcal{C}_a} - C^{\max}_{p_{i,j}}, \infty) \end{array} \right\}$$

□

Let $gc^{\text{last}}_{p_{i,j}}$ be that last time instant before time $\hat{t}_{\mathcal{C}_a}(s)$ where $p_{i,j}$ may not be able

to be transmitted according to $\mathcal{G}^{\text{pos}}_{p_{i,j}}$, and let $go^{\text{next}}_{p_{i,j}}$ be the first time instant after $gc^{\text{last}}_{p_{i,j}}$ at which packet $p_{i,j}$ may start being transmitted according to $\mathcal{G}^{\text{pos}}_{p_{i,j}}$. That is, we have

$$gc^{\text{last}}_{p_{i,j}} = \max\{t \mid t \in \mathcal{G}^{\text{pos}}_{p_{i,j}} \wedge t \leq \hat{t}_{\mathcal{C}_a}(s)\} \tag{5.7}$$

$$go^{\text{next}}_{p_{i,j}} = \min\{t \mid t \notin \mathcal{G}^{\text{pos}}_{p_{i,j}} \wedge t > gc^{\text{last}}_{p_{i,j}}\} \tag{5.8}$$

Then, the following lemma holds.

Lemma 5.11. *If $p_{i,j} \in top_{\mathcal{C}_a}$ is at the top of $\mathcal{C}_a$'s FIFO queue, then the earliest time when $p_{i,j}$ is certainly ready and the egress is certainly available is*

$$t^{\text{ready}}_{p_{i,j}} = \max\{\hat{t}_{\mathcal{C}_a}(s),\ go^{\text{next}}_{p_{i,j}}\} \tag{5.9}$$

Proof. According to the TSN scheduling policy presented in Section 5.1, to be ready, a packet $p_{i,j}$ must respect Rules R1 to R3. By Lemma 5.9, the earliest time $p_{i,j}$ certainly respects R1 and the egress is certainly available is at time $\hat{t}_{\mathcal{C}_a}(s)$. If $\hat{t}_{\mathcal{C}_a}(s) \geq go^{\text{next}}_{p_{i,j}}$, then by definition of $go^{\text{next}}_{p_{i,j}}$ and Lemma 5.10, Rules R2 and R3 are certainly respected at $\hat{t}_{\mathcal{C}_a}(s)$. Thus, $p_{i,j}$ is certainly ready (i.e., it respects R1, R2 and R3), and the egress is certainly available. This proves the first case of Equation (5.9). If, on the other hand, $\hat{t}_{\mathcal{C}_a}(s) < go^{\text{next}}_{p_{i,j}}$, then by definition of $go^{\text{next}}_{p_{i,j}}$ and Lemma 5.10, $p_{i,j}$ may not be able to start to transmit, and thus may not respect Rules R2 and R3 until time $go^{\text{next}}_{p_{i,j}}$. Since by definition of $go^{\text{next}}_{p_{i,j}}$ and Lemma 5.10, Rules R2 and R3 are certainly respected at $go^{\text{next}}_{p_{i,j}}$, $go^{\text{next}}_{p_{i,j}}$ is the earliest time R1 to R3 are certainly respected and the egress is certainly available after $\hat{t}_{\mathcal{C}_a}(s)$. This prove the second case of Equation (5.9). □

Lemma 5.11 refers to a specific packet of class $\mathcal{C}_a$, but, by Lemma 5.4, we know that any packet in $top_{\mathcal{C}_a}$ may be at the top of $\mathcal{C}_a$'s FIFO queue. Thus, the earliest time when the packet at the top of $\mathcal{C}_a$'s FIFO queue is certainly ready and the egress is certainly available in abstract state s is given by

$$\max_{\forall p_{i,j} \in top_{\mathcal{C}_a}} \{t^{\text{ready}}_{p_{i,j}}\}.$$

Since according to the TSN scheduling policy, a packet is transmitted when at least one packet is ready and the egress is available, the earliest time the next packet of any class is certainly being transmitted on the egress in abstract state s is

$$t^{\text{ub}}_{p_{i,j}} = \min_{\forall\, \mathcal{C}_a \in \mathcal{C}} \left\{ \max_{\forall p_{i,j} \in top_{\mathcal{C}_a}} \{t^{\text{ready}}_{p_{i,j}}\} \right\}. \tag{5.10}$$

We state now our main result about $t^{\text{ub}}_{p_{i,j}}$.

Lemma 5.12. *If packet $p_{i,j}$ is the next packet transmitted in a concrete state represented by the abstract state s, then $p_{i,j}$ is transmitted no later than $t^{\mathrm{ub}}_{p_{i,j}}$.*

Proof. Since there is certainly a packet transmitted at $t^{\mathrm{ub}}_{p_{i,j}}$, if $p_{i,j}$ is the next packet transmitted, then it must be transmitted at or before $t^{\mathrm{ub}}_{p_{i,j}}$. □

Packet's potential eligibility

We say that a packet $p_{i,j}$ is potentially eligible in abstract state s if it may be the next packet transmitted in abstract state s. Lemma 5.13 below proves a condition that must be respected for a packet $p_{i,j}$ to be potentially eligible.

Lemma 5.13. *Packet $p_{i,j}$ may be the next packet transmitted in abstract state s only if $t^{\mathrm{lb}}_{p_{i,j}} \leq t^{\mathrm{ub}}_{p_{i,j}}$.*

Proof. According to Lemma 5.8, $p_{i,j}$ cannot be transmitted earlier than $t^{\mathrm{lb}}_{p_{i,j}}$ in abstract state s. Moreover, according to Lemma 5.12, $p_{i,j}$ must start being transmitted before $t^{\mathrm{ub}}_{p_{i,j}}$ if it is the next packet transmitted in abstract state s. Thus, $p_{i,j}$ cannot be the next packet transmitted in abstract state s if $t^{\mathrm{lb}}_{p_{i,j}} > t^{\mathrm{ub}}_{p_{i,j}}$. Taking the contrapositive of the last statement proves the lemma. □

5.3.3 Eligible transmission intervals of potential eligible packets

In this section, we elaborate on how to find the eligible transmission intervals of potentially eligible packets in abstract state s. Let $p_{i,j}$ be a potentially eligible packet of class $\mathcal{C}_a$. $t^{\mathrm{lb}}_{p_{i,j}}$ and $t^{\mathrm{ub}}_{p_{i,j}}$ provide lower and upper bounds on the time interval within which $p_{i,j}$ may possibly start being transmitted in abstract state s. However, $p_{i,j}$ cannot start being transmitted at every time instance within $[t^{\mathrm{lb}}_{p_{i,j}}, t^{\mathrm{ub}}_{p_{i,j}}]$ because this interval does not fully consider the shaping imposed by TAS on the gate of class $\mathcal{C}_a$, nor does it account for interference from ready packets of higher-priority classes. Lemmas 5.14 and 5.15 account for the former, and Lemma 5.16 accounts for the latter.

Lemma 5.14. *A packet $p_{i,j}$ can certainly not start its transmission in the set of time intervals*

$$\mathcal{G}^{\mathrm{cert}}_{p_{i,j}} = \left\{ [0, go^{\mathcal{C}_a}_1) \cup \left\{ \bigcup_{1 \leq l < n_{\mathcal{C}_a}} (gc^{\mathcal{C}_a}_l - C^{\min}_{p_{i,j}}, go^{\mathcal{C}_a}_{l+1}) \right\} \cup (gc^{\mathcal{C}_a}_{n_{\mathcal{C}_a}} - C^{\min}_{p_{i,j}}, \infty) \right\} \tag{5.11}$$

Proof. According to Rule R2, packet $p_{i,j}$ may never start being transmitted in any interval where the gate of its class is closed. Thus, $p_{i,j}$ cannot transmit in any

of the intervals $\left\{[0, go_1^{\mathcal{C}_a}) \cup \left\{\bigcup_{1\leq l<n_{\mathcal{C}_a}}[gc_l^{\mathcal{C}_a}, go_{l+1}^{\mathcal{C}_a})\right\} \cup [gc_{n_{\mathcal{C}_a}}^{\mathcal{C}_a}, \infty)\right\}$. Furthermore, according to Rule R3, $p_{i,j}$ may not start being transmitted during the $C_{p_{i,j}}$ time units before the gate closes. Therefore, $p_{i,j}$ may not start being transmitted in the time intervals in the set $\bigcup_{1\leq l\leq n_{\mathcal{C}_a}}(gc_l^{\mathcal{C}_a} - C_{p_{i,j}}, gc_l^{\mathcal{C}_a})$. The lengths of those intervals is minimized when $C_{p_{i,j}} = C_{p_{i,j}}^{\min}$. Therefore, $p_{i,j}$ can *certainly* not start transmitting in the union of the above two sets of time intervals, thus proving the lemma. □

Lemma 5.15. *A potentially eligible packet $p_{i,j}$ in abstract state s may only start its transmission at the time instants in the set*

$$\left\{\left[t_{p_{i,j}}^{\text{lb}}, t_{p_{i,j}}^{\text{ub}}\right]\right\} \setminus \mathcal{G}_{p_{i,j}}^{\text{cert}}$$

Proof. Lemmas 5.8 and 5.12 prove that, if $p_{i,j}$ is the next packet to be transmitted in abstract state s, then it cannot start transmitting outside the time interval $\left[t_{p_{i,j}}^{\text{lb}}, t_{p_{i,j}}^{\text{ub}}\right]$. Furthermore, Lemma 5.14 proves that $p_{i,j}$ can certainly not start being transmitted in any interval in the set $\mathcal{G}_{p_{i,j}}^{\text{cert}}$. Thus, $p_{i,j}$ may only start transmitting at time instants in the set $\{\left[t_{p_{i,j}}^{\text{lb}}, t_{p_{i,j}}^{\text{ub}}\right]\} \setminus \mathcal{G}_{p_{i,j}}^{\text{cert}}$. □

Lemma 5.16. *Let $p_{i,j}$ be the packet in $top_{\mathcal{C}_b}$ with the largest possible transmission time, i.e., $p_{i,j} = \underset{\forall p_{i,k}\in top_{\mathcal{C}_b}}{\arg\max}\{C_{p_{i,k}}^{\max}\}$. Then, there is certainly a packet of class $\mathcal{C}_b$ that is ready for transmission in all time intervals in the set*

$$\mathcal{I}_{\mathcal{C}_b}^{\text{hp}} = [tr_{\mathcal{C}_b}^{\max}, \infty) \setminus \mathcal{G}_{p_{i,j}}^{\text{pos}}. \tag{5.12}$$

Proof. A packet is ready if it respects Rules R1, R2 and R3. According to Lemma 5.4, $top_{\mathcal{C}_b}$ contains all packets of class $\mathcal{C}_b$ that may be at the top of $\mathcal{C}_b$'s FIFO queue in abstract state s. Thus, only packets in $top_{\mathcal{C}_b}$ may respect R1. Lemma 5.3 proves that there is certainly a packet at the top of the FIFO queue at time $tr_{\mathcal{C}_b}^{\max}$. Thus, R_1 is certainly respected at and after $tr_{\mathcal{C}_b}^{\max}$. Lemma 5.10 proves that $\mathcal{G}_{p_{i,j}}^{\text{pos}}$ contains all time intervals in which a packet P_l may not respect R2 and/or R3. According to Equation (5.6), $\mathcal{G}_{p_{i,j}}^{\text{pos}}$ contains the largest intervals when $C_{p_{i,j}}^{\max}$ is maximized. Thus, $\mathcal{G}_{p_{i,j}}^{\text{pos}}$ contains the largest intervals in which R2 and/or R3 is not respected for the packet $p_{i,j}$ as defined in the claim. Thus, the set of time intervals during which R1, R2 and R3 are certainly respected is obtained when $p_{i,j}$ is at the top of $\mathcal{C}_b$'s FIFO queue, and is given by $[tr_{\mathcal{C}_b}^{\max}, \infty) \setminus \mathcal{G}_{p_{i,j}}^{\text{pos}}$. □

Combining Lemmas 5.15 and 5.16, we build the set $\mathcal{E}_{p_{i,j}}$ that contains all

eligible transmission intervals of a potentially eligible packet $p_{i,j}$.

$$\mathcal{E}_{p_{i,j}} = \left\{ \left[t^{\text{lb}}_{p_{i,j}}, t^{\text{ub}}_{p_{i,j}} \right] \setminus \mathcal{G}^{\text{cert}}_{p_{i,j}} \right\} \setminus \left\{ \bigcup_{\forall \mathcal{C}_b \in \mathcal{C}_a | \pi_{\mathcal{C}_b} < \pi_{\mathcal{C}_a}} \mathcal{I}^{\text{hp}}_{\mathcal{C}_b} \right\} \tag{5.13}$$

EXAMPLE 5.1

Let us consider 3 packets p_1, p_2 and p_3 that are yet to be dispatched at state s, with parameters as given in Figure 5.3. Assume the egress becomes possibly available at time 2 and certainly available at time 22. Assume all three packets are on the top of their respective queues. Figure 5.3 also shows when the gates of each class is closed. For this example, we aim to calculate the eligible transmission intervals of p_3 i.e., $\mathcal{E}_{p_3}$.
The earliest time packet p_3 can possibly start being transmitted is when it is arrived and the egress is possibly available (as computed by Equation (5.4)). This happens at time 3. Moreover, if p_3 is the first packet being transmitted in state s, it must certainly have started being transmitted no later than when the egress is certainly available, and there certainly is a packet in any queue with an open gate for which the guard band did not start yet (as computed by Equation (5.10)). That happens at time 22 as the egress is certainly available and p_1, p_2 and p_3 are all certainly arrived in their queue with an open gate for which the guard band did not start. Thus, p_3 will start being transmitted in the interval [3, 22], only if p_3's gate is open, the guard-band is not started and no higher priority packet is eligible to be transmitted.
As captured by $\mathcal{G}^{\text{cert}}_{p_3}$ (shaded region on p_3's timeline) calculated by Lemma 5.14, p_3 can certainly not transmit in the interval (5,14) as either its gate is closed or the guard band started. On the other hand, classes $\mathcal{C}_1$ and $\mathcal{C}_2$ can certainly start transmitting a packet at or after time 22 and 16, respectively, since there is certainly a packet in their queue and their gate is open. It is captured by $\mathcal{I}^{\text{hp}}_{\mathcal{C}_1}$ and $\mathcal{I}^{\text{hp}}_{\mathcal{C}_2}$ calculated by Lemma 5.16 and depicted by the shaded regions on the timelines of packets p_1 and p_2. Since p_3 cannot start when higher priority classes are able to transmit, the eligible transmission intervals of p_3 are $[3, 5]$ and $[14, 16)$ (i.e., $[3, 22] \setminus (5, 14) \setminus [22, \infty) \setminus [16, \infty)$ calculated by Equation (5.13)).

We now prove that a packet $p_{i,j}$ may start transmitting at a time instant t if an only if $t \in \mathcal{E}_{p_{i,j}}$.

Lemma 5.17. *There is a concrete state represented by abstract state s in which a packet $p_{i,j} \in top_{\mathcal{C}_a}$ can start its transmission at any time instant $t \in \mathcal{E}_{p_{i,j}}$.*

Proof. For a packet $p_{i,j} \in \mathcal{C}_a$ to possibly start its transmission at time t, it must

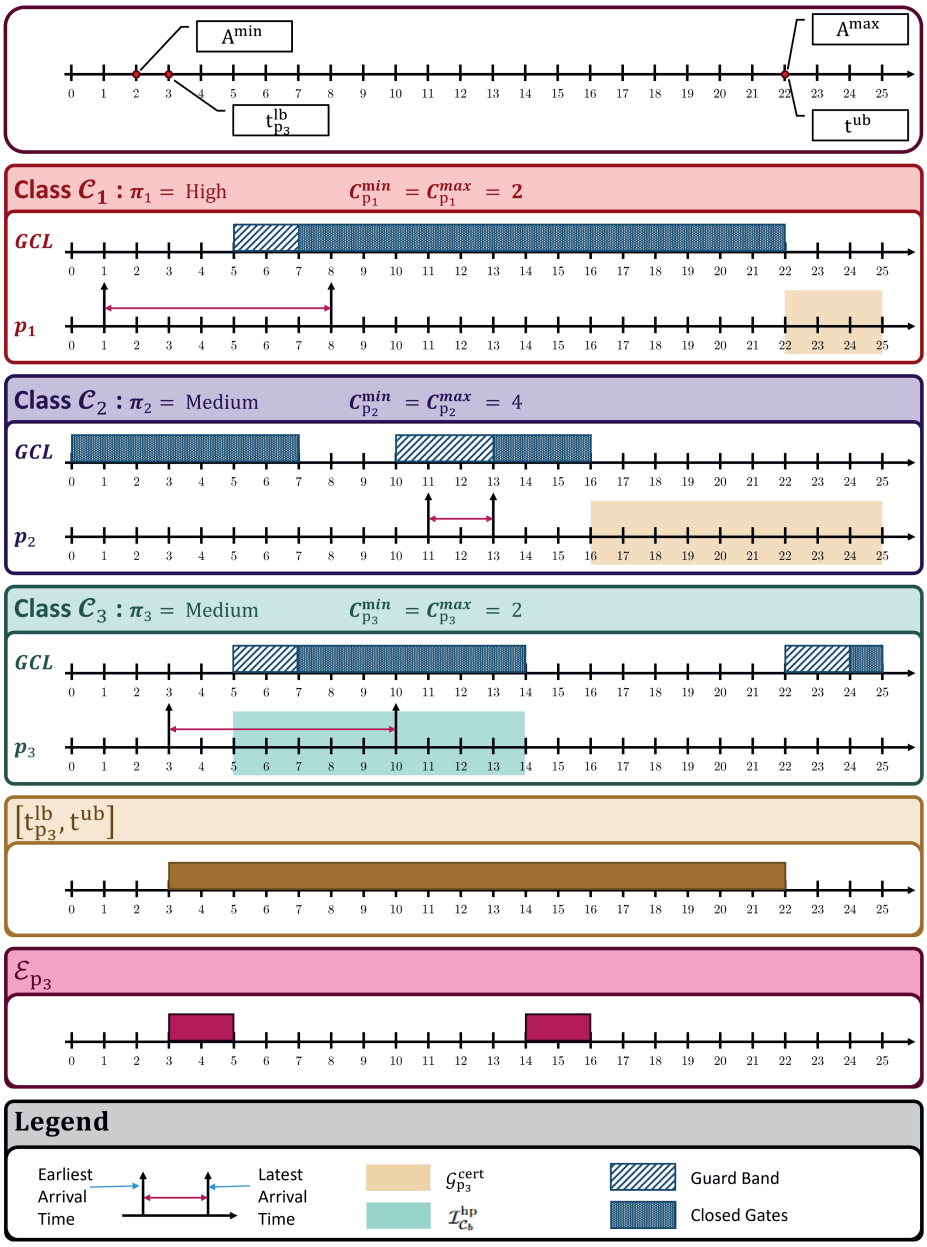


Figure 5.3: Example that shows how eligible transmission intervals are calculated for packet p_3

meet the following conditions: (i) it may be ready at t, (ii) all higher priority packets may not be ready at t, and (iii) the egress may be available at t.

By Equation (5.13), $\mathcal{E}_{p_{i,j}}$ only contains time instants at or later than $t^{\text{lb}}_{p_{i,j}}$. Since, by Equation (5.4), $t^{\text{lb}}_{p_{i,j}} \geq A^{\min}$, Condition (iii) is satisfied for every time instant $t \in \mathcal{E}_{p_{i,j}}$.

Now, we prove that (i) is respected. To be ready, and thus meet Condition (i), a packet $p_{i,j}$ must satisfy R1, R2 and R3. Since $p_{i,j} \in top_{\mathcal{C}_a}$, it satisfies R1. Since, by Lemma 5.14, the set $\mathcal{G}^{\text{cert}}_{p_{i,j}}$ contains all the time instants where R2 and/or R3 is certainly not respected in all transmission scenarios, by taking the contra-positive, there is at least one transmission scenario where $p_{i,j}$ respects R2 and R3 for every time instant $t \notin \mathcal{G}^{\text{cert}}_{p_{i,j}}$. Since, by Equation (5.13), $\mathcal{E}_{p_{i,j}}$ only contains time instants that are not in $\mathcal{G}^{\text{cert}}_{p_{i,j}}$, Condition (i) is met.

Similarly, since $\mathcal{I}^{\text{hp}}_{\mathcal{C}_b}$ contains all instants when a packet of class $\mathcal{C}_b$ is certainly ready, and because Equation (5.13) removes all such instants from $\mathcal{E}_{p_{i,j}}$ for all classes with higher priority than $p_{i,j}$, Condition (ii) is met.

Thus, Conditions (i), (ii) and (ii) are met for all $t \in \mathcal{E}_{p_{i,j}}$, and thus $p_{i,j}$ can possibly start at any time instant in $\mathcal{E}_{p_{i,j}}$. □

Lemma 5.18. *There is no concrete state represented by abstract state s where a packet $p_{i,j} \in top_{\mathcal{C}_a}$ can start its transmission at any time instant $t \notin \mathcal{E}_{p_{i,j}}$.*

Proof. Let us assume that packet $p_{i,j} \in \mathcal{C}_a$ begins its transmission at time t such that $t \notin \mathcal{E}_{p_{i,j}}$. Then, Conditions (i), (ii) and (iii) as stated in Lemma 5.17 must be satisfied at t.

We analyse three cases: $t < t^{\text{lb}}_{p_{i,j}}$, $t > t^{\text{ub}}_{p_{i,j}}$ or $t \in [t^{\text{lb}}_{p_{i,j}}, t^{\text{ub}}_{p_{i,j}}]$.

Case 1: If $t < t^{\text{lb}}_{p_{i,j}}$ then, by Equation (5.4), $t < A^{\min}$ or $t < a^{\min}_{p_{i,j}}$. If the former is true, the egress cannot be available at t, contradicting Condition (iii). If the latter is true, then $p_{i,j}$ cannot have arrived in the switch yet and thus cannot be ready, contradicting Condition (i).

Case 2: If $t > t^{\text{ub}}_{p_{i,j}}$, Lemma 5.12 proves that $p_{i,j}$ cannot be the next packet transmitted in abstract state s. Thus, either Condition (i) or (ii) is not respected.

Case 3: If $t \in [t^{\text{lb}}_{p_{i,j}}, t^{\text{ub}}_{p_{i,j}}]$ and $t \notin \mathcal{E}_{p_{i,j}}$, then, by Equation (5.13), $t \in \mathcal{G}^{\text{cert}}_{p_{i,j}}$ or $\exists \mathcal{C}_b$ such that $t \in \mathcal{I}^{\text{hp}}_{\mathcal{C}_b}$ and the priority of $\mathcal{C}_b$ is higher than $p_{i,j}$'s priority. If the former is true, then Lemma 5.14 proves that $p_{i,j}$ is not ready at t, contradicting Condition (i). If the latter is true, then Lemma 5.16 proves that a higher priority packet of class $\mathcal{C}_b$ is certainly ready, contradicting Condition (ii).

Therefore, we reached a contradiction for all cases. Meaning that if $p_{i,j}$ is the next packet transmitted in abstract state s, $p_{i,j}$ cannot start transmitting at any time instant $t \notin \mathcal{E}_{p_{i,j}}$. □

Theorem 5.19. *$\mathcal{E}_{p_{i,j}}$ contains all the instants at which $p_{i,j} \in top_{\mathcal{C}_a}$ may start transmitting and does not contain any instant at which $p_{i,j}$ may not start transmitting in a concrete state represented by abstract state s.*

Proof. It is a direct application of Lemmas 5.17 and 5.18. □

5.4 Expansion Phase

As a result of Theorem 5.19, if $\mathcal{E}_{p_{i,j}}$ is not empty, then packet $p_{i,j}$ may be the next packet transmitted through the egress in abstract state s. $p_{i,j}$'s transmission makes the system evolve to a new state. Since the availability of the egress in the new abstract state s' depends on when $p_{i,j}$ starts transmitting. Since, according to our discussion in the previous section, there may be more than one continuous time interval in which $p_{i,j}$ may start transmitting (i.e., there may be more than one time interval in $\mathcal{E}_{p_{i,j}}$), Algorithm 3 analyses each continuous time interval in which $p_{i,j}$ may start separately, i.e., a new abstract state s' encoding the new abstract state after transmitting $p_{i,j}$ is created for every interval in $\mathcal{E}_{p_{i,j}}$.

We denote the start of the k^{th} time interval in $\mathcal{E}_{p_{i,j}}$ by $\mathcal{E}^{k,\text{start}}_{p_{i,j}}$, and its end by $\mathcal{E}^{k,\text{end}}_{p_{i,j}}$. By Theorem 5.19, $p_{i,j}$ may start transmitting at any time within $[\mathcal{E}^{k,\text{start}}_{p_{i,j}}, \mathcal{E}^{k,\text{end}}_{p_{i,j}}]$. Thus, $p_{i,j}$'s transmission may end at any time within

$$[\mathcal{E}^{k,\text{start}}_{p_{i,j}} + C_{p_{i,j}},\ \mathcal{E}^{k,\text{end}}_{p_{i,j}} + C_{p_{i,j}}]$$

We also know from the definition of the guard-band that $p_{i,j}$ certainly finished its transmission when the gate of its class closes (see rule R3). Therefore, the latest time $p_{i,j}$ may finish its transmission when starting its transmission in the k^{th} interval in $\mathcal{E}_{p_{i,j}}$ is either when $p_{i,j}$ starts at $\mathcal{E}^{k,\text{end}}_{p_{i,j}}$ and transmits for its maximum transmission time $C^{\max}_{p_{i,j}}$, or when the gate of $p_{i,j}$'s class (say $\mathcal{C}_a$) closes for the first time after $p_{i,j}$ started transmitting. That is, it is given by $LFT_{p_{i,j}}$ as defined in Equation (5.14).

$$LFT_{p_{i,j}} = \min\{\mathcal{E}^{k,\text{end}}_{p_{i,j}} + C^{\max}_{p_{i,j}},\ gc^{\text{next}}_{p_{i,j}}\} \tag{5.14}$$

where $gc^{\text{next}}_{p_{i,j}} = \min\{t \mid t \notin \mathcal{G}_{\mathcal{C}_a} \wedge t > \mathcal{E}^{k,\text{end}}_{p_{i,j}}\}$.

Similarly, the earliest time $p_{i,j}$ may finish its transmission when starting its transmission in the k^{th} interval in $\mathcal{E}_{p_{i,j}}$ is when it starts at $\mathcal{E}^{k,\text{start}}_{p_{i,j}}$ and transmits for its minimum transmission time.

$$EFT_{p_{i,j}} = \mathcal{E}^{k,\text{start}}_{p_{i,j}} + C^{\min}_{p_{i,j}} \tag{5.15}$$

Using the earliest and latest finish time of $p_{i,j}$, we get that the earliest and

latest availability time of the egress $A^{\min}$ and $A^{\max}$ after transmitting $p_{i,j}$ are given by Equations (5.16) and (5.17), respectively.

$$A^{\min} = EFT_{p_{i,j}} + C_{\mathsf{IPG}} \tag{5.16}$$

$$A^{\max} = LFT_{p_{i,j}} + C_{\mathsf{IPG}} \tag{5.17}$$

where C_{IPG} is the length of the inter-packet gap, i.e., the minimum time duration defined by the Ethernet protocol that must separate the transmission of any two packets.

Lemma 5.20. *The $LFT_{p_{i,j}}$ and $EFT_{p_{i,j}}$ calculated with Equations* (5.14) *and* (5.15) *are exact lower and upper-bounds on the finish time of packet $p_{i,j}$ if $p_{i,j}$ starts transmitting in the k^{th} interval of $\mathcal{E}_{p_{i,j}}$.*

Proof. To prove a bound is exact, we must prove that there is at least one transmission scenario such that the bound is reached, and that no transmission scenario may go beyond the bound.

We first prove the claim for $EFT_{p_{i,j}}$. Since $C^{\min}_{p_{i,j}}$ is a valid transmission time for $p_{i,j}$, if $p_{i,j}$ starts at $\mathcal{E}^{k,\text{start}}_{p_{i,j}} \in \mathcal{E}_{p_{i,j}}$ and executes for $C^{\min}_{p_{i,j}}$ it finishes at $EFT_{p_{i,j}}$ as defined by Equation (5.15). Furthermore, to finish earlier than $EFT_{p_{i,j}} = \mathcal{E}^{k,\text{start}}_{p_{i,j}} + C^{\min}_{p_{i,j}}$, $p_{i,j}$ must either start earlier than $\mathcal{E}^{k,\text{start}}_{p_{i,j}}$ or transmit for less than $C^{\min}_{p_{i,j}}$. Any of those would either contradict that $\mathcal{E}^{k,\text{start}}_{p_{i,j}}$ is the start of $\mathcal{E}_{p_{i,j}}$, or that $C^{\min}_{p_{i,j}}$ is a lower bound on $p_{i,j}$'s transmission time.

We now prove the claim for $LFT_{p_{i,j}}$. Since $C^{\max}_{p_{i,j}}$ and lower values are valid values for the transmission time of $p_{i,j}$, if $p_{i,j}$ starts at $\mathcal{E}^{k,\text{end}}_{p_{i,j}} \in \mathcal{E}_{p_{i,j}}$ and executes for $C_{p_{i,j}} = LFT_{p_{i,j}} - \mathcal{E}^{k,\text{end}}_{p_{i,j}} \leq C^{\max}_{p_{i,j}}$, it finishes at $LFT_{p_{i,j}}$ as defined by Equation (5.14). Furthermore, to finish later than $LFT_{p_{i,j}} = \min\{\mathcal{E}^{k,\text{end}}_{p_{i,j}} + C^{\max}_{p_{i,j}},\ gc^{\text{next}}_i\}\}$, $p_{i,j}$ must either start later than $\mathcal{E}^{k,\text{end}}_{p_{i,j}}$ or transmit for more than $C^{\max}_{p_{i,j}}$ or continues to transmit until later than gc^{next}_i. Any of those would either contradict that $\mathcal{E}^{k,\text{end}}_{p_{i,j}}$ is the end of $\mathcal{E}_{p_{i,j}}$, or that $C^{\max}_{p_{i,j}}$ is an upper bound on $p_{i,j}$'s transmission time, or that rule R3 of TSN does not allow a packet to start transmitting if it would finish later than the next closing event of $p_{i,j}$'s gate, i.e., after gc^{next}_i. □

Corollary 5.21. *$A^{\min}$ and $A^{\max}$ calculated with Equation* (5.16) *and* (5.17) *are exact lower and upper-bounds on the availability time of the egress if $p_{i,j}$ start transmitting in the j^{th} interval of $\mathcal{E}_{p_{i,j}}$.*

5.5 Merge Phase

As said in Section 5.2, to curb the growth of the graph, if there are two abstract states s_1 and s_2 with the same set of transmitted packets $\mathcal{P}(s_1) = \mathcal{P}(s_2)$, and with

overlapping or contiguous availability intervals, Algorithm 3 merges s_1 and s_2 into an equivalent state s'

Lemma 5.22. *Let s_1 and s_2 be two states. If $\mathcal{P}(s_1) = \mathcal{P}(s_2)$ and $A^{\min}(s_1) \leq A^{\min}(s_2) \leq A^{\max}(s_1)$, then a state s' with $\mathcal{P}(s') = \mathcal{P}(s_1) = \mathcal{P}(s_2)$, and $A^{\min}(s') = A^{\min}(s_1)$ and $A^{\max}(s') = \max\{A^{\max}(s_1), A^{\max}(s_2)\}$ covers all transmission scenarios covered by s_1 and s_2 and no transmission scenario that is not covered by either s_1 or s_2.*

Proof. We first prove that all transmission scenarios covered by s' are also covered by either s_1 or s_2. If $A^{\min}(s_1) \leq A^{\min}(s_2) \leq A^{\max}(s_1)$, then the time interval $[A^{\min}(s_1), A^{\max}(s_1)) \cup [A^{\min}(s_2), A^{\max}(s_2))$ is a continuous interval. Thus, there is at least one transmission scenario covered by s_1 and/or s_2 in which all packets in $\mathcal{P}(s_1) = \mathcal{P}(s_2)$ are transmitted and the egress becomes available at any time $t \in [A^{\min}(s_1),\ \max\{A^{\max}(s_1), A^{\max}(s_2)\})$.

We now prove that all transmission scenarios covered by s_1 and s_2 are covered by s'. In all transmission scenarios covered by s_1 and s_2, the egress is certainly available at time $\max\{A^{\max}(s_1), A^{\max}(s_2)\}$, and in no transmission scenario covered by s_1 and s_2, the egress becomes available before $A^{\min}(s_1)$. Since s' covers all scenario where the egress may become available from $A^{\min}(s') = A^{\min}(s_1)$ to $A^{\max}(s') = \max\{A^{\max}(s_1), A^{\max}(s_2)\}$, it covers all transmission scenarios covered by s_1 and s_2. □

5.6 Proof of Correctness

In this section, we establish that Algorithm 3 analyses all possible transmission scenarios and returns exact response time bounds for every packet in $\mathcal{P}$.

Lemma 5.23. *If Algorithm 3 does not create an edge labelled with packet $p_{i,j}$ from abstract state s, then there exists no transmission scenario where $p_{i,j}$ is the next packet transmitted in s.*

Proof. If Algorithm 3 does not create an edge labelled with packet $p_{i,j}$ from abstract state s, then either $p_{i,j}$ is not possibly eligible, i.e., $t^{\text{lb}}_{p_{i,j}} > t^{\text{ub}}_{p_{i,j}}$ (line 11 of Algorithm 3) or the set of eligible transmission intervals $\mathcal{E}_{p_{i,j}}$ is empty (line 13 of Algorithm 3). By contradiction, assume that there exists an transmission scenario where $p_{i,j}$ is the next packet transmitted in abstract state s and $t^{\text{lb}}_{p_{i,j}} > t^{\text{ub}}_{p_{i,j}}$ and/or $\mathcal{E}_{p_{i,j}} = \emptyset$. Then, it contradicts Lemma 5.13 and/or Theorem 5.19. □

Lemma 5.24. *If Algorithm 3 creates an edge labelled with packet $p_{i,j}$ originating from abstract state s, then there exists a transmission scenario where $p_{i,j}$ is the next packet transmitted in s.*

Proof. Algorithm 3 creates an edge labelled with packet $p_{i,j}$ originating from abstract state s when $t^{\text{lb}}_{p_{i,j}} \leq t^{\text{ub}}_{p_{i,j}}$ (line 11 of Algorithm 3) and $\mathcal{E}_{p_{i,j}} \neq \emptyset$ (line 13 of Algorithm 3). If $\mathcal{E}_{p_{i,j}} \neq \emptyset$ then we always have $t^{\text{lb}}_{p_{i,j}} \leq t^{\text{ub}}_{p_{i,j}}$ by Equation (5.13), thereby making the condition $t^{\text{lb}}_{p_{i,j}} \leq t^{\text{ub}}_{p_{i,j}}$ irrelevant to prove the claim. Moreover, Theorem 5.19 proves that $\mathcal{E}_{p_{i,j}}$ does not contain any instant at which $p_{i,j}$ cannot start transmitting in s. Therefore, if $\mathcal{E}_{p_{i,j}} \neq \emptyset$, there is an transmission scenario where $p_{i,j}$ starts transmitting in s. □

Theorem 5.25. *The SAG generated by Algorithm 3 contains all reachable abstract states and the response time bounds calculated by Algorithm 3 are exact.*

Proof. We prove the claim by proving that the SAG generated by Algorithm 3 contains all states reachable by the system and does not contain any state that is not reachable by the system. We prove it by induction. Furthermore, for each abstract state, we prove that the bounds computed by Algorithm 3 on the finish times of packets are exact.

Base case: the state v_0 where the egress is free and no packet is transmitted yet is the only possible abstract state when the system starts and Algorithm 3 properly adds it to the SAG.

Induction step: Let s be a reachable abstract state that was already added to the SAG by Algorithm 3, Lemmas 5.23 and 5.24 prove that Algorithm 3 adds a new state v_l connected to s by an edge labelled with $p_{i,j}$ if and only if there is an transmission scenario where $p_{i,j}$ starts executing in s. Moreover, Lemma 5.20 and Corollary 5.21 prove that the bounds computed by Algorithm 3 on the finish time of $p_{i,j}$ and the resulting availability of the egress are exact. Thus, Algorithm 3 only adds a state v_l if it is reachable by the system, and all states reachable from s are added to the SAG by Algorithm 3.

Applying the induction step until Algorithm 3 ends proves the theorem. □

5.7 Evaluation

The analysis we proposed in the previous sections can analyse TSN switches configured such that gates controlled by Time-Aware Shapers (TAS) adhere to any imaginable schedule, including those where multiple gates are open simultaneously, a flexibility that goes beyond what existing response time analyses for TAS can support. Therefore, we claim that the strengths of our analysis are: (1) its flexibility, enabling design exploration of unconventional configurations, and (2) its exactness. We already proved the exactness of the analysis in Section 5.6. Therefore, in this section, we investigate whether stepping outside conventional boundaries on how TAS gates are configured might bring new benefits since

one may question the necessity of such a comprehensive analysis for such a well-studied concept as TAS.

5.7.1 Experimental Setup

We generate systems made of 8 traffic classes (i.e., the default number of classes mentioned in the TSN standard). We assume each traffic class comprises f flows where $f = 2$, $f = 4$, or $f = 8$ depending on the experiments (i.e., there are between 16 and 64 flows equally distributed between the 8 traffic classes). For each experiment, we start by randomly generating a Gate Control List (GCL) for the time-aware shapers. To do so, for each traffic class $\mathcal{C}_a \in \mathcal{C}$, we generate a utilization values U_a using the Emberson and Davis' tool [25] such that the sum of the classes utilization $\sum_{\mathcal{C}_a \in \mathcal{C}} U_a$ is equal to 100% of the switch egress. That is, U_a represents the portion of the egress bandwidth reserved for class $\mathcal{C}_a$. Then, for each class $\mathcal{C}_a \in \mathcal{C}$, we randomly pick a period T_a from the log-uniformly distributed set of values $\{x \times 10^y \mid x \in [1, 9] \wedge y \in [2, 4]\}$. The total duration O_a during which the gate of class $\mathcal{C}_a$ is open within each period of length T_a is then calculated as $O_a = U_a \times T_a$. Given that a class comprises f communication flows, we assume the gate of each class $\mathcal{C}_a$ opens f times within each period T_a, with each gate opening lasting $\frac{O_a}{f}$ time.

We then use the Critical Window Earliest Deadline First (CW-EDF) scheduling algorithm [47, 45] to build the time-triggered schedule deciding the exact times at which each gate of each class is opened and closed over the hyperperiod of the traffic classes. This forms our baseline GCL. Note that CW-EDF enforces that no two gates are open simultaneously.

To test alternative non-conventional GCL configurations, we construct variations of the baseline GCL. We adjust the opening times of each gate in the GCL so that it opens when the previous gate in the GCL has not closed yet. We investigate three distinct scenarios, setting the the time two gates are open simultaneously at 50%, 99%, and 100% of the duration of the previous gate's opening. An example of how the GCL would look like after a 50% overlap is added is shown in the bottom part of Figure 5.4. The initial GCL before adding the 50% overlap is shown in the upper part of the figure.

In our experiments, we assume the communication flows share the same period as their traffic class. To model the fact that communication flow of a class rarely use 100% of the bandwidth reserved for that class, we distribute 80% of each class bandwidth between the flows of that class using Emberson and Davis' tool [25] and enforcing that the maximum transmission time of each packet does go over the time the class' gate opens. We model the variation in transmission times of the communication flow by assuming that the minimum transmission

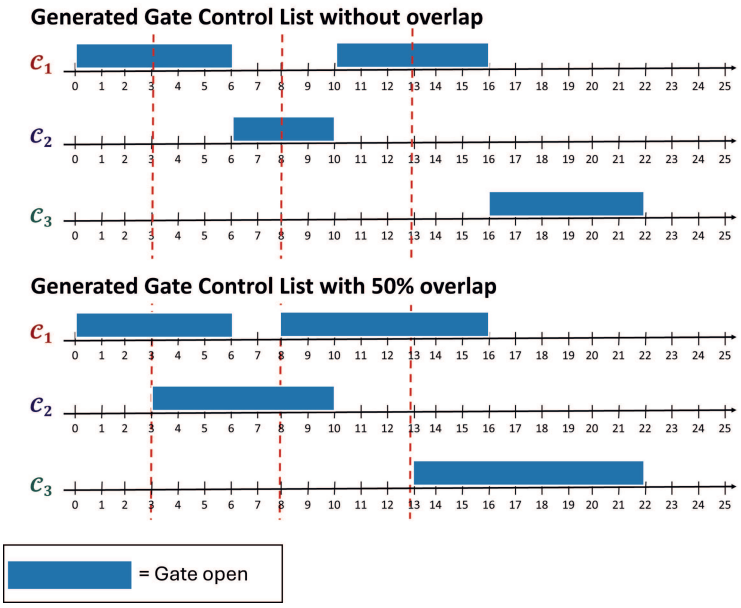


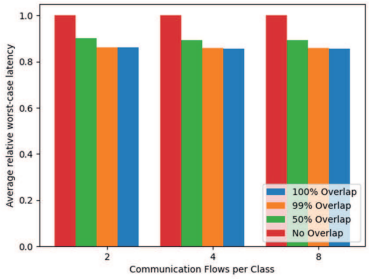
Figure 5.4: Example that shows how overlapping gates are configured at 50% overlap

time is set to 50% of its maximum transmission time.

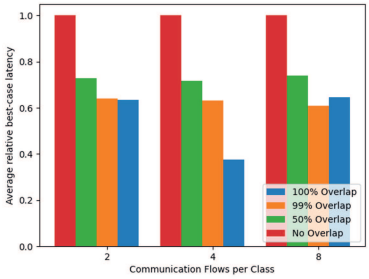
5.7.2 Results

Our experiments are designed to explore whether allowing gates to overlap can lead to lower latency for both best-case and worst-case scenarios in packet transmission. The underlying idea is that in situations where there is uncertainty, be it in the timing of packet arrivals at the switch ingress or in the length of their transmission times, overlapping gates could help some classes benefit from the reserved bandwidth of other classes that might otherwise go unused. By potentially starting the transmission of a packet of a class sooner as another class transmission is concluding, we expect to see a significant drop in the overall time packets spend in the switch queues. To test this hypothesis, we examine how the worst-case and best-case latency change when gates are allowed to overlap.

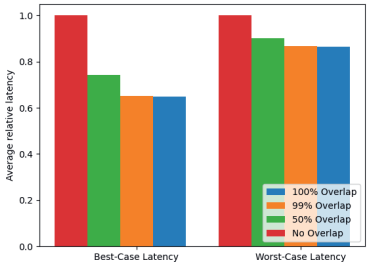
Figure 5.5(a)-(d) shows the impact of gate overlap on the latency of packets transmitted through a switch. We investigated two distinct scenarios: the first involves communication flows with varying transmission times but no release jitter (Figure 5.5(a)-(b)), while the second adds an additional layer of complexity by adding release jitter equal to 5% of each flow's period (Figure 5.5(c)-(d)). Figure 5.5(a)-(b) further show the impact of the number of communication flows per traffic class on latency. The plots of Figure 5.5 are the result of running 40



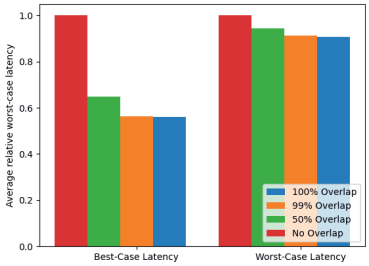
(a) Relative worst-case latency under transmission time variation



(b) Relative best-case latency under transmission time variation



(c) Relative best- and worst-case latency when there is release jitter for 2 flows per traffic class



(d) Relative best- and worst-case latency when there is release jitter for 4 flows per traffic class

Figure 5.5: Packets' latency comparison for different gate overlap scenarios.

experiments per scenario. The latency is plotted relative to the latency in the baseline scenario with no overlap, that is, if the average best-case latency is 0.5 for a given scenario, then it means that, in average, the best-case latency of packets was half the best-case latency of the same packets in comparison to the scenario without gate overlap.

The results across all scenarios are consistent: introducing gate overlap significantly reduces the best-case latency of packets (Figure 5.5(b)-(d)), but probably more surprisingly, it also improves the worst-case latency of packets (Figure 5.5(a),(c) and (d)). These results are the first that allow to formally evaluate the cost of using strictly non-overlapping GCLs that under-utilizes the available bandwidth, thereby particularly impacting best-case performance.

While we see a clear improvement of the average latency across all classes, we should consider that TAS is usually used to shape traffic classes that deal with high priority or control traffic that cannot tolerate interference by other traffic classes. Strict non-overlapping gates are a way to ensure the absence of such interference. By allowing gate overlaps across traffic classes, we bring the risk to lose that expected property of TAS. Thus, we tested the impact of overlapping gates on

the potential additional interference between classes by checking whether the worst-case latency of packets may increase when gate overlaps are allowed. The result of our tests showed that none of the packets experienced a missed deadline even when gates overlap by 100%.

One concern with overlapping gates is that they introduce greater variability in the response times of TAS packets, which is undesirable. Increased variability in a packet's departure from one switch can lead to larger uncertainty bounds on the arrival times at the next switch in the network. The higher the uncertainty, the more difficult it is to ascertain predictability. However, our analysis remains exact for TAS packets, even when considering jitter in their arrival times. Thus, the SAG method effectively maintains the predictability of TAS packets, even when overlapping gates cause increased variability in the response times.

The fact that we do not see any decrease in the ability to meet transmission deadlines in scenarios with gate overlap suggests a promising TAS configuration strategy where it is possible to ensure that high-priority packets are always sent on time without having to compromise as much on bandwidth usage. This insight could only be possibly obtained thanks to proposing a new analysis able to test such TAS configuration scenarios. Our new analysis is thus crucial for design-space exploration that may now explore non-conventional TAS gates configurations.

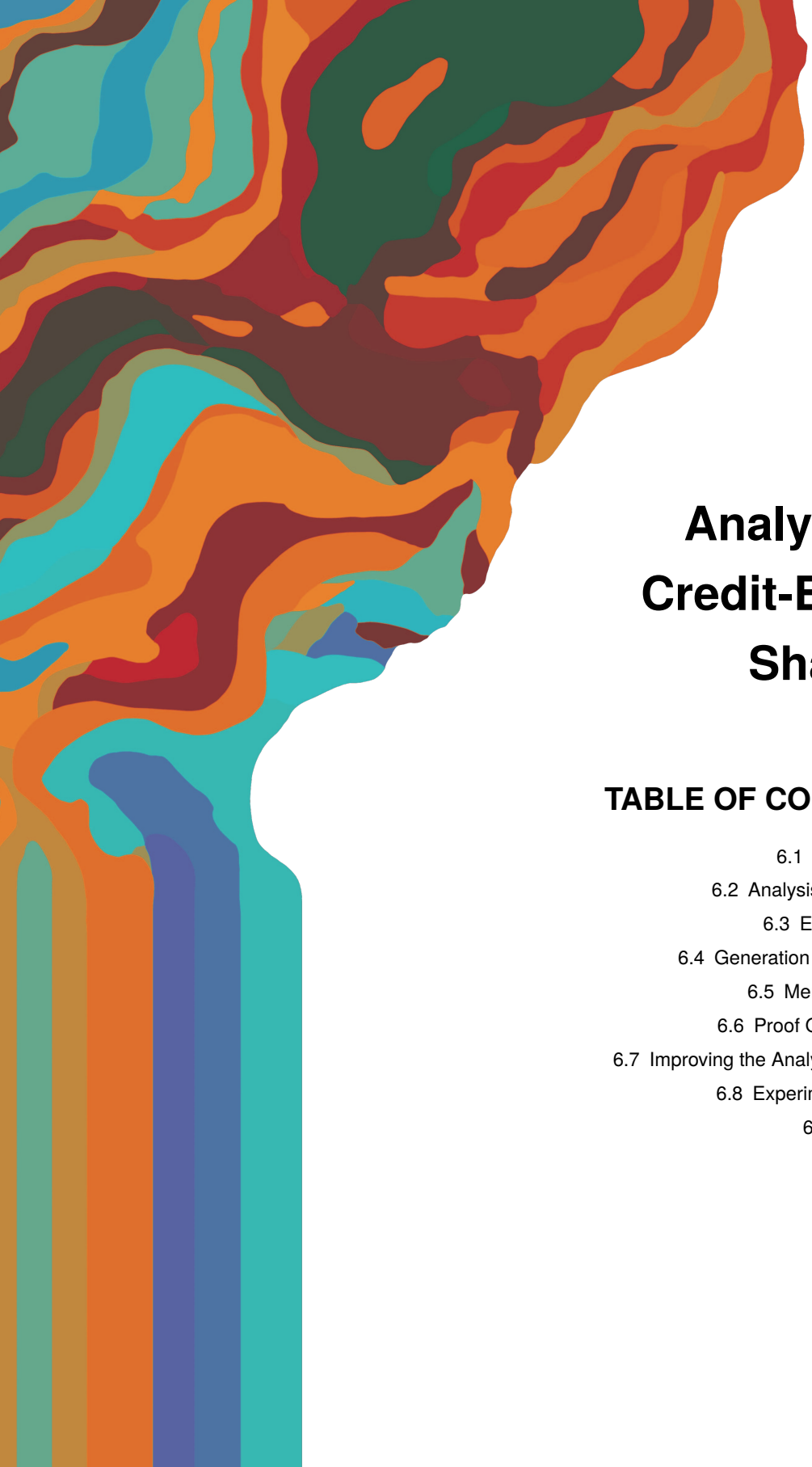
5.8 Conclusion

We presented the first best-case and worst-case latency analysis for packets shaped with Time-Aware Shapers with no restriction on their GCLs configuration. The analysis is a variation of the SAG approach. We extend the SAG in non-obvious ways to support the analysis of FIFO scheduling and TAS. We proved that the analysis is exact, meaning that it returns tight bounds on the best-case and worst-case latency of each analysed packet.

Our evaluation section highlights the utility of our new analysis, which allows to analyse out-of-the-box solutions for configuring TAS. In our evaluation section, we extracted just one of potentially many insights on how TAS could be configured to improve best-case and average performance without degrading the worst-case performance of communication flows in comparison to the conventional non-overlapping configuration of TAS. However, there are many different settings and parameters that could be explored with our new analysis to further our understanding of Time-Aware Shapers and potentially reveal more efficient ways to use them. Their exploration is out of the scope of this paper, but our analysis is a first step towards this broader goal, offering a way to explore any possible

TAS configuration.

Another extension to this work would be to apply the analysis across an entire network of switches. The best and worst-case latency that we obtain for each packet in a single switch can model the minimum and maximum arrival time of each packet into the next switch in the network, enabling us to analyse the end-to-end latency of packets in a network of switches.



6

Analysis of Credit-Based Shapers

TABLE OF CONTENTS

6.1 System Model

6.2 Analysis Methodology

6.3 Eligible Packets

6.4 Generation of New States

6.5 Merging of States

6.6 Proof Of Correctness

6.7 Improving the Analysis Scalability

6.8 Experimental Results

6.9 Conclusion

Chapter 4 introduced Time-Sensitive Networking (TSN) as a communication paradigm for real-time systems, highlighting how timing guarantees can be enforced through traffic shaping mechanisms. In Chapter 5, the Schedule Abstraction Graph (SAG) framework was adapted to analyse the timing behaviour of Time-Aware Shapers (TAS), where transmission opportunities are determined by a time-triggered gating schedule. While TAS provides strong determinism through predefined schedules, many practical systems instead rely on more flexible, rate-based mechanisms. This chapter therefore focuses on the analysis of the Credit-Based Shaper (CBS), a fundamentally different traffic shaping mechanism defined in the TSN standard.

TAS, analysed in the previous chapter, uses a time-triggered schedule in which transmission gates are opened and closed according to a predefined timetable, thereby granting critical traffic exclusive access to the link at specific times. In contrast, CBS regulates transmissions through a credit-based mechanism. Each traffic class accumulates credit while waiting and consumes credit while transmitting. This mechanism prevents unbounded transmission bursts and ensures that bandwidth is shared in a controlled manner, while still providing latency guarantees. Unlike TAS, however, CBS does not rely on a globally synchronised schedule, which introduces additional variability in transmission behaviour and complicates timing analysis.

A wide range of worst-case analysis techniques have been proposed for deriving safe upper bounds on the latency of traffic shaped with CBS. These include Network Calculus [52, 21, 66], Compositional Performance Analysis (CPA) [22, 23], eligibility-interval analyses [17], the trajectory approach [41], and conventional response-time analyses [8, 10]. Related efforts have analysed TAS using Network Calculus, CPA, and the Schedule Abstraction Graph (SAG) [62, 67, 60]. More recent work has extended eligibility-interval analyses to jointly consider TAS, CBS, and pre-emptive interference [13], while network-calculus-based approaches have also been developed to integrate multiple TSN mechanisms [66]. Beyond TSN, related work on switched Ethernet and Avionics Full-Duplex Switched Ethernet (AFDX) networks has investigated worst-case delay analysis using techniques such as Network Calculus, offset-based scheduling, and timed-automata modelling [1, 42, 43, 56, 11].

Despite these advances, existing approaches often face the trade-off between scalability and analytical precision discussed in Chapter 2. Methods based on analytical bounds scale well but may introduce pessimism, while approaches that explicitly explore system behaviour can become computationally expensive. The Schedule Abstraction Graph framework, introduced in Chapter 2 and extended to TSN in Chapter 5, provides a promising middle ground by exploring system behaviour selectively at relevant decision points.

In this chapter, we extend the SAG-based analysis framework to CBS-shaped traffic. While the overall analysis philosophy remains consistent with previous chapters, the underlying system behaviour differs significantly due to the dynamic evolution of credit. In particular, scheduling decisions are no longer governed by explicit transmission windows, but instead depend on the continuously evolving credit of each traffic class at each egress port. This introduces new challenges in modelling system state, identifying relevant decision points, and ensuring that the abstraction remains both precise and scalable.

To address these challenges, we modify the SAG framework to explicitly capture the evolution of credit for each traffic class and to incorporate this information into the state-space exploration. We develop a formal model of CBS behaviour, identify the key properties required for correct analysis, and introduce techniques to control the growth of the state space. These include adaptations of state representation, pruning strategies, and merging techniques tailored to the credit-based dynamics of CBS. Through these extensions, the framework is able to analyse interactions between multiple CBS classes while maintaining tractable complexity.

We assume that traffic arrival patterns are known a priori and analyse worst-case latency under these conditions. Our evaluation demonstrates that the proposed approach achieves higher schedulability ratios compared to existing methods, while also supporting more general CBS configurations, including scenarios with multiple interacting traffic classes.

The remainder of this chapter is structured as follows. Section 6.1 introduces the system model for CBS-based communication. Section 6.2 provides an overview of the analysis methodology. Section 6.3 defines how eligible packets are identified for graph construction, and Section 6.4 describes the generation of new states. Section 6.5 presents the state-merging strategy used to control complexity. Section 6.6 establishes correctness, Section 6.7 discusses implementation aspects for scalability, Section 6.8 presents the experimental evaluation, and Section 6.9 concludes the chapter.

6.1 System Model

Building on the TSN abstraction introduced in Chapter 4 and the SAG-based modelling approach developed in Chapter 5 for TAS, this section defines the system model used for analysing Credit-Based Shapers (CBS). While the overall network structure remains consistent with the previously introduced TSN model, the key difference lies in how transmission opportunities are determined. In contrast to the time-triggered behaviour of TAS, CBS introduces a dynamic, credit-driven mechanism that must be explicitly captured in the system state.

We consider Time-Sensitive Networking (TSN) switches that support multiple full-duplex communication streams, each connecting an ingress port to an egress port. Each egress port maintains multiple queues, one for each traffic class. Communication flows are assigned to a traffic class and, upon arrival, packets are placed into the queue corresponding to both their destination egress port and assigned class. As in Chapter 5, the analysis focuses on a single egress port, since ports operate independently under the considered assumptions.

Traffic Classes Let $\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$ denote the set of traffic classes available at each egress port. Each class has a unique priority, consistent with the priority-based scheduling assumptions introduced earlier. Each egress port maintains one queue per traffic class, and transmissions from these queues are governed by CBS.

Unlike TAS, where transmission eligibility is determined by gate states, CBS determines eligibility through a credit mechanism. Each class is associated with a credit value that evolves over time depending on whether the class is transmitting or waiting. The credit increases when the class is not transmitting and decreases while transmitting. A class is eligible to transmit only when its credit is non-negative.

Formally, the queue of traffic class C_a (for $1 \leq a \leq |\mathcal{C}|$) at egress port p is defined as

$$Q_{p,a} = \left(\pi_{C_a},\ \alpha^+_{p,a}\right),$$

where $\pi_{C_a} \in \mathbb{N}_{\geq 0}$ denotes the priority of class C_a (lower values indicate higher priority), and $\alpha^+_{p,a} > 0$ is the idle slope, i.e., the rate at which credit is replenished when the class is not transmitting.

When a queue has negative credit or is waiting to transmit a packet, its credit increases at rate $\alpha^+_{p,a}$. If no packet is waiting in the queue once the credit becomes non-negative, the accumulated credit is reset to 0. The send slope, which determines the rate at which credit is consumed during transmission, is given by

$$\alpha^-_{p,a} = \alpha^+_{p,a} - 1.$$

Thus, when a packet of class C_a is transmitted, the credit decreases at rate $\alpha^-_{p,a}$.

To ensure that all classes can be serviced without starvation, the following condition must hold:

$$\sum_{a=1}^{|\mathcal{C}|} \alpha^+_{p,a} \leq 1.$$

As in Chapter 5, to simplify notation, we omit the index p when referring to parameters such as idle slope and send slope, since all properties are defined

with respect to a single egress port.

Flows and Packets Let $\mathcal{F} = \{F_1, \ldots, F_{|\mathcal{F}|}\}$ denote the set of communication flows transmitted through the switch. Each flow $F_i \in \mathcal{F}$ is assigned to a traffic class C_a and generates a sequence of packets $p_{i,j}$, where $j \in \mathbb{N}$ denotes the j-th packet of the flow. As in previous chapters, packet releases follow a recurring pattern and repeat over a hyperperiod.

The set of all packets is denoted by $\mathcal{P}$. As in Chapter 5, each packet $p_{i,j}$ is characterised by

$$p_{i,j} = \left(a^{\min}_{p_{i,j}}, a^{\max}_{p_{i,j}}, d_{p_{i,j}}, \ell^{\min}_{p_{i,j}}, \ell^{\max}_{p_{i,j}}\right),$$

where $a^{\min}_{p_{i,j}}, a^{\max}_{p_{i,j}} \in \mathbb{R}$ (with $a^{\max}_{p_{i,j}} \geq a^{\min}_{p_{i,j}}$) denote the earliest and latest possible arrival times, $d_{p_{i,j}} \in \mathbb{R}$ is the absolute deadline, and $\ell^{\min}_{p_{i,j}}, \ell^{\max}_{p_{i,j}} \in \mathbb{R}_{>0}$ (with $\ell^{\min}_{p_{i,j}} \leq \ell^{\max}_{p_{i,j}}$) are lower and upper bounds on packet length (in bytes).In this work, the inter-packet gap is not modelled as a separately but is instead assumed to be part of each packet's transmission duration. Let C_{IPG} denote the transmission time corresponding to the inter-packet gap. Then, the minimum and maximum transmission durations are derived as

$$\hat{C}^{\min}_{p_{i,j}} = \frac{\ell^{\min}_{p_{i,j}}}{R^{\mathrm{tx}}} + C_{\mathrm{IPG}} \quad \text{and} \quad \hat{C}^{\max}_{p_{i,j}} = \frac{\ell^{\max}_{p_{i,j}}}{R^{\mathrm{tx}}} + C_{\mathrm{IPG}}, \tag{6.1}$$

where R^{tx} is the transmission rate of the egress port.

Transmission Scenarios Different combinations of arrival times and packet lengths within their respective bounds can therefore result in significantly different transmission orders and credit trajectories. To capture all possible behaviours, we define the notion of transmission scenarios, analogous to execution scenarios introduced in Chapter 2. Using the same notion as in Chapter 5, a transmission scenario γ specifies a concrete arrival time $a_{p_{i,j}}$ and transmission duration $C_{p_{i,j}}$ for each packet $p_{i,j} \in \mathcal{P}$:

Given a transmission scenario γ, the CBS behaviour is deterministic, and the finish time $f_{p_{i,j}}(\gamma)$ of each packet is uniquely defined. Let Γ denote the set of all possible transmission scenarios.

A packet set is considered schedulable if no packet misses its deadline in any scenario:

$$\forall \gamma \in \Gamma,\ f_{p_{i,j}}(\gamma) \leq d_{p_{i,j}}.$$

Conversely, the system is unschedulable if there exists at least one scenario in which a packet misses its deadline:

$$\exists \gamma \in \Gamma,\ f_{p_{i,j}}(\gamma) > d_{p_{i,j}}.$$

The latency of a packet $p_{i,j}$ in a scenario γ is defined as $f_{p_{i,j}}(\gamma) - a^{\min}_{p_{i,j}}$. The best-case and worst-case latency are given by

$$\mathrm{BCRT}_{p_{i,j}} = \min_{\gamma \in \Gamma}\{f_{p_{i,j}}(\gamma) - a^{\min}_{p_{i,j}}\}, \quad \mathrm{WCRT}_{p_{i,j}} = \max_{\gamma \in \Gamma}\{f_{p_{i,j}}(\gamma) - a^{\min}_{p_{i,j}}\}.$$

This formulation aligns with the general analysis problem introduced in Chapter 2, where the objective is to reason about all possible system behaviours without explicitly enumerating each scenario. In the following sections, we show how the SAG framework can be adapted to explore this space efficiently for CBS-based systems.

6.2 Analysis Methodology

To determine whether a packet set is schedulable, we explore all possible combinations of:

- packet arrivals,
- transmission durations, and
- CBS-driven scheduling decisions

throughout the system execution.

Every transmission updates the system state, and by exhaustively exploring all reachable states, we obtain safe best- and worst-case latency bounds. To perform this exploration efficiently, we use the Schedule Abstraction Graph (SAG) methodology [48, 46, 60], which models how scheduling decisions drive state space exploration. In this chapter, we extend SAG to analyse traffic shaping by CBS.

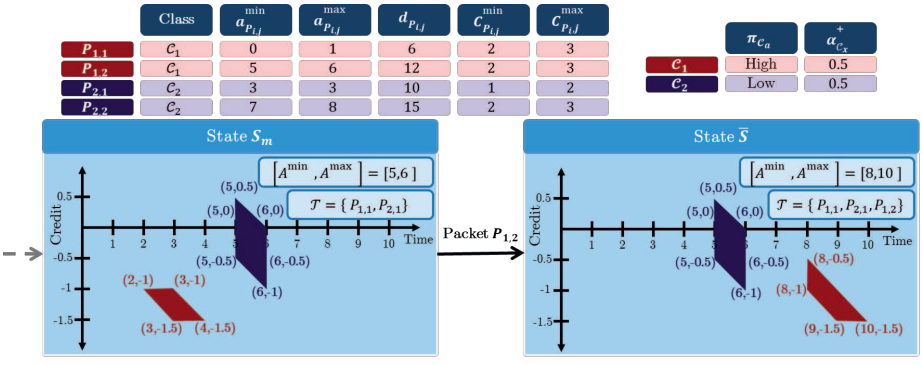


Figure 6.1: State Transition Diagram for Example 6.2

The SAG is a directed acyclic graph in which:

- vertices represent abstract system states, and
- edges represent transitions caused by packet transmissions.

Construction starts from an initial state with all queues empty. Whenever the scheduler must choose a packet to transmit, all feasible choices are explored, generating successor states. This process continues until no new states appear, yielding a graph that captures all valid transmission scenarios and enables derivation of safe latency bounds.

Each vertex v_m corresponds to an abstract state s, and an edge from s to s' labelled with $p_{i,j}$ indicates that transmitting $p_{i,j}$ in s leads to s'.

Each abstract state s stores:

1. The set $\mathcal{T}$ of packets already transmitted in the scenarios represented by s.
2. A time interval $[A^{\min}, A^{\max}]$, where:
   - $A^{\min}$ is a lower bound on the earliest possible time that the egress port may become available to transmit a new packet, and
   - $A^{\max}$ is an upper bound on the earliest time at which the egress port availability is guaranteed.
3. For each class C_a, a set ϵ_{C_a} of all possible finish-time/credit pairs (t, c) for the most recently transmitted packet of that class. If no such packet has been transmitted yet, then:

$$\epsilon_{C_a} = \{(0, 0)\}.$$

An abstract state s represents all concrete system states in which:

- the same set of packets $\mathcal{T}$ has been transmitted,
- the egress port availability time lies within $[A^{\min}, A^{\max}]$, and
- the credit values of all traffic classes lie within the corresponding sets ϵ_{C_a}.

EXAMPLE 6.1

To show how graph states are built, consider Figure 6.1. The switch supports two traffic classes, C_1 and C_2, where C_1 has higher priority. The packet set has four packets, each in one of the two classes, with parameters as defined in the system model.

For the state s to be reachable, packets $p_{1,1}$ and $p_{2,1}$ must have already been transmitted, giving:

$$\mathcal{T} = \{p_{1,1}, p_{2,1}\}.$$

After transmitting these packets:

- the state may possibly become available at time 5, and
- it is certainly available at time 6,

which is reflected by the interval:

$$[A^{\min}, A^{\max}] = [5, 6].$$

For each traffic class, the set $\epsilon_{\mathcal{C}_a}$ is visualised as a polygon in the credit-time plane. Consider C_a, which has already transmitted $p_{1,1}$. The red polygon represents all possible credit-time pairs at the finishing times of $p_{1,1}$.
Since $p_{1,1}$ may:

- arrive between time 0 and 1,
- begin transmission at any point in that interval, and
- transmit for 2 to 3 time units,

the credit decreases during transmission according to the send slope, producing a range of possible credit values at completion. The set of all such completion points forms the depicted polygon.
Extending the graph from s to create s' updates each polygon according to how the transmission of $p_{1,2}$ affects the credit. The following sections and examples illustrate in detail how this transition is computed.

Although the polygons shown in the example appear bounded, the theoretical credit-time regions they represent are not. Each region contains infinitely many possible credit-time pairs. While this could raise scalability concerns, Section 6.7 explains how we interpret and store these regions so that the overall analysis remains efficient and scalable.

The SAG is built using Algorithm 4. It takes as input:

- the set of packets $\mathcal{P}$, and
- for each class $C_a \in \mathcal{C}$, its priority π_{C_a} and idle slope α_a^+.

Initialisation Phase (lines 1–9 of Algorithm 4).
For each packet $p_{i,j} \in \mathcal{P}$:

- the Best-Case Latency (Best-Case Response Time (BCRT)) is initialised to ∞, and
- the Worst-Case Latency (WCRT) is initialised to 0.

Algorithm 4 Schedule Abstraction Graph Construction

Require: Packet set $\mathcal{P}$, and $\forall C_a \in \mathcal{C}, \pi_{C_a}$ and α_a^+
1: **for** each packet $p_{i,j} \in \mathcal{P}$ **do**
2: $\quad \mathrm{BCRT}_{p_{i,j}} \leftarrow \infty$ ▷ Initialize the BCRT for $p_{i,j}$
3: $\quad \mathrm{WCRT}_{p_{i,j}} \leftarrow 0$ ▷ Initialize the WCRT for $p_{i,j}$
4: **end for**
5: $s \leftarrow \{\mathcal{T} = \emptyset,\ [A^{\min}, A^{\max}] = [0,0],\ \epsilon_{C_a} = (0,0)\ \forall\, C_a \in \mathcal{C}\}$ ▷ Initialise the state
6: $L^{\mathrm{cur}} \leftarrow \{s\}$ ▷ Initialise current layer of the SAG
7: $L^{\mathrm{next}} \leftarrow \emptyset$ ▷ Initialise next layer of the SAG
8: **while** $L^{\mathrm{cur}} \neq \emptyset$ **do**
9: $\quad$ **for** each state $s \in L^{\mathrm{cur}}$ **do**
10: $\quad\quad$ Find the set of eligible packets ▷ Refer to Section 6.3
11: $\quad\quad$ **for** each eligible packet $p_{i,j}$ **do**
12: $\quad\quad\quad$ Compute $f_{p_{i,j}}^{\min}$ and $f_{p_{i,j}}^{\max}$ using Equation (6.14)
13: $\quad\quad\quad$ $\mathrm{BCRT}_{p_{i,j}} \leftarrow \min\{f_{p_{i,j}}^{\min} - a_{p_{i,j}}^{\min},\ \mathrm{BCRT}_{p_{i,j}}\}$
14: $\quad\quad\quad$ $\mathrm{WCRT}_{p_{i,j}} \leftarrow \max\{f_{p_{i,j}}^{\max} - a_{p_{i,j}}^{\min},\ \mathrm{WCRT}_{p_{i,j}}\}$
15: $\quad\quad\quad$ ▷ Update BCRT and WCRT
16: $\quad\quad\quad$ **if** $f_{p_{i,j}}^{\max} > d_{p_{i,j}}$ **then**
17: $\quad\quad\quad\quad$ **return** Unschedulable
18: $\quad\quad\quad$ **end if**
19: $\quad\quad\quad$ Create a new state s' ▷ Refer to Section 6.4
20: $\quad\quad\quad$ **if** $\exists s \in L^{\mathrm{next}}$ that can be merged with s' **then**
21: $\quad\quad\quad\quad$ Merge s and s' ▷ Refer to Section 6.5
22: $\quad\quad\quad$ **else**
23: $\quad\quad\quad\quad$ $L^{\mathrm{next}} \leftarrow L^{\mathrm{next}} \cup \{s'\}$
24: $\quad\quad\quad$ **end if**
25: $\quad\quad$ **end for**
26: $\quad$ **end for**
27: $\quad L^{\mathrm{cur}} \leftarrow L^{\mathrm{next}}$
28: $\quad L^{\mathrm{next}} \leftarrow \emptyset$
29: **end while**
30: **return** Schedulable

The SAG begins with the initial state s, where:

- no packet has been transmitted, i.e., $\mathcal{T} = \emptyset$,
- the egress port is both possibly and certainly available at time 0, i.e., $[A^{\min}, A^{\max}] = [0,0]$, and
- each traffic class starts with credit 0, i.e., $\epsilon_{C_a} = \{(0,0)\}$.

The SAG is constructed in a breadth-first, layer-by-layer manner. To avoid storing the entire graph, the algorithm maintains:

- L^{cur}: the set of states in the current layer, and
- L^{next}: the set of states generated for the next layer.

The construction alternates between two phases:

Expansion Phase (lines 10–21 of Algorithm 4).

At each step:

- all states in L^{cur} are explored,
- for each state, all possible next scheduling decisions are identified (Section 6.3),
- for every valid decision, a new state is generated (Section 6.4),
- for all scheduled packets, the BCRT and WCRT are updated.

The analysis continues as long as no deadline miss is found.

Merge Phase (lines 22–26 of Algorithm 4).

To control state-space growth:

- each newly generated state is checked against existing states in L^{next}, and
- if a merge is possible, the states are combined (Section 6.5),
- otherwise, the new state is added to L^{next}.

6.3 Eligible Packets

The main step in the expansion phase is to determine which packets may be eligible for transmission next. To do this, we consider all switch components that influence scheduling decisions. When a packet arrives at a switch, it is placed in the FIFO queue associated with its traffic class, so the first requirement for eligibility is that the packet must be at the head of this queue. In addition, the traffic class must have non-negative credit for transmission to begin. However, these two conditions alone do not guarantee that the packet will be scheduled next. We must also ensure that no higher-priority packet becomes eligible earlier, and that no other packet is already able to start transmission sooner due to being ready and having access to the egress port. In this section, we describe how all these factors are combined to determine the full set of packets that may be eligible for transmission.

6.3.1 Accounting for FIFO Queues

As in Chapter 5, packets of each traffic class are stored in a dedicated FIFO queue, and a packet must be at the head of its queue to be eligible for transmission. The quantities $tr_{C_a}^{\max}$ and top_{C_a}, defined in Chapter 5 in Equations (5.2) and (5.3), are reused here unchanged to account for which packets may be at the head of the FIFO queue of class C_a.

In addition, for the CBS analysis we introduce the quantity $tr_{C_a}^{\min}$, which lower-bounds the earliest time at which a not-yet-transmitted packet of class C_a may have entered its FIFO queue.

Lemma 6.1. *If the system is in abstract state s, then for any concrete state represented by s, the earliest time when a not-yet-transmitted packet of class C_a may possibly have entered its FIFO queue is lower bounded by*

$$tr_{C_a}^{\min} = \min\{a_{p_{i,j}}^{\min} \mid p_{i,j} \notin \mathcal{T}, p_{i,j} \text{ is of } C_a\}. \tag{6.2}$$

Proof. Only packets of class C_a that have not yet been transmitted can still be in C_a's FIFO queue in any concrete state represented by s, that is, only packets in $C_a \setminus \mathcal{T}$. Since $a_{p_{i,j}}^{\min}$ denotes the earliest possible arrival time of a packet $p_{i,j}$ at the switch, the earliest time at which such a packet may possibly arrive and enter the FIFO queue is $\min\{a_{p_{i,j}}^{\min} \mid p_{i,j} \notin \mathcal{T}, p_{i,j}$ is of $C_a\}$. □

Once we have identified the packets that may be at the head of their respective queues, the next step is to determine whether each of these packets may begin transmission before any other packet. To do this, we calculate a lower and upper bound on when each packet $p_{i,j} \in top_{C_a}$ may possibly start its transmission assuming it is the first packet transmitted in abstract state s. The bound $tr_{C_a}^{\min}$ is the additional quantity introduced here for the CBS analysis and will be used later in the derivation of transmission-start bounds. We then prove that $p_{i,j}$ may begin transmission before any other packet in abstract state s only if the lower bound is smaller than or equal to the upper bound.

6.3.2 Transmission Start Time Bounds

The start time bounds of a packet $p_{i,j}$ in top_{C_a} are influenced by several factors, including when the egress port becomes available for transmission, when the credit of $p_{i,j}$'s traffic class reaches a non-negative value, and whether there are packets of higher-priority classes eligible for transmission.

This section is divided into four parts:

1. We first introduce a function that calculates when the credit of a class becomes non-negative.

2. We then derive a lower bound on the transmission start time of any packet of a given class C_a in abstract state s.

3. Then, we similarly derive an upper bound on the transmission start time of any packet of a given class C_a assuming C_a is the first to transmit in abstract state s.

4. These results are combined to establish an eligibility criterion that identifies the set of packets that could be the first transmitted in abstract state s and

derive upper and lower bounds on the completion of their transmission if they are the first to be transmitted in s.

Non-negative Credit Attainment Function

As defined in Section 6.1, the set ϵ_{C_a} represents all possible credit value of class C_a for all possible time instant at which the last transmitted packet of class C_a finished its transmission. Given that information, we can determine the earliest and latest time at which C_a's credit will become non-negative after its last transmission. To capture this behaviour, we define a function zc_{C_a} that takes as input a pair (t, c), where t is a time instant and c is the credit at that time, and returns (t') such that t' is the earliest time at or after t at which the credit of C_a is non-negative assuming that C_a's credit was c at t.

$$zc_{C_a}(t, c) = \begin{cases} t & \text{if } c \geq 0, \\ t - \dfrac{c}{\alpha_a^+} & \text{otherwise.} \end{cases} \tag{6.3}$$

where α_a^+ is the idle slope of class C_a, which defines the rate with which the credit rises when C_a's credit is negative.

We can now use the function zc_{C_a} to project all pairs (t, c) in ϵ_{C_a} to all time instant at which C_a's credit becomes non-negative in any transmission scenario that led to the abstract system abstract state s. We call this set of time instants ι_{C_a}. Formally, we define

$$\iota_{C_a} = \{t' \mid \exists (t, c) \in \epsilon_{C_a},\ t' = zc_{C_a}(t, c)\} \tag{6.4}$$

We can then define the earliest time $t_{C_a}^{+,\min}$ and latest time $t_{C_a}^{+,\max}$ at which C_a's credit becomes non-negative in abstract state s as follows.

$$t_{C_a}^{+,\min} = \min\{t \mid t \in \iota_{C_a}\} \tag{6.5}$$

$$t_{C_a}^{+,\max} = \max\{t \mid t \in \iota_{C_a}\} \tag{6.6}$$

EXAMPLE 6.2

To illustrate how the coordinates in ϵ_{C_a} are transformed into ι_{C_a}, we refer to the same example shown in Figure 6.1. Specifically, we focus on the class C_1 and examine how its corresponding set ϵ_1 is assigned to ι_1. An isolated view of ϵ_1, along with a visualisation of the transformation, is provided in Figure 6.2. In this example, ϵ_1 contains all coordinates in the red space. For each of these coordinates, zc_{C_a} projects all points of ϵ_1 located below the x-axis onto

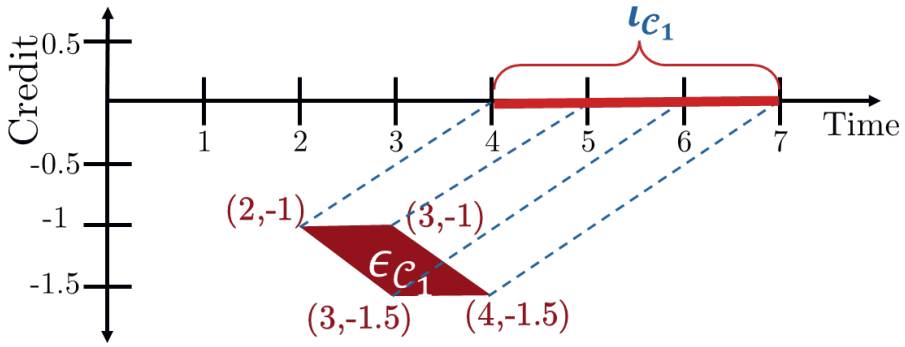


Figure 6.2: Example showing the calculation of ι_{C_a}

the x-axis with a slope equal to C_1's idle slope. This results in the set ι_1 containing all time instant within the interval $[4, 7]$. We then have that C_a's credit becomes non-negative at the earliest at $t_{C_a}^{+,\min} = 4$ and at the latest at $t_{C_a}^{+,\max} = 7$.

Lower Bound on Start of Transmission of Class C_a

The earliest time at which class C_a can begin the transmission of any packet in abstract state s is determined by three factors: the earliest possible time $tr_{C_a}^{\min}$ at which a packet of that class may enter its FIFO queue, the earliest time $A^{\min}$ at which the egress port becomes available in s, and the earliest time $t_{C_a}^{+,\min}$ at which C_a's credit becomes non-negative.

Lemma 6.2. *For any traffic class C_a, and for all concrete state represented by the abstract state s, the start time of transmission of any packet of that class is lower bounded by*

$$t_{p_{i,j}}^{\text{lb}} = \max\{tr_{C_a}^{\min},\, A^{\min},\, t_{C_a}^{+,\min}\}. \tag{6.7}$$

Equivalently, no packet of class C_a in any concrete state in the abstract state s can start transmission before time $t_{p_{i,j}}^{\text{lb}}$.

Proof. For a packet of class C_a to start its transmission at time t, three necessary conditions must be satisfied:

1. Packet Arrival: A packet must be in C_a's FIFO queue. By Lemma 6.1, $tr_{C_a}^{\min}$ is the earliest such time. Therefore, the start time t must satisfy $t \geq tr_{C_a}^{\min}$.

2. Egress Port Availability: The egress port must be available for transmission. Since $A^{\min}$ is the earliest time at which the port is available in s, we have $t \geq A^{\min}$.

3. Non-negative Credit: Credit for class C_a must be non-negative. By definition, $t_{C_a}^{+,\min}$ is the earliest time the credit becomes non-negative. Thus, $t \geq t_{C_a}^{+,\min}$.

Therefore, $t_{p_{i,j}}^{\text{lb}} = \max\{tr_{C_a}^{\min}, A^{\min}, t_{C_a}^{+,\min}\}$. Any earlier time would violate at least one of the three conditions above. □

Upper Bound on Start of Transmission of Class C_a

Assuming C_a is the first class transmitting a packet in any concrete state in abstract state s, the latest time this transmission may start is upper bounded by two critical factors. First, if any other class becomes eligible for transmission earlier than C_a, then C_a cannot be the next to transmit since the switch always transmits the earliest eligible packet. Second, if any class with a higher priority than C_a becomes eligible before C_a start transmitting, then C_a loses its opportunity to be the first transmitting.

To formalise these properties, we introduce two key lemmas. In Lemma 6.4, we define t^{eet}, which captures the earliest time at which some class becomes certainly eligible to transmit, potentially preventing C_a from being selected next. In Lemma 6.5, we define $t_{C_a}^{\text{high}}$ to capture the effect of higher priority classes becoming eligible, which further limits when C_a can transmit.

As established earlier, a traffic class becomes eligible to transmit a packet only when three conditions are met: there is a packet in its queue, the egress port is available, and the class credit is non-negative. To determine the time at which a class is certainly eligible, we consider the earliest time that each of these conditions is guaranteed to hold.

Lemma 6.3. *For any class C_a and any concrete state represented by the abstract state s, the time at which C_a becomes eligible to transmit is upper bounded by* $\max\{tr_{C_a}^{\max}, A^{\max}, t_{C_a}^{+,\max}\}$

Proof. The three following conditions must be satisfied:

1. Packet Arrival: A packet must be in C_a's FIFO queue. By Lemma 5.3, there is certainly a packet in the FIFO queue at $tr_{C_a}^{\max}$.

2. Egress Port Availability: The egress port must be available for transmission. By definition, the egress port is certainly available at and after $A^{\max}$ in s.

3. Non-negative Credit: Credit for class C_a must be non-negative. By definition, $t_{C_a}^{+,\max}$ is the latest time C_a's credit becomes non-negative in s.

It results that all three conditions are certainly respected at and after $\max\{tr_{C_a}^{\max}, A^{\max}, t_{C_a}^{+,\max}\}$ □

Lemma 6.4. *For every concrete state represented by the abstract state s, there exists a class that is certainly eligible for transmission at or before*

$$t^{eet} = \min_{\forall C_a \in \mathcal{C}} \left\{ \max\{tr_{\mathcal{C}_a}^{\max}, A^{\max}, t_{C_a}^{+,\max}\} \right\} \tag{6.8}$$

and thus t^{eet} is an upper bound on the time by which some class is certainly eligible.

Proof. By Lemma 6.3, at least one class $C_a \in \mathcal{C}$ is certainly eligible at

$$\max\{tr_{\mathcal{C}_a}^{\max}, A^{\max}, t_{C_a}^{+,\max}\}$$

in s. Thus, a class is certainly eligible at $\min_{\forall C_a \in \mathcal{C}} \left\{ \max\{tr_{\mathcal{C}_a}^{\max}, A^{\max}, t_{C_a}^{+,\max}\} \right\}$. □

We now capture the effect of higher priority classes on when C_a may start transmitting for every concrete state in abstract state s.

Lemma 6.5. *For every concrete state represented by the abstract state s, the time at which a class with higher-priority than C_a becomes eligible is upper bounded by*

$$t_{C_a}^{high} = \min_{\forall C_b \in \mathcal{C} | \pi_{C_b} < \pi_{C_a}} \left\{ \max\{tr_{\mathcal{C}_a}^{\max} b, t_{C_b}^{+,max}\} \right\} \tag{6.9}$$

Proof. For any class C_b with higher priority than C_a (i.e., $\pi_{C_b} < \pi_{C_a}$), consider the two conditions that do not depend on egress-port availability: a packet of C_b is guaranteed to be at the head of its queue no later than $tr_{\mathcal{C}_a}^{\max} b$, and its credit is guaranteed to become non-negative no later than $t_{C_b}^{+,\max}$. Once both of these conditions are satisfied, class C_b can transmit as soon as the egress port becomes available. Thus, from time $t_{C_a}^{\text{high}}$ onward, whenever the egress port becomes available, at least one higher-priority class is already ready to transmit. □

We now use the bounds proven in Lemmas 6.2, 6.4 and 6.5 to decide whether a traffic class C_a may possibly be the first to transmit a packet in any concrete state present in the abstract state s.

Theorem 6.6. *There may exists a concrete state represented by the abstract state s where class C_a is the first class to transmit a packet, only if*

$$t_{p_{i,j}}^{\text{lb}} \leq t^{eet} \wedge t_{p_{i,j}}^{\text{lb}} < t_{C_a}^{high} \tag{6.10}$$

Proof. By Lemma 6.4, t^{eet} is the earliest time at which some class is certainly eligible to transmit. Therefore, if C_a is the first class to transmit, it must transmit

at or before t^{eet}, implying that it must become eligible at or before t^{eet}. Since, by Lemma 6.2, $t^{\text{lb}}_{p_{i,j}}$ is a lower bound on when C_a becomes eligible in state s, we must have $t^{\text{lb}}_{p_{i,j}} \leq t^{\text{eet}}$.

Similarly, by Lemma 6.5, C_a cannot be the highest-priority eligible class from $t^{\text{high}}_{C_a}$ onward in any concrete state in abstract state s. Thus, if C_a has not started transmission strictly before $t^{\text{high}}_{C_a}$, it will no longer be chosen by the switch. Hence, to be the first class to start a transmission in s, C_a must be eligible strictly before $t^{\text{high}}_{C_a}$, thereby requiring $t^{\text{lb}}_{p_{i,j}} < t^{\text{high}}_{C_a}$. □

Eligible Packets

We can now use the properties proven above to decide whether there can be a concrete state represented by the abstract state s in which a packet $p_{i,j}$ is the first packet transmitted.

Theorem 6.7. *There may be a concrete state represented by the abstract state s where a packet $p_{i,j}$ of class C_a is the first packet transmitted only if $p_{i,j} \in top_{C_a} \wedge \max\{a^{\min}_{p_{i,j}}, t^{lb}_{C_a}\} \leq t^{eet} \wedge \max\{a^{\min}_{p_{i,j}}, t^{lb}_{C_a}\} < t^{high}_{C_a}$.*

Proof. For $p_{i,j}$ to be the first packet transmitted in some concrete state present in abstract state s, it must respect the following properties:

1. Its class must be able to be the first class that transmit a packet. By Theorem 6.6, we must then have $t^{\text{lb}}_{p_{i,j}} \leq t^{\text{eet}} \wedge t^{\text{lb}}_{p_{i,j}} < t^{\text{high}}_{C_a}$.

2. $p_{i,j}$ must be at the head of its FIFO queue in some concrete state present in abstract state s, that is, we must have $p_{i,j} \in top_{C_a}$.

3. It must have entered in the FIFO queue before any other class than C_a starts transmitting. Since $a^{\min}_{p_{i,j}}$ is a lower bound on when $p_{i,j}$ arrived at the switch and thus enters its FIFO queue, and because by Theorem 6.6, t^{eet} and $t^{\text{high}}_{C_a}$ are upper bounds on when C_a must start transmitting to be the first to transmit, it requires that $a^{\min}_{p_{i,j}} \leq t^{\text{eet}} \wedge a^{\min}_{p_{i,j}} < t^{\text{high}}_{C_a}$.

Applying all four properties yields the claim. □

A packet that passes the requirements of Theorem 6.7 is called an eligible packet in abstract state s. For every eligible packet $p_{i,j}$ of class C_a, we can compute a lower bound $\sigma^{\min}_{p_{i,j}}$ and upper bound $\sigma^{\max}_{p_{i,j}}$ on its start time in s as follows.

$$\sigma^{\min}_{p_{i,j}} = \max\{a^{\min}_{p_{i,j}}, t^{\text{lb}}_{p_{i,j}}\} \tag{6.11}$$

$$\sigma^{\max}_{p_{i,j}} = \min\{t^{\text{eet}}, t^{\text{high}}_{C_a}\} \tag{6.12}$$

The upper bound $\sigma^{\max}_{p_{i,j}}$ is a direct consequence of Theorem 6.6, and the lower bound $\sigma^{\min}_{p_{i,j}}$ comes from the fact that, to start its transmission, a packet must have entered its FIFO queue, which happens at the earliest at $a^{\min}_{p_{i,j}}$, and its class must be eligible, which happens at the earliest at $t^{\text{lb}}_{p_{i,j}}$ according to Lemma 6.2.

Since packet transmissions are non-pre-emptive, the earliest and latest finish time bounds on $p_{i,j}$'s transmission can be computed by adding the minimum and maximum transmission duration to the start time bounds.

$$f^{\min}_{p_{i,j}} = \sigma^{\min}_{p_{i,j}} + \hat{C}^{\min}_{p_{i,j}} \tag{6.13}$$

$$f^{\max}_{p_{i,j}} = \sigma^{\max}_{p_{i,j}} + \hat{C}^{\max}_{p_{i,j}} \tag{6.14}$$

After a packet has been transmitted, the abstract system state must be updated to reflect this change. The procedure for constructing the updated state is in the following section.

6.4 Generation of New States

Once all eligible packets are identified, we determine how transmitting a packet $p_{i,j} \in C_a$ transforms an abstract state s into a new abstract state s'.

The new abstract state s' is built as follows: (1) The set of transmitted packets is updated to include packet $p_{i,j}$, i.e., $\mathcal{T}(s') = \mathcal{T}(s) \cup \{p_{i,j}\}$. (2) The availability of the egress port is updated to reflect the finish time of $p_{i,j}$. Since the port becomes available immediately after the transmission completes, we have $A^{\min}(s') = f^{\min}_{p_{i,j}}$ and $A^{\max}(s') = f^{\max}_{p_{i,j}}$. (3) The sets $\epsilon_{C_o}, \forall C_o \in \mathcal{C} \setminus C_a$ remain unchanged since the last packet transmitted by every class $C_o \in \mathcal{C} \setminus C_a$ is still the same as in s. (4) The set ϵ_{C_a} must be updated in ways that are discussed in the rest of this section.

For each time–credit pair $(t, c) \in \epsilon_{C_a}$, we must determine the corresponding time–credit pairs at the possible finish times of $p_{i,j}$. To do so in two steps, we first compute the credit at the possible transmission start times $\sigma \in [\sigma^{\min}_{p_{i,j}}, \sigma^{\max}_{p_{i,j}}]$ of the packet, and then calculate the credit when the packet completes its transmission.

The credit at a given start time depends on when the packet arrived, since credit evolves differently depending on whether the queue is empty or non-empty. The TSN standard states that the credit evolves according to the following CBS rules:

- **Rule 1 (Reset when queue empty).** If the credit is non-negative when the queue becomes empty, it is reset to zero. The credit start increasing again at rate α^+_a from the next packet arrival.

- **Rule 2 (Accumulation).** If the credit is negative or the credit is positive but

the queue is non-empty, the credit increases continuously at rate α_a^+ until the packet at the top of the queue starts to be transmitted.

- **Rule 3 (Decrease during transmission).** While a packet is transmitted, credit decreases at rate of the send slope.

Rules 1 and 2 define how to calculate the value of the credit at the start of a packet transmission, and Rule 3 defines how the credit evolves from the start to the finish time of the transmission. In the following, we describe the first step of this two-step procedure.

Lemma 6.8. *The credit accumulation function*

$$g_{C_a}(t, c, a_{p_{i,j}}, s) = \begin{cases} \left(\sigma,\ \alpha_a^+(\sigma - a_{p_{i,j}})\right) & \text{if } \begin{array}{l}\left(c \geq 0 \wedge t < a_{p_{i,j}}\right) \vee \\ \left(c < 0 \wedge t - \dfrac{c}{\alpha_a^+} < a_{p_{i,j}}\right)\end{array} \\ \left(\sigma,\ \alpha_a^+(\sigma - t) + c\right) & \text{otherwise} \end{cases} \tag{6.15}$$

*gives the credit of class C_a at time σ, starting from credit c at time t, when a packet arrives at time $a_{p_{i,j}}$. The two cases correspond directly to the behaviours defined by **Rule 1** and **Rule 2**.*

Proof. Consider a coordinate (t, c) in ϵ_{C_a}, the arrival interval of the packet $a_{p_{i,j}} \in [a_{p_{i,j}}^{\min}, tr_{C_a}^{\max}]$ and $\sigma \in \left[\sigma_{p_{i,j}}^{\min}, \sigma_{p_{i,j}}^{\max}\right]$

Case 1: Credit becomes non-negative before $a_{p_{i,j}}$ and no packet is waiting. If $c \geq 0$ and $t < a_{p_{i,j}}$, then the queue is empty before $a_{p_{i,j}}$, so, according to Rule 1, the credit credit was reset and maintained at zero until the packet arrives at $a_{p_{i,j}}$. Likewise, if $c < 0$ but the recovered credit would become zero at $t_0 = t - c/\alpha_a^+ < a_{p_{i,j}}$, the queue is still empty at t_0, so the shaper holds the credit at zero until $a_{p_{i,j}}$. In both situations, the credit is zero at $a_{p_{i,j}}$ and starts to increase with rate α_a^+ from $a_{p_{i,j}}$ to σ, thus yielding a credit of $\alpha_a^+(\sigma - a_{p_{i,j}})$ at time σ. This proves the first case of this lemma.

Case 2: A packet is waiting when the credit becomes non-negative. If neither of the above conditions holds, then the time when the credit becomes non-negative occurs at or after $a_{p_{i,j}}$, meaning a packet is already queued. Hence, CBS does not reset the credit, which increases linearly with slope α_a^+ from its initial value c at time t to reach $\alpha_a^+(\sigma - t) + c$ at time σ. This proves the second case of this lemma. □

EXAMPLE 6.3

To illustrate the behaviour of the transformation function g, we continue the scenario from Figure 6.1. Assume the system is in state s' and the next eligible packet is $p_{2,2}$. Since $p_{2,2}$ arrives in $[7, 8]$ and the egress port is available in

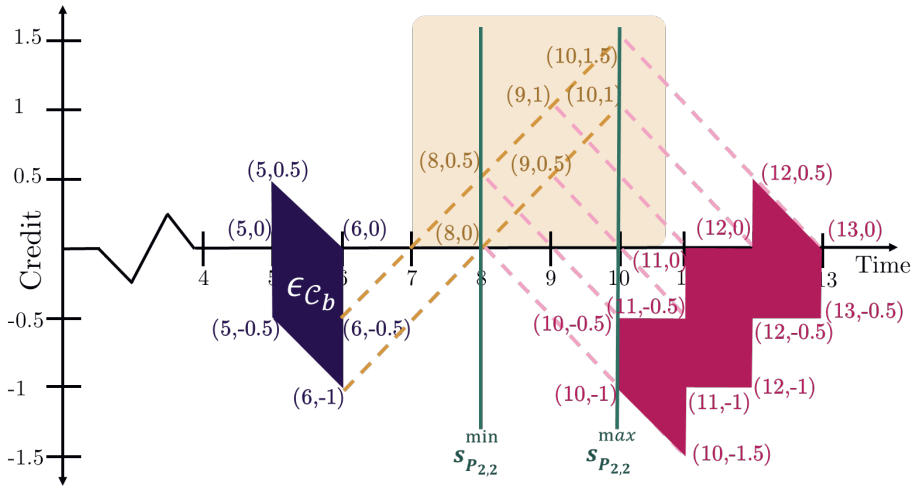


Figure 6.3: Example showing how g_{C_2} projects the coordinates in ϵ_{C_b} from Example 6.2 to the possible start times of $p_{2,2}$ (blue polygon projected to the yellow region), and how h_{C_2} projects the output of g_{C_2} to the possible finish times of $p_{2,2}$ (Yellow region projected to the pink polygon)

$[8, 10]$, Equations (6.7) and (6.10) give $t^{\text{lb}}_{p_{i,j}} = 8$ and $t^{\text{ub}}_{p_{i,j}} = 10$.
The function $g_{C_a}(t, c, a_{p_{i,j}}, \sigma)$ is applied to all coordinates in $\epsilon_{C_b} = \{(5, 0.5), (5, 0), (5, -0.5), (6, 0), (6, -0.5), (6, -1)\}$, with $a_{p_{i,j}} \in [7, 8]$ and $\sigma \in [8, 10]$. The resulting transformed coordinates are shown in Figure 6.3. All three behaviours of the CBS shaper appear:
Credit drops or remains at zero: For $(5, 0.5), (5, 0), (6, 0)$, the credit either falls to zero or stays at zero because, at times 5 and 6, the C_b queue is empty. This corresponds to the case $t < a_{p_{i,j}} \wedge c \geq 0$.
Credit reaches zero and stays there until arrival For $(5, -0.5)$, the credit becomes zero at time 6 and remains zero because the queue is still empty. This matches the condition $t - \frac{c}{\alpha^+_b} < a_{p_{i,j}} \wedge c \leq 0$, ensuring no credit increase before the packet arrives.
Credit becomes non-negative, then accumulates For $(6, -0.5)$ and $(6, -1)$, the credit reaches zero at times 7 and 8, respectively. A packet may already be queued at those times, so credit may begin to accumulate. In the case of $(6, -0.5)$, the credit becomes non-negative at time 7, but the packet may or may not have arrived yet (arrival is only guaranteed by time 8). Thus, two possibilities must be considered: one where the packet is present at time 7 and one where it only arrives at time 8. In both cases, once the packet is certainly present, the credit increases as expected.

Now that we have determined the credit values at all possible start times in

$[\sigma^{\min}_{p_{i,j}}, \sigma^{\max}_{p_{i,j}}]$, we proceed to the second step: computing the credit at the finish times based on the transmission duration $C_{p_{i,j}} \in [\hat{C}^{\min}_{p_{i,j}}, \hat{C}^{\max}_{p_{i,j}}]$ of packet $p_{i,j}$. According to CBS, once a transmission begins, the credit reduces according to the send slope α^-_a. If the state at the start of transmission is (t, c) and the packet transmits for $C_{p_{i,j}}$ time units, then the time advances to $t + C_{p_{i,j}}$ while the credit decreases by $\alpha^-_a \cdot C_{p_{i,j}}$. The function

$$h_{C_a}(t, c, C_{p_{i,j}}) = \left(t + C_{p_{i,j}},\ \alpha^-_a \cdot C_{p_{i,j}} + c\right) \tag{6.16}$$

captures this update, yielding the new credit–time coordinate at the end of the transmission interval.

EXAMPLE 6.4

Following the application of the transformation function g_{C_a}, we now demonstrate how the resulting coordinates are further processed using the function h_{C_a} for the same example as in Example 6.4. For each output coordinate (t, c) produced by g_{C_a} (marked separately in the Figure), we evaluate all possible transmission durations $C_{p_{i,j}} \in \left[C^{\min}_{p_i}, C^{\max}_{p_i}\right]$.
The function h_{C_a} computes the finish-time coordinate of a packet, based on the current time t, credit c, shaping parameter α^-_a, and transmission duration $C_{p_{i,j}}$. This results in a set of new coordinates, each representing a possible finish time and credit value after transmitting a packet with duration $C_{p_{i,j}}$.
The complete set of such coordinates, obtained by applying h_{C_a} to every $g(t, c, a_{p_{i,j}}, \sigma)$ and all values of $C_{p_{i,j}} \in \left[C^{\min}_{p_i}, C^{\max}_{p_i}\right]$, forms the set of finish-time coordinates for the corresponding class. This transformed set is given by the space marked in pink in Figure 6.3

Using the functions g_{C_a} and h_{C_a}, the updated credit-time space for class C_a in the new state s' is computed as

$$\epsilon_{C_a}(s') = \left\{ h_{C_a}\left(g_{C_a}(t, c, a_{p_{i,j}}, \sigma); C_{p_{i,j}}\right) \;\middle|\; \begin{array}{l} (t, c) \in \epsilon_{C_a}(s), \\ a_{p_{i,j}} \in \left[a^{\min}_{p_i}, tr^{\max}_{C_a}\right], \\ \sigma \in \left[\sigma^{\min}_{p_{i,j}}, \sigma^{\max}_{p_{i,j}}\right], \\ C_{p_{i,j}} \in \left[C^{\min}_{p_i}, C^{\max}_{p_i}\right] \end{array} \right\} \tag{6.17}$$

Theorem 6.9. *The transformation from the credit-time space $\epsilon_{C_a}(s)$ in state s to the updated space $\epsilon_{C_a}(s')$ in state s' captures all possible transmission scenarios that occur after the transmission of a packet.*

Proof. The transformation is carried out in two stages:

1. The function $g_{C_a}(t, c, a_{p_{i,j}}, \sigma)$ maps each coordinate $(t, c) \in \epsilon_{C_a}(s)$ to an intermediate coordinate at the start of $p_{i,j}$'s transmission, considering the

evolution of C_a's credit from t to the start time σ assuming that the packet entered C_a's FIFO queue at $a_{p_{i,j}}$ (see Lemma 6.8). Lemma 6.8 proves that the function $g_{C_a}(t, c, a_{p_{i,j}}, \sigma)$ mimics exactly the CBS behaviour for all possible cases (i.e., whether the FIFO queue is empty or not when the credit becomes non-negative at or after t).

2. The function $h_{C_a}(t, c, C_{p_{i,j}})$ then updates this intermediate coordinate by simulating the transmission of the packet over a duration $C_{p_{i,j}}$. In line with the CBS protocol, the credit is decreased at a rate equal to the send slope α_a^-. This correctly accounts for the consumption of credit during transmission.

By quantifying over all admissible values of $a_{p_{i,j}}$, σ, and $C_{p_{i,j}}$, namely, $a_{p_{i,j}} \in \left[a_{p_i}^{\min}, tr_{C_a}^{\max}\right], \sigma \in \left[\sigma_{p_{i,j}}^{\min}, \sigma_{p_{i,j}}^{\max}\right], C_{p_{i,j}} \in \left[C_{p_{i,j}}^{\min}, C_{p_{i,j}}^{\max}\right]$, we ensure that every possible variation in the packet arrival, start time, and transmission duration is considered. Consequently, the set $\epsilon_{C_a}(s')$ includes all valid time-credit combinations for class C_a in the new state s'. Thus, the transformation completely characterises the updated credit-time space and, by extension, all possible transmission scenarios resulting from the transmission of a packet from state s. □

6.5 Merging of States

When building a graph that explores every possible scheduling decision, state space explosion can occur due to the large number of states being explored. To mitigate this, states with similar content can be merged because similar states yield similar future scheduling decisions.

Two states, $\mathcal{S}_1$ and $\mathcal{S}_2$, are considered similar if they satisfy the following criteria:

1. The set of transmitted packets is identical, i.e., $\mathcal{T}(\mathcal{S}_1) = \mathcal{T}(\mathcal{S}_2)$.

2. The availability intervals of the egress port are either overlapping or contiguous, i.e.,

$$\begin{matrix}[A^{\min}(\mathcal{S}_1), A^{\max}(\mathcal{S}_1)] \\ \cup \\ [A^{\min}(\mathcal{S}_2), A^{\max}(\mathcal{S}_2)]\end{matrix} = \left\{ \begin{matrix}[\min\{A^{\min}(\mathcal{S}_1), A^{\min}(\mathcal{S}_2)\}, \\ \max\{A^{\max}(\mathcal{S}_1), A^{\max}(\mathcal{S}_2)\}]\end{matrix} \right\}$$

If states $\mathcal{S}_1$ and $\mathcal{S}_2$ meet these criteria, they are merged into a single state $\mathcal{S}'$ defined as follows:

1. The set of finished jobs is kept identical, $\mathcal{T}(\mathcal{S}') = \mathcal{T}(\mathcal{S}_1) = \mathcal{T}(\mathcal{S}_2)$.

2. The egress port availability interval is the union of the intervals, $A^{\min}(\mathcal{S}') = \min\{A^{\min}(\mathcal{S}_1),\, A^{\min}(\mathcal{S}_2)\}$, and $A^{\max}(\mathcal{S}') = \max\{A^{\max}(\mathcal{S}_1),\, A^{\max}(\mathcal{S}_2)\}$.

3. The credit–time space of the merged node $\mathcal{S}'$ is defined on a per-class basis. For each traffic class $\mathcal{C}_a$, we take the union of the corresponding credit-time spaces from $\mathcal{S}_1$ and $\mathcal{S}_2$, $\epsilon_{\mathcal{C}_a}(\mathcal{S}') = \epsilon_{\mathcal{C}_a}(\mathcal{S}_1) \cup \epsilon_{\mathcal{C}_a}(\mathcal{S}_2)$.

The merged state $\mathcal{S}'$ replaces both nodes $\mathcal{S}_1$ and $\mathcal{S}_2$, and all the edges incoming to either node are redirected to $\mathcal{S}'$.

6.6 Proof Of Correctness

In the previous sections, we proved that a single scheduling decision in the SAG is modelled correctly. We now extend this to the entire graph. We prove that the SAG explores all possible scheduling decisions and their combinations, ensuring that every valid transmission scenario corresponds to some path in the graph generated by Algorithm 4. We further show that this construction yields safe lower and upper bounds on the latency of every packet in all scenarios.

Theorem 6.10. *The SAG explores all reachable concrete system states across all transmission scenarios in* Γ*.*

Proof. We prove by induction over the layers of the SAG that every concrete system state reachable by a transmission scenario in Γ is represented by some abstract state s at each layer of the graph.

Let L^l be the l^{th} layer of the SAG. For an abstract state $s \in L^l$, we define $\Gamma(s)$ as a superset of concrete transmission scenarios $\gamma \in \Gamma$ such that the concrete system state after exactly l packet transmissions is contained by s. We aim to show that

$$\bigcup_{s \in L^l} \Gamma(s) \supseteq \Gamma \quad \text{for all } l = 0, 1, 2, \ldots$$

Base case ($l = 0$): At the start of all transmission scenarios, no packets have been transmitted and the egress port is free. This only possible concrete initial system state for all transmission scenarios is represented by the initial abstract state $s \in L^0$. Hence,

$$\Gamma(s) \supseteq \Gamma \quad \Rightarrow \quad \bigcup_{s \in L^0} \Gamma(s) \supseteq \Gamma.$$

Induction Step ($l \mapsto l + 1$): Assume that at layer l,

$$\bigcup_{s \in L^l} \Gamma(s) \supseteq \Gamma$$

i.e., for every transmission scenario $\gamma \in \Gamma$, the concrete system state reached after the transmission of l packets is represented by some abstract state $s \in L^l$.

Consider any such abstract state $s \in L^l$ and any transmission scenario $\gamma \in \Gamma(s)$. Let $p_{i,j}$ be the packet transmitted next in the scenario γ. By the necessary conditions for packet transmission established in Theorem 6.7, packet $p_{i,j}$ satisfies the eligibility conditions implied by the abstract information stored in s. Algorithm 4 therefore includes $p_{i,j}$ in the set of eligible packets considered at Line 8.

For this eligible packet $p_{i,j}$, Algorithm 4 constructs a successor abstract state $s' \in L^{l+1}$. By construction, s' represents the concrete system state reached after transmitting $p_{i,j}$ in scenario γ. Any merging of successor states preserves this representation.

Consequently, for every transmission scenario $\gamma \in \Gamma(s)$, there exists at least one successor abstract state $s' \in L^{l+1}$ such that $\gamma \in \Gamma(s')$. Hence,

$$\bigcup_{s' \in L^{l+1}} \Gamma(s') \supseteq \bigcup_{s \in L^l} \Gamma(s) \supseteq \Gamma.$$

Conclusion: By induction, each layer L^l of the SAG represents all concrete system states reachable after l packet transmissions in all transmission scenarios in Γ. Therefore, across all its layers, the SAG explores all reachable concrete system states in all transmission scenarios in Γ. □

Theorem 6.11. *Algorithm 4 returns safe bounds on the* BCRT *and* WCRT *of every packet.*

Proof. By Theorem 6.10, for every transmission scenario in Γ and every packet transmission in those scenario, the resulting concrete system state is represented by some abstract state in the SAG. For any abstract state s and any packet $p_{i,j}$ transmitted in that state, Algorithm 4 computes values $f_{p_{i,j}}^{\min}$ and $f_{p_{i,j}}^{\max}$ that bound the finish time of $p_{i,j}$ for all concrete system states represented by s (Line 14). The algorithm then updates the global best- and worst-case latencies of $p_{i,j}$ by taking, respectively, the minimum and maximum of the relative finish times observed across all abstract states (Lines 15 and 16).

Since every concrete transmission is represented in the SAG and the finish times computed at each abstract state bound the corresponding concrete transmissions, taking the minimum and maximum over all abstract states yields safe lower and upper bounds on the BCRT and WCRT of $p_{i,j}$ across all transmission scenarios in Γ. □

6.7 Improving the Analysis Scalability

Implementing the SAG analysis from Algorithm 4 introduces challenges not present in previous SAG-based work. Earlier analyses tracked only scalar bounds on time-intervals, whereas our method must store and update two-dimensional time-credit spaces for each class, i.e., the sets ϵ_{C_a} for all C_a. This substantially increases memory usage and run time, as each state may contain large (possibly unbounded) number of time-credit pairs. Moreover, unlike prior systems, no finite observation window can be easily defined to guarantee completeness of the state-space analysis. Without such a bound, the analysis could explore infinitely many packet arrivals, making it impractical. To address these issues, we propose two improvements to the analysis: (1) we adopt a compact state representation that approximates the content of the sets ϵ_{C_a} to optimize runtime and memory storage while keeping the analysis safe, and (2) we introduce a safe termination mechanism that replaces the need for an explicit observation window.

6.7.1 Optimising State Representation

When constructing a new state, each class must store a credit–time region ϵ_{C_a}. Storing this region explicitly to preserve exactness is inefficient and impractical, so we apply a controlled over-approximation based on the structure of the SAG updates. Since ϵ_{C_a} contains all possible credit–time pairs arising from all feasible finish times of the most recently transmitted packet of class C_a, the interval $[f^{\min}_{p_{i,j}}, f^{\max}_{p_{i,j}}]$ provides a safe and tight bound on the time interval in which ϵ_{C_a} is contained. By construction, no finish time generated by the analysis can lie outside this interval, so it fully captures the admissible domain of ϵ_{C_a}.

To obtain a safe vertical bound for the new region, we exploit the fact that all credit-time trajectories evolve exclusively along the idle slope or the send slope. Because these are the only slopes used when transforming the region, projecting every coordinate of ϵ_{C_a} along the idle slope to $x = f^{\min}_{p_{i,j}}$ and $x = f^{\max}_{p_{i,j}}$ yields conservative lower and upper credit limits for the new state. The minimum and maximum of these projected values safely enclose all possible credit levels at $f^{\min}_{p_{i,j}}$ and $f^{\max}_{p_{i,j}}$, thereby defining the vertical extent of the over-approximating parallelogram. The projected credit intervals on $f^{\min}_{p_{i,j}}$ and $f^{\max}_{p_{i,j}}$ define a parallelogram that safely encloses the entire region ϵ_{C_a} and captures all credit-time pairs relevant to future transitions for the newly created state (see Figure 6.4). Consequently, each region ϵ_{C_a} of a newly created state can be encoded compactly using four values that define the four corners of the parallelogram. This yields a sound and efficient representation for storing credit-time regions in each abstract state (see Figure 6.4).

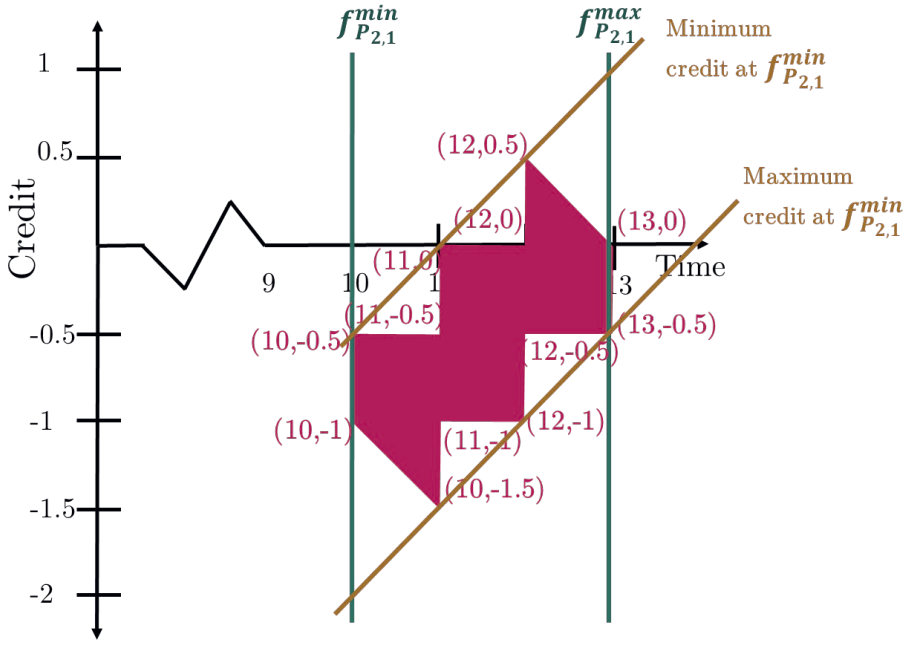


Figure 6.4: Over-approximating the set of credit–time coordinates into a compact parallelogram representation

When two states are merged, their credit-time regions must be safely combined. Using the parallelogram representation, this is simple: the merged state forms a finish-time interval that spans both states and a credit interval that spans both projected credit ranges. The resulting parallelogram encloses both original regions, ensuring the merge remains a sound over-approximation of the original regions.

Because the abstract state is constructed using an over-approximation, and due to the merging rules applied, this approach indeed analyses additional transmission scenarios that may not actually occur. However, since all real transmission scenarios are still covered, and the over-approximation only adds extra, non-existent paths, the analysis remains sound.

EXAMPLE 6.5

In Example 6.4, we derived a final set of coordinates that a state must retain after transmitting a packet. Figure 6.4 shows how this set is approximated. The earliest and latest finishing times, marked by the vertical green lines, form the first interval: $[f^{\min}_{p_{i,j}}, f^{\max}_{p_{i,j}}] = [10, 13]$. Projecting all coordinates back to $f^{\min}_{p_{i,j}}$ along the idle slope yields the second interval $[-2, -0.5]$. These two intervals, together with the idle slope, define the parallelogram that replaces

the full list of coordinates.

6.7.2 Safe Termination in the Absence of an Observation Window

Previous SAG analyses relied on the existence of safe observation windows, where a safe observation window is defined as a time interval such that if no deadline is missed in that interval then it can be proven that no deadline will be missed in any other time interval. Since, to the best of our knowledge, there does not exist a proven bound on the length of a safe observation window for CBS, we adopt a different technique to decide when it is safe to stop the analysis, that is we detect convergence across hyperperiods. Since we assume periodic flows, we define the hyperperiod of the system as the time duration after which packet arrivals repeat. After each hyperperiod, we record the set of abstract states reached by the SAG at the current layer. We then compare this set with the sets of abstract states recorded at the end of previous hyperperiods.

Let H denote the hyperperiod. The k^{th} hyperperiod boundary is the time instant $k \times H$, $k \in \mathbb{N}$. Let s and s be abstract states reached at the k_1 and k_2 hyperperiod boundaries, respectively. We say that s is subsumed by s iff (i) for every traffic class C_a and every $(t, c) \in \epsilon_{C_a}(s)$, there exists $(t', c') \in \epsilon_{C_a}(s)$ such that $c = c'$ and $t - k_1 H = t' - k_2 H$, (ii) for every flow $F_i \in \mathcal{F}$, $\min\{a^{\min}_{p_{i,j}} \mid p_{i,j} \notin \mathcal{T}(s)\} - k_1 H = \min\{a^{\min}_{p_{i,p}} \mid p_{i,p} \notin \mathcal{T}(s)\} - k_2 H$ and (iii) $[A^{\min}(s), A^{\min}(s)] \subseteq [A^{\min}(s), A^{\min}(s)]$ [1].

If, for every abstract state reached at the current hyperperiod boundary, there exists a previously recorded state that subsumes it under these criteria, then all future scheduling decisions that may arise from the current states have already been explored, and the SAG analysis can safely terminate. Otherwise, any newly discovered states are added to the recorded set, and exploration continues. This guarantees safe termination without requiring a known observation window.

6.8 Experimental Results

In the previous section, we introduced a scalable analysis for evaluating TSN networks shaped by CBS. Since this scalability relies on deliberate over-approximations, we now assess its impact on accuracy by presenting the schedulability results

[1] Note that we do not need to explicitly compare the concrete content of the queues in states s and s because their content at the hyperperiod boundary is directly implied by the the arrival times of the packets that are still pending at the beginning of the next hyperperiod. Thus, if all flows have the same relative arrival time intervals for their next undispatched packet, the content of the queues will be identical too.



Figure 6.5: Network architecture with a single switch

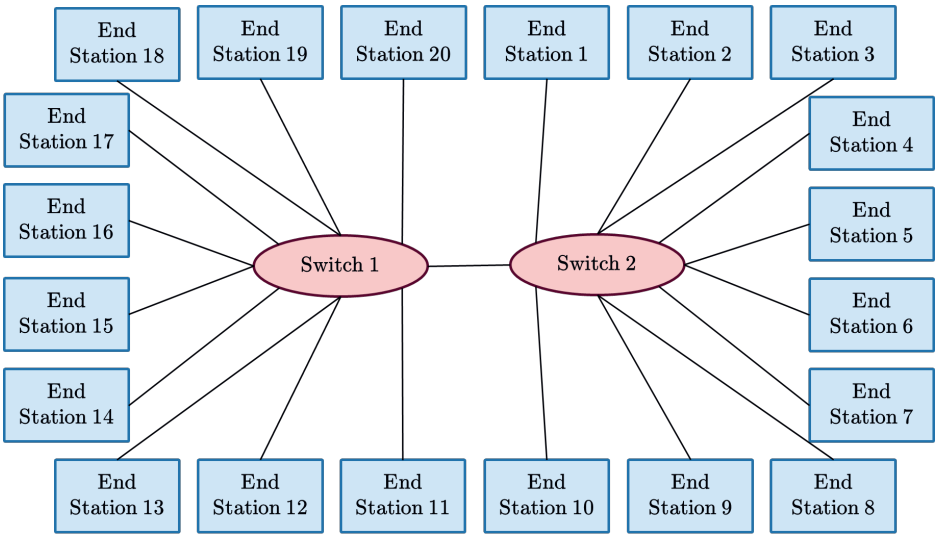
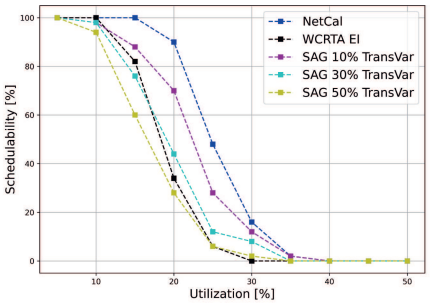


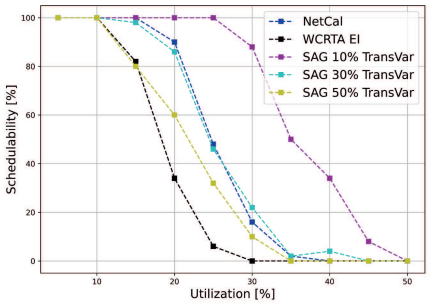
Figure 6.6: Network architecture with two switches

obtained using our method. We compare our analysis against two existing approaches, the state-of-the-art worst-case response-time analysis based on eligible intervals (Worst-Case Response-Time Analysis based on Eligible Intervals (WCRTA-EI)) [13] and a network-calculus-based method (Network Calculus (NetCal)) [66], across two network configurations. For these comparisons, only two CBS-shaped classes are considered, as the implementations of these two analyses provided by the authors of [13] and published in [12] only support at most two such classes, even though the theoretical analyses published in [13, 66] can handle more. In addition, we also study how the number of traffic classes affects schedulability using our own analysis by systematically varying the class count. We begin by describing the experimental setup and flow-generation process, followed by the experimental results.

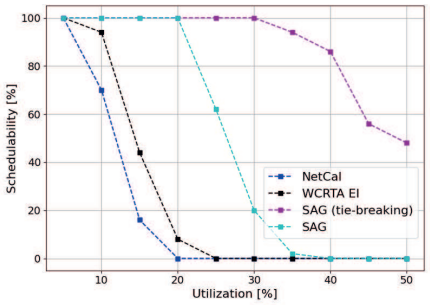
We employ the network and traffic generator described in [12], which enables systematic creation of both network topologies and traffic configurations. Two network setups are considered in our experiments, shown in Figure 6.5 and Figure 6.6. All network traffic is shaped by a Credit-Based Shaper (CBS). The generator produces up to 100 traffic flows transmitted over full-duplex links with a bandwidth of 100 Megabits per second (Mbps). Message sizes range from 500 to 1500 bytes. All traffic flows are periodic, with a maximum period of 10 000 µs, and



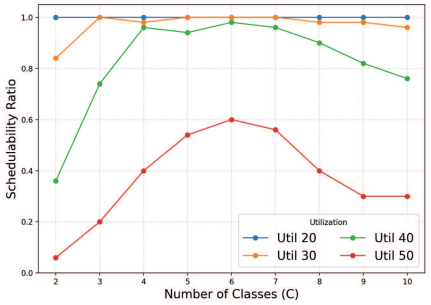
(a) Schedulability ratio for the network in Figure 6.5



(b) Schedulability ratio for the network in Figure 6.5 with tie-breaking mechanism



(c) Schedulability ratio for the network in Figure 6.6



(d) Impact of the number of traffic classes on schedulability

Figure 6.7: Experimental Results

each flow's deadline is equal to its period. For each experimental configuration, 50 unique sets of communication flows are generated.

We assume known arrival patterns and phasing in all experiments. Traditionally, such traffic is shaped using Time-Aware Shapers (TAS), which assign fixed transmission windows. However, this can waste bandwidth, since windows must be conservatively sized and may not be fully utilized. Using CBS instead avoids this inefficiency while still providing deterministic shaping, making it a valid and effective choice in our context.

We present two variants of the SAG analysis. The first uses Equation (5.3), which does not assume any tie-breaking rule between packets with equal arrival when deciding which packets may be on top of the FIFO queue. The second assumes that when two packets arrive at exactly the same time in the FIFO queue, the tie is resolved deterministically by selecting the packet belonging to the communication flow with the smaller flow identifier to be inserted first in the queue (i.e., if two or more packets of C_a have $a^{\min}_{p_{i,j}} = tr^{\max}_{\mathcal{C}_a}$, then only the packet with smallest flow ID is added to top_{C_a}).

Each experiment was executed on a Linux platform and required from a few tens to several hundreds of milliseconds and consumed on the order of a

dozen megabytes of memory. We do not, however, report runtime and memory usage comparisons with [13] and [66], since their implementations could not be executed on the same platform. Indeed, the implementation of [13] and [66] were provided to us in the form of a Windows executable that we ran on a much less powerful laptop. Thus, a comparison of runtime would have been unfair for the state-of-the-art.

Evaluation of a single switch full-duplex network. We compare our proposed analysis against two existing approaches: a worst-case response time analysis based on eligible intervals (WCRTA-EI) [13] and a network calculus–based method (NetCal) [66]. Both WCRTA-EI and NetCal assume that all packets generated by the same traffic flow have the same size sizes. In contrast, our analysis allows to account for uncertainty in packet transmission durations. We consider three levels of transmission duration variation in this experiment: 10%, 30%, and 50%. The generated traffic is divided into two traffic classes, each allocated 50% of the available bandwidth and shaped by a Credit-Based Shaper. Each communication flow may have release jitter with a 50% probability. Flows with release jitter have their jitter randomly chosen within the range of 1-100 µs.

As shown in Figure 6.7a, the generic SAG analysis achieves schedulability results comparable to WCRTA-EI, sometimes better, sometimes worse depending on the transmission-duration variation. Figure 6.7b shows the results when tie-breaking information is exploited. In this case, SAG achieves higher schedulability, demonstrating that additional system knowledge can be effectively leveraged to reduce pessimism. This also highlights a key advantage of SAG: unlike WCRTA-EI and NetCal, it can incorporate such implementation-specific information to refine the analysis and improve accuracy.

Evaluation of a Multi-Switch Network. To further evaluate scalability, we consider the larger network topology in Figure 6.6. Again we compare with WCRTA-EI and NetCal as baselines. In this experiment, our SAG analysis includes a 10% transmission-duration variation, while both baselines assume fixed packet sizes.

As shown in Figure 6.7c, SAG consistently achieves a higher schedulability ratio than both reference methods with both the generic analysis without tie-breaking and the variant exploiting tie-breaking information. In both cases, SAG outperforms the state of the art, while incorporating tie-breaking further improves accuracy by reducing pessimism in the analysis. However, we note that the tie-breaking rule may not be known in most cases.

Multi-switch support is achieved by executing SAG iteratively along each flow's route. At each switch, SAG computes best- and worst-case completion times and forwards them as arrival-time jitter bounds for the packet in the next switch. This process continues until each packet has been analysed along its entire path. For acyclic networks like the one of Figure 6.6, this iterative procedure enables SAG

to evaluate multi-hop topologies in a straightforward way.

[13] and [66] are offset-agnostic and assume unknown phasing between flows. This is both a strength and a limitation, as it enables support for a broader class of communication patterns, including sporadic traffic, but prevents exploiting known phasing and arrival information to derive tighter bounds. In contrast, our approach leverages this information to obtain tighter results. Prior work in switched Ethernet and AFDX networks [1, 42, 43, 56, 11] has demonstrated that incorporating end-system scheduling and flow offsets into worst-case analysis, using techniques like Network Calculus and timed-automata-based modelling, can significantly reduce pessimism in worst-case delay bounds. However, these techniques have not been applied to CBS so far.

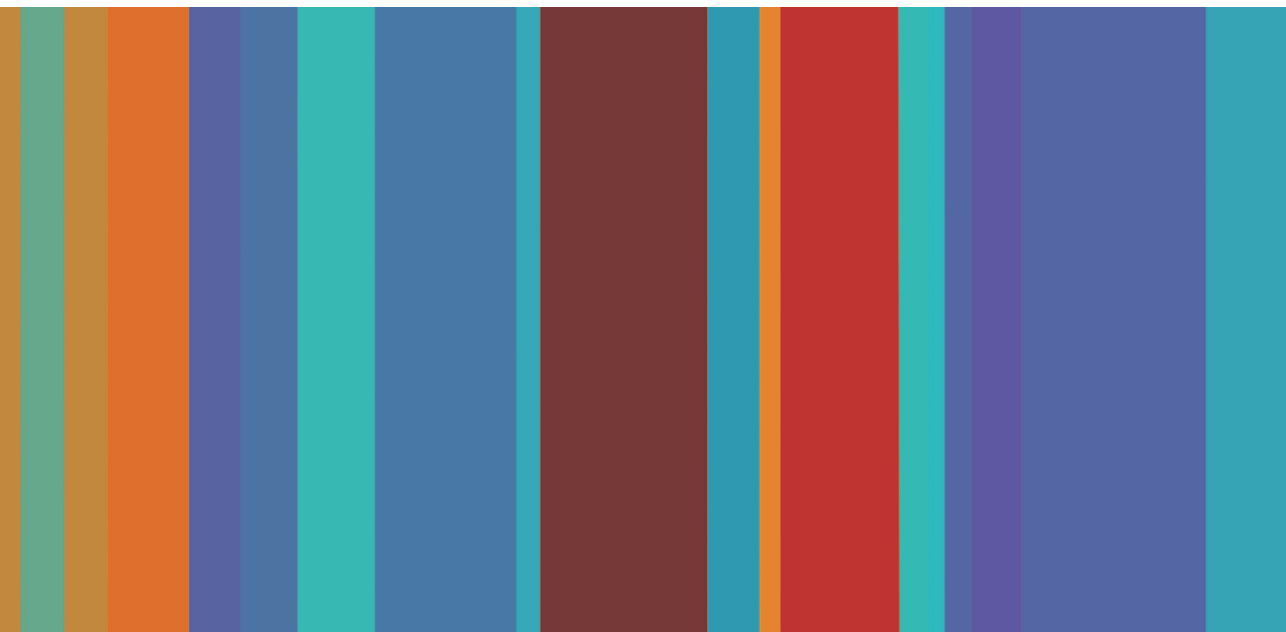
Exploring the Impact of the Number of Traffic Classes. In this experiment, we evaluate how the number of traffic classes affects schedulability. Starting from a configuration with two CBS-shaped classes, we increase the number of classes from two up to ten by randomly reassigning flows while keeping all other parameters fixed. Each class is assigned an idle slope proportional to the total traffic load mapped to that class.

As shown in Figure 6.7d, schedulability initially increases as the number of classes grows, since fewer flows share each FIFO queue and interference is reduced. When too many classes are introduced, however, each class receives a smaller fraction of the available bandwidth, leading to a decline in schedulability. In the configuration studied, the highest schedulability is observed with six classes, representing a balance between reduced contention and sufficient bandwidth per class.

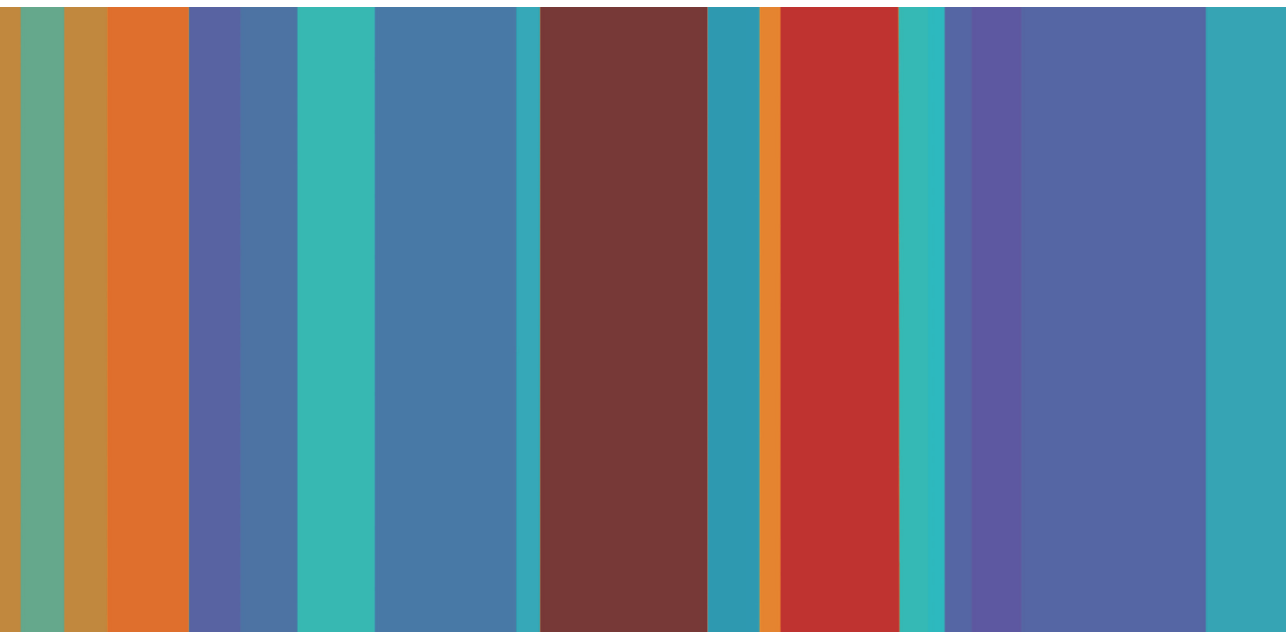
6.9 Conclusion

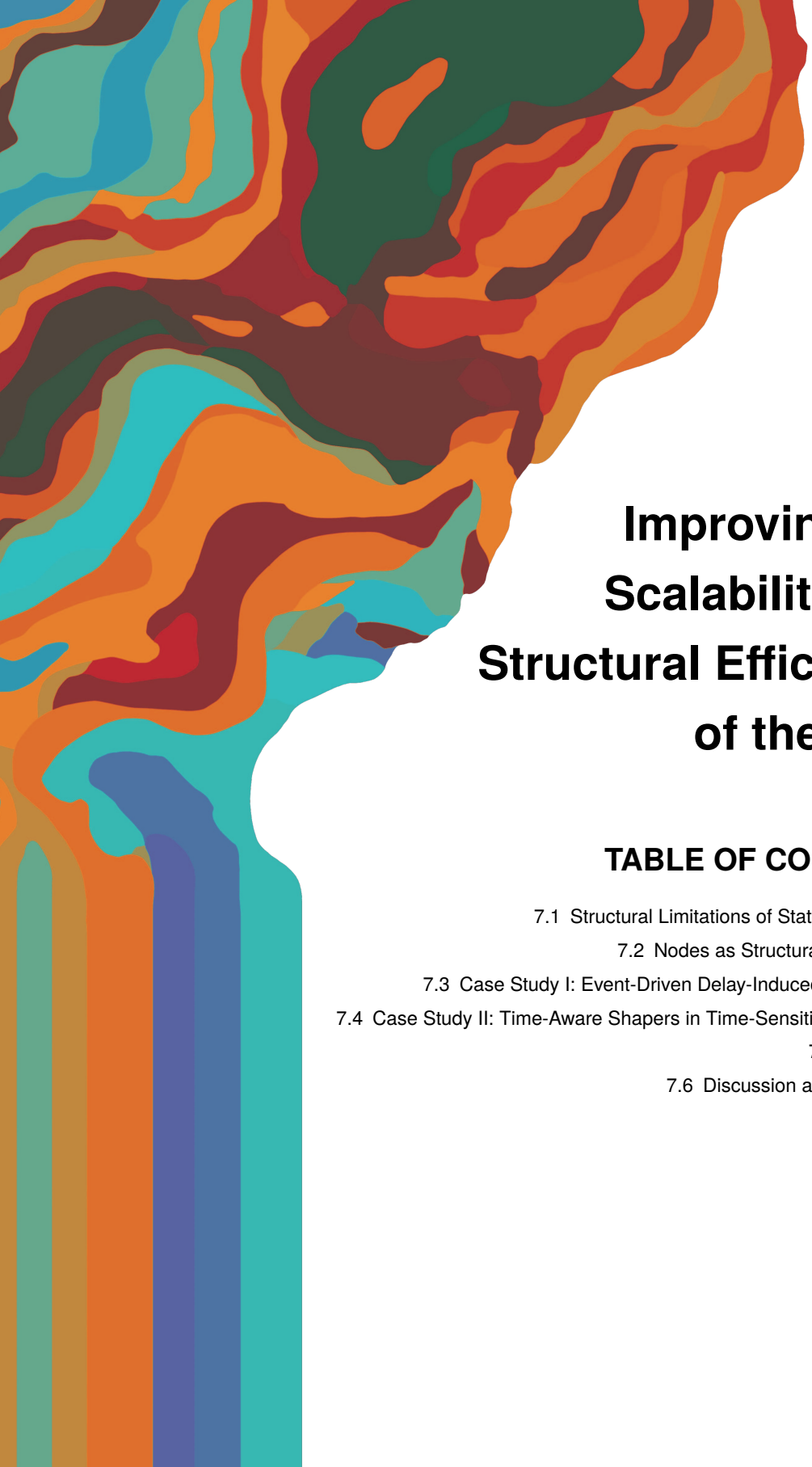
This work extends the Schedule Abstraction Graph (SAG) methodology to analyse communication flows with a known arrival pattern shaped by Credit-Based Shaping (CBS). We introduced a scalable and flexible schedulability analysis for CBS-based TSN networks and showed through comparison with existing methods that it maintains accuracy, outperforming state-of-the-art analyses, especially for multi-switches networks. These experiments show that the analysis proposed in this paper properly balances scalability with accuracy. Future work will integrate this analysis with existing approaches for Time-Aware Shaping in Chapter 5 to provide a unified framework covering the major TSN shapers. We also plan to extend the method to incorporate pre-emptive scheduling effects reusing results of [29].

PART III



Improving Scalability and Structural Efficiency of the Schedule Abstraction Graph





7

Improving the Scalability and Structural Efficiency of the SAG

TABLE OF CONTENTS

7.1 Structural Limitations of State-Centric SAG

7.2 Nodes as Structural Abstractions

7.3 Case Study I: Event-Driven Delay-Induced (EDD) Tasks

7.4 Case Study II: Time-Aware Shapers in Time-Sensitive Networking

7.5 Evaluation

7.6 Discussion and Conclusion

The Schedule Abstraction Graph (SAG) addresses the combinatorial growth in execution scenarios by grouping together scheduling behaviours that are equivalent with respect to the analysis. This grouping is achieved through abstraction and merging. This reduces the number of abstract states that must be explored while preserving correctness. However, as the system models considered in this thesis become more expressive, the number of abstract states may still grow significantly.

Different abstract system states may encode the same scheduling configuration. In particular, two abstract states may differ in their timing information, such as when processors may become available or when previously dispatched jobs may complete, while still representing the same set of already dispatched jobs and the same set of jobs that are ready for execution.

This means that the information stored in an abstract state serves two different roles in the analysis. On the one hand, some information describe what has happened so far in the schedule, such as which jobs have already been dispatched, which dependencies have been resolved, or which execution choices remain possible. This information determines the logical scheduling situation, independent of the exact times at which events occurred. On the other hand, certain elements specify when events occur, such as arrival bounds for jobs or packets that have not yet been dispatched, completion times for dispatched jobs or packets, eligibility intervals as in Chapter 5, or when resources such as processors or egress ports will become available. These values are essential for timing analysis, but they do not necessarily change what must still be executed/transmitted.

In summary, each abstract state contains two types of information: information that depends only on the set of events that led to the state, but not on their ordering or precise timing, and information that depends on when these events occur. The timing-independent information is identical for many states explored by the SAG. Storing and processing that information separately in each abstract state is therefore redundant and leads to unnecessary memory usage and computational effort.

To address this, we introduce the notion of a node, which groups together abstract states that share the same timing-independent history. This allows the analysis to factor out the common scheduling information while still preserving the timing information associated with each abstract state.

It is important to emphasise that nodes do not replace states. States are still required to capture detailed timing information to ensure correctness of the analysis. Nodes provide an additional layer of organisation that separates scheduling history from timing variations.

The remainder of this chapter presents this idea in detail. It first examines the structural properties of states that make aggregation possible, then introduces

nodes as a means of capturing these shared properties, and finally discusses how this decomposition can be applied systematically to improve the scalability of the Schedule Abstraction Graph.

7.1 Structural Limitations of State-Centric SAG

The Schedule Abstraction Graph reduces the number of execution scenarios by grouping them together into abstract states and merging overlapping abstract states that share concrete states. However, as the system models considered in this thesis become more expressive, the number of abstract states can still grow rapidly. This is particularly evident in models with precedence constraints, suspension behaviour, and traffic shaping mechanisms such as TAS and CBS, where additional sources of variability lead to many distinct states.

To understand this growth, it is useful to examine what information is stored in a state. As seen in the previous chapters, a state typically captures:

- which jobs or packets have already been processed,
- which ones are currently active or pending,
- the availability of processing or communication resources,
- when some of the jobs/packets finished executing/being transmitted,
- and in the case of TSN, the state of shaping mechanisms.

All this information does not play the same role in the analysis.

Some parts of a state describe *what has happened* in the execution so far. For example:

- the set of jobs that have already been dispatched,
- which jobs are still pending,
- which jobs have their precedence relations fulfilled,
- which jobs are eligible to execute next.

Other parts of the state describe *when things happened*, such as:

- the intervals in which events happened,
- the resource availability intervals,
- the precise finish times of jobs or packet transmissions.

These values distinguish different timing evolutions of the same scheduling situation.

A key limitation of the state-centric representation is that both kinds of information are stored together in a single entity. As a result, states that represent the same progression in terms of what has happened, but differ only in when it happened, are treated completely separately.

This leads to several inefficiencies. First, information about what has happened is duplicated across many states. Multiple states may represent the same set of dispatched jobs and the same resolved dependencies, yet each state stores this scheduling history independently.

Second, the analysis must process each of these states separately, even when the same future scheduling choices follow from what has happened so far. This leads to repeated treatment of equivalent scheduling situations that differ only in their timing information.

This motivates the need for a representation that separates the history component from the timing component. Such a separation allows states with similar scheduling history to be grouped together, while still preserving the distinctions required for correct timing analysis.

The next section introduces this idea through the notion of nodes, which capture shared timing-independent scheduling histories and provide a more efficient organisation of the explored state space.

7.2 Nodes as Structural Abstractions

A node represents a shared scheduling history of the system. It captures all information about what has happened so far that determines which scheduling decisions are possible. In particular, a node represents properties such as:

- set of jobs or packets that have already been processed,
- set of jobs or packets that are still pending,
- set of jobs whose precedence constraints are satisfied,
- set of entities that are eligible for execution or transmission.

All abstract states within a node share this history information.

At the same time, different abstract states within the same node may differ in their timing-related information. These differences include:

- timing bounds of events,
- resource availability intervals,

- finish times of jobs or transmissions.

These values capture when relevant events occur within a fixed scheduling history and are therefore stored at the state level.

A node can thus be viewed as a collection of states that share similar scheduling history but differ in their timing behaviour. Organising the analysis around nodes changes how the state space is explored. Instead of treating each state independently, the exploration proceeds at the level of nodes. For each node, the set of possible scheduling decisions is determined once. Then, for each such decision, it is checked for each state in the node whether the decision still applies in that state.

This provides several advantages. First, history information is stored only once per node, rather than being duplicated across multiple states. This is particularly beneficial when the history contains large data structures, such as the sets of already dispatched jobs or transmitted packets, which can become very large in systems with many jobs or packets. Storing such information separately in every state would therefore be highly redundant. In addition, computations based on these data structures also need to be performed only once per node rather than once per state. As a result, expensive operations that parse or derive information from these structures are reduced, and the computed results can be cached at the node level for access by all states in the node.

Second, the evaluation of possible next dispatches can be decomposed into a node-level step and a state-level step. In the state-centric representation, each state had to be inspected separately to determine which jobs or packets might be eligible next. With nodes, all information that depends only on the shared scheduling history can be computed once per node, making it possible to determine the set of potentially ready jobs or packets for all states in that node at once. The analysis then considers the individual states only to resolve timing-dependent behaviour. This allows node-level information to be cached and reused across all contained states, and avoids recomputing the same scheduling logic for each state independently.

The next sections illustrate how this idea can be applied in practice through concrete case studies.

7.3 Case Study I: Event-Driven Delay-Induced (EDD) Tasks

To illustrate how nodes can be used to improve the structural efficiency of the Schedule Abstraction Graph, we consider the analysis of Event-Driven Delay-

Induced (EDD) tasks introduced in Chapter 3.

EDD tasks extend the basic task model by introducing precedence constraints and suspensions between dependent jobs. This leads to more complex execution behaviour, where the eligibility of a job depends not only on its arrival time but also on the completion of its predecessors and any associated delays.

7.3.1 State Representation in EDD Analysis

In the state-centric representation used in Chapter 3, each state captured the following information:

- the set of jobs $\mathcal{D}$ that have already been dispatched for execution on a processor until reaching a system state abstracted by s,
- m intervals $[A_i^{\min}, A_i^{\max}]$ for $i = 1, \ldots, m$ that model processor availability in the abstract state s,
- the finishing time interval $[f_j^{\min}, f_j^{\max}]$ for each job $j \in \mathcal{D}$ that has been dispatched on a processor and has at least one successor that still needs to be executed, and
- a set of at most m jobs $\chi \subseteq \mathcal{D}$ that are certainly executing in the system state abstracted by s.

7.3.2 Identifying History and Timing Components

Analysing the content of a state, we observe that the set of dispatched job is the only information that is time-independent. However, it is a rather large data structure when tens of thousands of jobs are being analysed. reducing the number of times it is saved has thus a big impact on memory consumption and data caching. Furthermore, the set of jobs dispatched directly implies the set of precedence constraints that are satisfied, and thus the set of jobs that eligible for execution. Calculating those sets only once per node instead of once per state has therefore a huge impact on the execution time of the analysis.

This information captures what has happened so far in the schedule and determines which jobs can be scheduled next.

Timing information includes:

- finish times of jobs,
- delay intervals between dependent jobs,
- timing bounds affecting job eligibility.

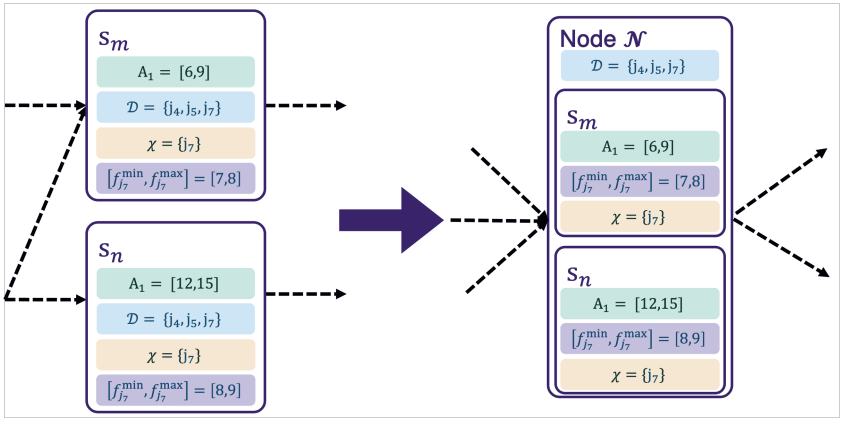


Figure 7.1: Two EDD states with the same scheduling history but different timing information are grouped into a single node.

These values affect when execution can proceed, but do not change the underlying scheduling history as long as the same jobs remain eligible.

7.3.3 Node Construction

Since all states within a node correspond to the same structural configuration, they share the same set of eligible (ready) jobs. As a result, this information does not need to be recomputed for each state, but can instead be computed once and stored at the node level.

Each node therefore represents:

- a fixed set of dispatched jobs,
- a fixed set of ready jobs.

Within each node, multiple states are maintained to capture different timing variations of this shared configuration.

This separation avoids redundant computation of structural information, while preserving the timing distinctions required for correct analysis.

EXAMPLE 7.1

Consider the two abstract states s_m and s_n shown in Figure 7.1, taken from a Schedule Abstraction Graph constructed for the EDD task model of Chapter 3. The two states contain the same dispatched set,

$$\mathcal{D} = \{j_4, j_5, j_7\}.$$

Hence, they agree on what has happened so far in the schedule.

However, they differ in their timing information. In particular, the processor-availability interval is $A_1 = [6, 9]$ in state s_m and $A_1 = [12, 15]$ in state s_n. Likewise, the finishing-time interval of job j_7 is $[f_{j_7}^{\min}, f_{j_7}^{\max}] = [7, 8]$ in s_m and $[f_{j_7}^{\min}, f_{j_7}^{\max}] = [8, 9]$ in s_n.
They may also differ in which jobs are certainly running at a given time. This information depends on the precise timing of events and is therefore part of the state-specific timing information.
In a purely state-centric representation, these two states would be stored and explored separately because their timing intervals differ. Nevertheless, from the perspective of scheduling history, they represent the same situation: the same jobs have already been dispatched.
This makes it possible to aggregate the two states into a single node, as illustrated in Figure 7.1. The node stores the shared history information, namely $\mathcal{D} = \{j_4, j_5, j_7\}$, while the state-specific timing information, including timing intervals and the set of certainly running jobs, remains attached to the individual states.
In this way, the shared scheduling history is represented only once, while the timing distinctions required for the analysis are preserved at the state level.

7.3.4 Summary of Information Distribution

The separation between node-level history information and state-level timing information for EDD tasks can be summarised as follows:

Information	Node	State	Reason
Dispatched set $\mathcal{D}$	✓		Records what has already happened
Certainly running jobs χ		✓	Records the currently running jobs
Availability intervals $[A_i^{\min}, A_i^{\max}]$		✓	Captures when processors may become available
Finish intervals $[f_j^{\min}, f_j^{\max}]$		✓	Captures when dispatched jobs may finish

This separation forms the basis for the node-based representation. Its impact on the efficiency of the analysis is evaluated in the Section 7.5, where we compare the performance of the state-centric and node-based variants.

7.4 Case Study II: Time-Aware Shapers in Time-Sensitive Networking

We now apply the same history–timing decomposition to the analysis of Time-Aware Shapers (TAS) introduced in Chapter 5. This demonstrates that the notion of nodes is not specific to task scheduling, but also applies more generally to communication systems analysed using the Schedule Abstraction Graph.

In TAS-based systems, packets are transmitted through queues that are regulated by gate control schedules. At any point in time, only certain queues are allowed to transmit, and packets must satisfy both eligibility conditions and gate constraints before they can be scheduled.

State Representation in TAS Analysis

In the abstract-state representation used in Chapter 5, each state stores:

- the set of packets that have already been transmitted, denoted by $\mathcal{D}$,
- an interval $[A^{\min}, A^{\max}]$ describing when the egress port may become available.

As in the EDD case, different states may share the same history while differing in their timing information.

Identifying History and Timing Components

From this representation, we can distinguish between history information and timing information.

History information consists of:

- set of packets that have already been transmitted.

This captures what has happened so far in the communication schedule.

Timing information consists of:

- interval $[A^{\min}, A^{\max}]$ describing when the egress port may become available.

This captures when transmission can continue.

Nodes can therefore be constructed by grouping together all states that share the same set of transmitted packets. Again, this is a rather big data structure that consumes a lot of memory and is very frequently accessed. Avoiding multiple copies of that data structure allows to save memory and improve data cache usage.

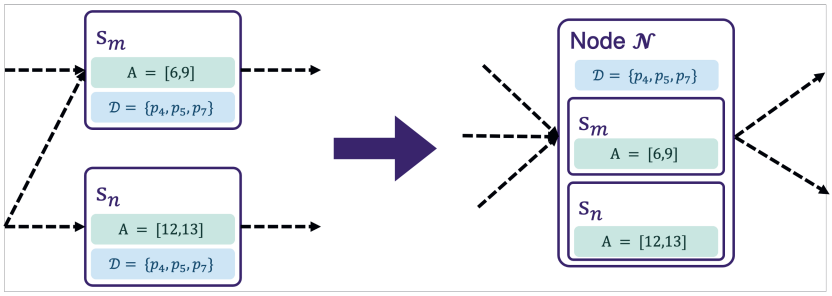


Figure 7.2: Two TAS states with the same transmission history but different egress-port availability intervals are grouped into a single node.

Within each node, multiple states may be maintained to capture different timing intervals for egress-port availability.

EXAMPLE 7.2

Consider the two abstract states s_m and s_n shown in Figure 7.2, taken from a Schedule Abstraction Graph constructed for the TAS model of Chapter 4. In both states, the same set of packets has already been transmitted, namely

$$\mathcal{D} = \{p_4, p_5, p_7\}.$$

Hence, the two states agree on what has happened so far in the communication schedule.

They differ, however, in their timing information. In state s_m, the egress-port availability interval is $A = [6, 9]$, whereas in state s_n, it is $A = [12, 13]$. Thus, the two states represent different temporal realisations of the same transmission history.

In a purely state-centric representation, these two states would be treated as distinct because their timing intervals differ. With the node-based representation, by contrast, the shared history information can be stored once at the node level, while the egress-port availability intervals remain attached to the individual states.

The two states can therefore be grouped into a single node, as illustrated in Figure 7.2. The node records the shared transmission history through $\mathcal{D} = \{p_4, p_5, p_7\}$, while the states s_m and s_n retain their different availability intervals. This illustrates how nodes allow aggregation of TAS states that agree on what has happened, even when they differ in when transmission may continue.

Information	Node	State	Reason
Transmitted packets $\mathcal{D}$	✓		Records what has already happened
Egress-port availability interval $[A^{\min}, A^{\max}]$		✓	Captures when transmission may continue

Summary of Information Distribution

The separation between node-level history information and state-level timing information for TAS can be summarised as follows:

This separation forms the basis for the node-based representation for SAG with Time-Aware Shapers. Similar to Section 7.3 its impact on the efficiency of the analysis is evaluated in the Section 7.5, where we compare the performance of the state-centric and node-based variants.

7.5 Evaluation

This section evaluates the impact of introducing structural aggregation through nodes on the efficiency of the Schedule Abstraction Graph. In this context, efficiency refers to both the computational cost of the analysis and its ability to scale to larger problem instances. We therefore study how the organisation of information within the analysis affects its performance, measured in terms of runtime, and how this in turn influences the scalability of the approach.

We compare three variants of the analysis:

- **v1**: a purely state-centric representation,
- **v2**: a node-based representation that groups structurally identical states,
- **v3**: an enhanced node-based representation with additional refinements to the exploration.

v1: State-Centric Representation The first variant (v1) uses a purely state-based representation. Each state encodes the complete system configuration, including both structural information (such as dispatched jobs and precedence status) and timing-related information (such as finish times and eligibility bounds).

v2: Node-Based Representation The second variant (v2) introduces nodes as a structural abstraction over states.

In this representation:

- states that share the same structural configuration are grouped into a node,

- each node represents a unique scheduling configuration,
- multiple states within a node capture different timing variations of that configuration.

The exploration is reorganised accordingly:

- possible scheduling decisions are determined at the node level,
- resulting states are stored within the corresponding successor nodes.

This change eliminates repeated evaluation of identical scheduling decisions across structurally equivalent states and reduces duplication in the explored graph.

v3: Node-Based Representation with Improved Hashing The third variant (v3) retains the node-based representation of v2, but strengthens the hashing used to retrieve the node associated with a given set of dispatched jobs. In particular, the lookup key incorporates additional identifying information, making the mapping to nodes more specific and reducing hash collisions during exploration.

This change does not alter the structural abstraction introduced in v2, but improves the efficiency of node and state retrieval during exploration.

The evaluation is designed to answer the following questions:

- How does the introduction of nodes affect the runtime of the analysis?
- How does it change the size of the explored graph, in terms of states and nodes?
- How do different versions of the analysis scale with increasing system complexity, such as higher utilisation, larger job sets, and richer structural dependencies?

To answer these questions, we consider two representative application domains that we discussed earlier in Sections 7.3 and 7.4:

- Event-Driven Delay-Induced (EDD) task systems, which capture computation-oriented scheduling problems,
- Time-Aware Shaper (TAS) systems, which model communication scheduling under timing constraints.

In each case, we evaluate how the different analysis variants behave under varying system parameters, and examine the relationship between runtime and the size of the explored representation.

7.5.1 Case Study I: Event-Driven Delay-Induced (EDD) Tasks

We evaluate three variants of the analysis, each differing in how the state space is represented and how scheduling decisions are evaluated during exploration.

The three variants differ as follows:

- v1 explores all states independently, without distinguishing structural similarity,
- v2 introduces nodes to group structurally identical states and perform exploration at the level of structural configurations,
- v3 extends v2 by limiting higher-priority interference checks to jobs that can affect the next dispatch decision, in addition to the improved hashing described above.

More precisely, when v3 determines whether a job can be scheduled next, it does not search arbitrarily far into the future for higher-priority interfering jobs. Instead, it only considers higher-priority jobs that can become relevant before the latest time by which a work-conserving scheduler must dispatch some job in the current exploration step. This bound is derived from the latest time at which a processor is certainly available and the latest time at which some unfinished job is certainly ready. Higher-priority jobs arriving beyond this bound are ignored for the current decision because they cannot influence the next dispatch. This reduces unnecessary interference checks without changing the analysis result.

Experimental Methodology The experimental setup for the EDD case study follows the task generation procedure described in Chapter 3. We briefly summarise the key aspects relevant to this evaluation.

We generate periodic implicit-deadline task sets without offsets. The number of tasks and processors is varied depending on the experiment. Tasks are generated either as segmented self-suspending tasks or as DAG-based tasks.

For segmented self-suspending tasks, task utilizations are generated using UUnifast [9] to match a target total utilisation. Task periods are drawn from a logarithmic distribution over the interval $[10, 10000]$ as in [40]. Each task is divided into multiple execution segments and suspension intervals, with the ratio between execution and suspension controlled by a parameter. Best-case values are set to 50% of the corresponding worst-case values.

For DAG-based tasks, task graphs are generated following the approach in [18, 44]. The probability of a node forking is set to 0.6, with a maximum of two nested forks and up to three parallel segments originating from each fork. For each utilisation level, multiple random task sets are generated to capture variability in the task structure.

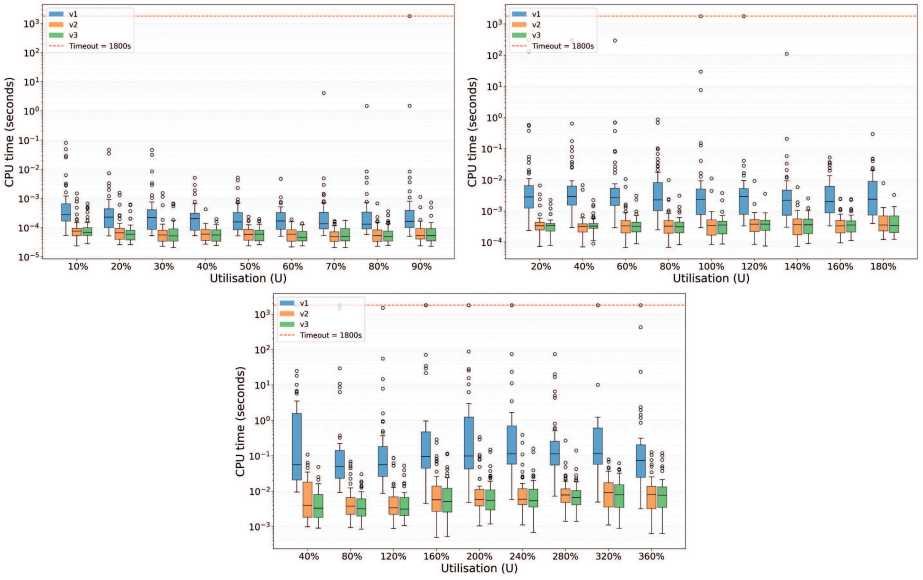


Figure 7.3: Runtime as a function of utilisation for self-suspending task systems with 2 (top left), 4 (top right), and 8 (bottom) processors.

All task sets are identical across the three analysis variants (v1, v2, and v3), ensuring a fair comparison. All experiments are executed on a server equipped with an Intel Xeon w7-3565X processor (32 cores, 64 threads, 82.5 MB shared cache), 256 GB of RAM, and M.2 storage.

Self-Suspending Tasks

Impact of Utilisation We first evaluate segmented self-suspending task systems with two execution segments per task and no release jitter. The ratio between execution and suspension is fixed at $\beta = 0.5$, resulting in an equal split between execution and suspension time.

We consider systems with $m \in \{2, 4, 8\}$ processors. For each configuration, the number of tasks is fixed to $n = 2m$, and the total utilisation is varied from 10% to 90% in steps of 10%. Tasks are scheduled using deadline-monotonic priority assignment.

Figure 7.3 shows the runtime of the three analysis variants as utilisation increases. In all configurations, the state-centric approach (v1) has higher runtime than the node-based approaches (v2 and v3), with the performance gap becoming more pronounced as the number of processors increases. Averaged across utilisation levels, v2 is 3.14×, 7.30×, and 14.06× faster than v1 on 2, 4, and 8 processors, respectively, while v3 is 3.39×, 7.31×, and 15.78× faster. This corresponds to average runtime reductions of 67.41%, 86.03%, and 92.64% for

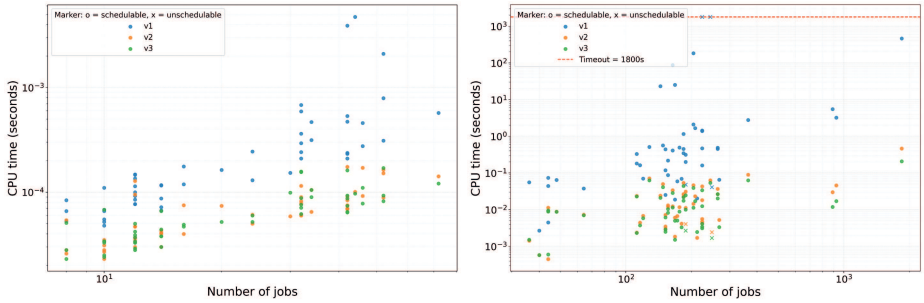


Figure 7.4: Runtime as a function of the number of jobs for systems with 4 (left) and 8 (right) tasks.

v2, and 69.46%, 86.02%, and 93.42% for v3. These averages are computed by first taking, at each utilisation level, the median runtime of each variant across all runs, then computing the speedup of v2 and v3 relative to v1 at that utilisation, and finally averaging these per-utilisation speedups across all utilisation levels.

A consistent ordering can be observed across the three plots: v1 has the highest runtime, v2 reduces runtime relative to v1, and v3 performs similar to v2. This pattern is visible for systems with 2, 4, and 8 processors.

Taken together, these results indicate that the dominant runtime improvement comes from the structural reorganisation introduced by nodes. The transition from the state-centric representation in v1 to the node-based representation in v2 yields the largest reduction in runtime, showing that changes to the structure of the analysis have a much greater effect than smaller implementation-level refinements. The additional optimisations in v3 provide a further reduction over v2 in some instances, but their impact is comparatively smaller.

Scaling with Number of Jobs We now investigate how the analysis scales with increasing numbers of jobs. To this end, we examine two segmented self-suspending task-system configurations: one comprising 4 tasks and another comprising 8 tasks. In both configurations, each task is composed of two execution segments, overall utilisation is fixed at 50%, there is no release jitter, and the execution-to-suspension ratio is set to $\beta = 0.5$. Tasks are scheduled according to the deadline-monotonic priority assignment policy.

Figure 7.4 shows the runtime of the three analysis variants as the number of jobs increases. Across both configurations, runtime increases with the number of jobs for all variants. The state-centric approach (v1) exhibits the highest runtime, while the node-based approaches (v2 and v3) consistently achieve lower runtime. This pattern holds for both the 4-task and 8-task configurations. For larger numbers of jobs, v1 approaches or reaches the timeout limit of 1800

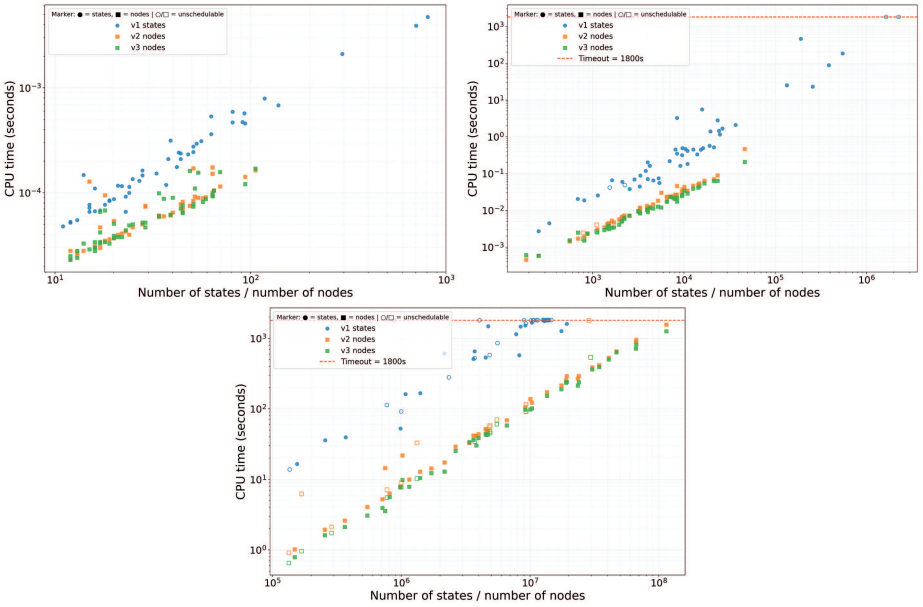


Figure 7.5: Runtime as a function of explored states (v1) and nodes (v2, v3) for systems with 4 (top left), 8 (top right), and 12 (bottom) tasks.

seconds in several cases, whereas v2 and v3 continue to complete the analysis within the time limit. The difference between v2 and v3 is less pronounced: v3 often performs slightly better, but not uniformly so. In many cases the two variants behave similarly, and in a few instances v3 is marginally worse. These results therefore suggest that the main scalability improvement comes from the node-based restructuring itself, while the additional refinements in v3 yield only a modest and less consistent gain.

Structural Efficiency: States vs Nodes We next evaluate the structural efficiency of the analysis by comparing the number of explored states and nodes against runtime.

We consider three configurations of segmented self-suspending task systems:

- 4 tasks on 2 processors,
- 8 tasks on 4 processors,
- 12 tasks on 6 processors.

In all configurations, the total utilisation is fixed at 50%, with $\beta = 0.5$ and no release jitter. Each task is divided into a number of segments equal to half the number of tasks in the system.

Figure 7.5 shows the relationship between runtime and the number of explored states or nodes. Across all configurations, runtime increases with the number of explored entities. The state-centric approach (v1) explores states directly, while the node-based approaches (v2 and v3) also process states but organise them within nodes. In this figure, the number of explored states in v1 is compared to the number of explored nodes in v2 and v3.

A clear separation is observed between v1 and the node-based variants. For comparable problem instances, v1 has the highest runtime, whereas v2 and v3 complete the analysis faster. This indicates that exploring fewer nodes leads directly to more efficient analysis. For the largest configurations, the difference becomes more significant: v1 often reaches the timeout limit before exploring as far as the node-based variants, while v2 and v3 continue the exploration and are able to analyse larger instances within the available time. Thus, the node-based representations are not only faster, but also allow the analysis to progress further before timing out.

Overall, these results show that grouping states by shared scheduling history improves both efficiency and scalability. The benefit is not merely a reduction in runtime, but also, as a corollary, an increased ability to handle larger explored representations within the same time bound.

DAG-Based Tasks

We next evaluate DAG-based task systems, which introduce additional structural complexity due to precedence relations between jobs.

Task graphs are generated following the approach in [18, 44], with a fork probability of 0.7 and up to three branches per fork. The generated graphs are further constrained by a maximum of 25 vertices and a maximum depth of 7. No release jitter is considered. Tasks are scheduled using the EDF scheduling policy.

We consider two sets of configurations:

- systems with 4 processors and a total utilisation of 45%, with 2 and 4 tasks,
- systems with 6 processors and a total utilisation of 30%, with 4 and 6 tasks.

Figures 7.6 and 7.7 show the runtime of the three analysis variants as a function of the number of explored entities for DAG-based task systems. An interesting pattern emerges here for the configurations considered, the number of explored nodes is equal to the number of explored states. This indicates that, in these DAG-based task sets, the node-based representation saves a single state per node.

We do not yet have a complete explanation for this behaviour. However, it suggests a potential property of the Schedule Abstraction Graph, namely that for

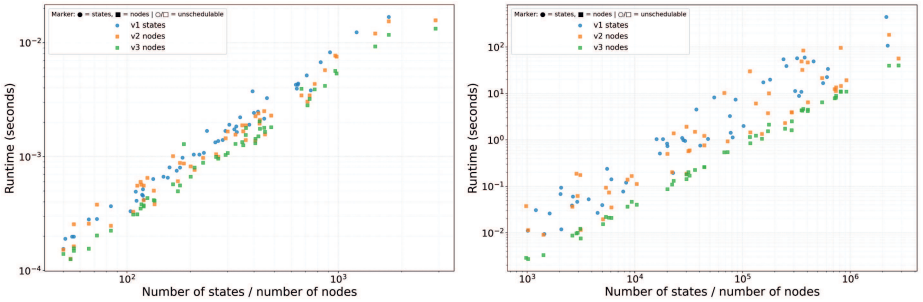


Figure 7.6: Runtime as a function of explored entities for DAG-based task systems on 4 processors with 2 (left) and 4 (right) tasks.

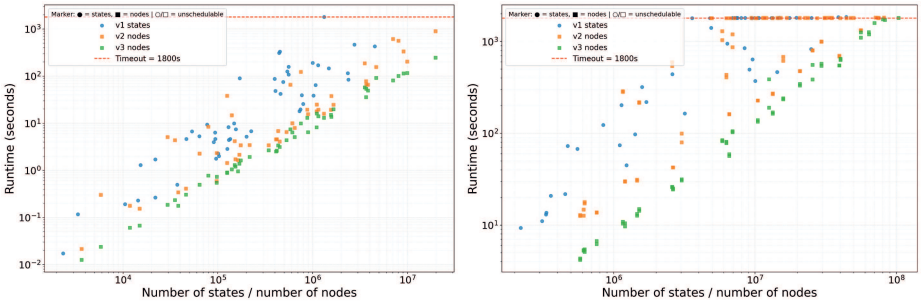


Figure 7.7: Runtime as a function of explored entities for DAG-based task systems on 6 processors with 4 (left) and 6 (right) tasks.

limited-pre-emptive systems without self-suspensions, states that share the same set of dispatched jobs may always merge. This behaviour has previously been observed for non-pre-emptive sequential tasks on single-core platforms, where such merging was expected, although not formally proven. The present results indicate that a similar property may also hold for multicore systems executing limited-pre-emptive DAG tasks.

Even though each node contains only a single state, v2 and v3 still achieve lower runtime than v1. This shows that, in the DAG case, the performance benefit of the node-based variants does not primarily come from reducing the number of explored entities, but rather from exploring them more efficiently. The reorganisation introduced in v2, and to a lesser extent the refinements in v3, reduce the cost of each exploration step even when the overall number of explored entities remains unchanged.

7.5.2 Case Study II: Time-Aware Shapers (TAS)

We next evaluate the impact of the different analysis variants on the TAS case study. The differences between the three variants therefore reflect the transition

from a state-based representation to a node-based representation, followed by further refinements to the underlying lookup structures.

The three variants differ as follows:

- v1 explores all TAS states independently, without structural aggregation,
- v2 introduces nodes to group structurally identical states and perform exploration at the level of transmission configurations,
- v3 preserves the node-based exploration of v2 while improving lookup efficiency through stronger hashing.

Experimental Methodology

The experimental setup follows the TAS GCL and packets generation described in Chapter 5. We consider systems with 2 communication flows per class, 8 gates, and a total utilisation of 80%.

We evaluate three timing configurations:

- no jitter and no transmission variation,
- no jitter with transmission variation,
- jitter with transmission variation.

For each configuration, we also vary the degree of gate overlap, considering no overlap, 50% overlap, 99% overlap, and 100% overlap.

All packet sets and GCL configurations are identical across the three analysis variants, ensuring a fair comparison.

Structural Efficiency: States vs Nodes

We first evaluate the structural efficiency of the node-based representation by comparing the number of states and nodes.

Figure 7.8 shows the ratio between the number of explored states and nodes for different gate overlap configurations.

In the case of no gate overlap, the ratio is very close to 1, meaning that each node typically contains a single state. This reflects the deterministic nature of the system in this configuration, where the absence of overlap leads to a fixed transmission order with limited variation.

As the degree of gate overlap increases, the ratio increases, indicating that more states are grouped within each node. This is because higher overlap allows for more variation in packet transmissions ordering between packets of different classes and thus to more variation on the timing of transmissions, leading to a

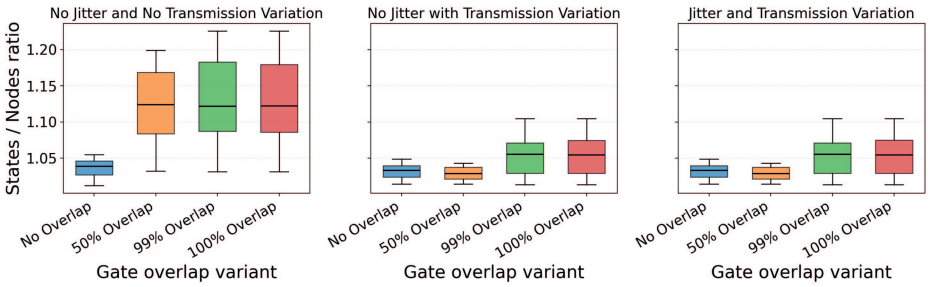


Figure 7.8: Ratio of states to nodes across different gate overlap variants and timing configurations.

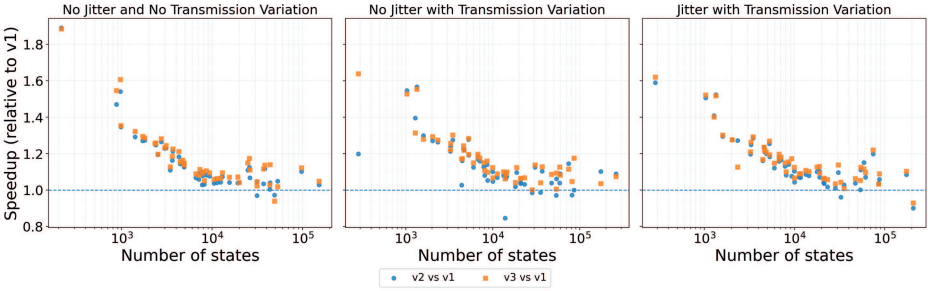


Figure 7.9: Speedup of v2 and v3 relative to v1 as a function of the number of states.

larger number of different states that share the same set dispatched packets. These states differ in timing but are grouped within the same node.

A different trend is observed when timing uncertainty is introduced through transmission variation and jitter. In these cases, the ratio of states to nodes decreases. This indicates that fewer distinct states are maintained per node.

This behaviour can be explained by the increased overlap between timing intervals. As timing variation increases, the temporal differences between states are more likely to overlap, making them compatible for merging. As a result, states that would otherwise be kept separate can now be combined, reducing the number of distinct states stored within each node.

These results show that node-based representations not only group structurally identical states, but also benefit from increased opportunities for state merging when timing uncertainty is higher.

Runtime Improvement

We next evaluate the runtime improvement achieved by the node-based representations relative to the state-centric baseline.

Figure 7.9 shows the speedup of v2 and v3 relative to v1.

Across all configurations, both v2 and v3 outperform v1 for the large majority of evaluated instances. Specifically, v2 achieves a speedup greater than 1 in 140 out of 150 instances (93.3%), while v3 does so in 148 out of 150 instances (98.7%). The improvement is more pronounced for smaller instances, where higher speedups are observed. As the number of states increases, the speedup decreases and approaches 1, indicating that although node-based representations reduce redundant structural exploration, the runtime of larger instances is increasingly dominated by the cost of state generation and timing analysis. A small number of instances exhibit a slowdown, with v2 performing worse than v1 in 10 cases and v3 in 2 cases. One possible explanation is the difference in how state merging is performed. In the state-centric representation, states are compared in the order in which they are generated, so merging decisions depend on the exploration order. In the node-based representation, states are organised according to their timing information within a node, and merging is performed with the first compatible state in that node. This can lead to different merge choices, which in some cases may produce less favourable state combinations and slightly higher runtime.

Overall, the results show that node-based representations provide a runtime benefit in almost all instances, while the specific ordering and selection of merge candidates can occasionally lead to deviations from this trend.

7.6 Discussion and Conclusion

The results across the EDD and TAS case studies show that the proposed node-based approach provides a consistent runtime benefit across different variants and use cases. Compared with the purely state-centric representation, it reduces redundant exploration more effectively, leading to lower analysis cost in most evaluated configurations.

In the EDD case study, for self-suspending tasks, the introduction of nodes leads to a clear reduction in runtime. Averaged across utilisation levels, the node-based variants achieve speedups of about 3× on 2 processors, 7× on 4 processors, and 14–16× on 8 processors relative to the state-centric variant. This improvement reflects two related effects. First, the node-based representation reorganises the exploration around structural configurations, which reduces the cost of individual exploration steps. Second, in the self-suspending task sets, many states share the same scheduling structure but differ only in timing. Grouping such states into a single node avoids repeatedly evaluating identical scheduling decisions. As the number of processors or jobs increases, the number of such structurally similar configurations grows, so the additional benefit from aggregation becomes more

pronounced.

In contrast, the DAG-based experiments show that the improvement does not depend solely on structural aggregation. In these cases, the number of nodes is equal to the number of states, so no aggregation takes place. Nevertheless, the node-based variants still achieve lower runtime, indicating that the reorganisation of exploration itself already improves efficiency, even when the overall size of the explored graph remains unchanged. Taken together, these results suggest that aggregation is an important amplifier of the runtime benefit in cases such as EDD, but not its sole source.

For the TAS case study, the observed improvements are more moderate. The results show that structural aggregation depends strongly on the degree of gate overlap and timing variability. When there is no overlap, the system behaves almost deterministically, and each node typically contains a single state. As overlap increases, more states share the same transmission structure and can be grouped within nodes. At the same time, increased timing uncertainty can lead to additional opportunities for state merging, as timing intervals overlap and states become compatible. However, the overall number of nodes remains close to the number of states, limiting the extent of structural compression.

The runtime results for TAS further show that node-based representations improve performance in most cases, but not uniformly. While v2 and v3 typically outperform the state-centric baseline, a small number of instances exhibit slight slowdowns. This behaviour highlights an important aspect of the approach that the efficiency of the analysis depends not only on the representation itself, but also on how states are merged within that representation. In particular, the order in which merge candidates are considered and the strategy used to select merge partners can influence both the number of states and the cost of the exploration.

Taken together, these results show that the introduction of nodes provides a systematic way to separate structural information from timing variation, which in turn enables more efficient exploration. However, they also show that this separation alone does not fully determine the performance of the analysis. Instead, it defines a design space in which different implementation choices can have a significant impact.

This design space opens several directions for further improvement. One key aspect is the merging strategy used within nodes. The current approach typically merges a state with the first compatible candidate, which is efficient but may not always lead to the most compact representation. Exploring alternative strategies, such as considering multiple merge candidates or selecting merge partners based on additional criteria, could lead to improved aggregation at the cost of additional computation. Understanding this trade-off between merge quality and merge cost is an important direction for future work.

Another aspect concerns the organisation of states within nodes. The ordering of states, the data structures used to store them, and the way compatibility checks are performed all influence the efficiency of merging and lookup operations. Different system models may benefit from different organisation strategies, suggesting that the node-based representation can be further specialised depending on the characteristics of the problem.

More generally, the results suggest that the key question is not only how to represent the state space, but also how to structure the exploration so that redundant work is avoided as early as possible. The introduction of nodes provides a first step in this direction by making structural similarity explicit. Building on this, further improvements can be achieved by refining how information is partitioned between nodes and states, and how operations such as expansion and merging are performed.

In this sense, the contribution of this work is twofold. First, it demonstrates that separating structural configuration from timing variation leads to consistent improvements in efficiency across different scheduling models. Second, it establishes a framework in which such improvements can be systematically developed and analysed. Rather than being a final optimisation, the proposed approach provides a foundation for designing more efficient analyses by making the structure of the problem explicit and enabling principled decisions about how to manage it. Overall, the results show that restructuring the Schedule Abstraction Graph around nodes leads to a more efficient exploration process and provides a clear path for further optimisation.



8

Conclusion and Future Work

TABLE OF CONTENTS

8.1 Answering the Research Questions
8.2 Key Insights
8.3 Final Remarks

This thesis investigated how the timing behaviour of realistic real-time systems can be analysed in a way that remains both informative and computationally feasible. The central challenge in such systems is not only to determine whether the required functional behaviour is produced, but also whether it is always produced within the required time bounds. Existing analysis techniques often approach this challenge from two opposite directions. On the one hand, analytical bound-based methods scale well, but may become overly pessimistic when applied to more expressive and realistic system models. On the other hand, exhaustive reachability-based methods can provide high precision, but typically suffer from poor scalability as the number of possible execution scenarios increases. This thesis therefore investigated abstraction-based timing analysis methods that seek to preserve scheduling-relevant behaviour while avoiding the full cost of exhaustive exploration.

Overall, the contributions of this thesis show that abstraction-based timing analysis can be applied to a broader range of real-time settings than had previously been considered. The work developed analyses for richer application-level task models involving precedence constraints and suspension behaviour, adapted the analysis to latency in Time-Sensitive Networking under different traffic shaping mechanisms, and improved the structural efficiency of the underlying exploration in order to support more scalable analysis. Across these settings, a common theme has been that effective timing analysis depends not only on exploring reachable behaviour, but on doing so through abstractions that preserve the information relevant for scheduling decisions while avoiding the explicit representation of unnecessary detail.

The methods developed in this thesis are realised within the Schedule Abstraction Graph (SAG) framework, which provides the abstraction-based foundation for representing and exploring scheduling behaviour. In this sense, SAG serves as the means through which the thesis develops new analyses and scalability improvements. The remainder of this chapter revisits the research questions introduced in Chapter 1, summarises how they have been answered, and reflects on the broader insights that this thesis provides for the analysis of real-time systems.

8.1 Answering the Research Questions

The significance of the contributions of this thesis is best understood in relation to the research questions formulated in Chapter 1, which guided the thesis as a whole. Collectively, these questions focused on how the timing behaviour of increasingly realistic real-time systems can be analysed while retaining a useful balance between precision and scalability.

RQ1: *To what extent can the timing behaviour of real-time systems be analysed accurately and in a scalable way across both the communication and computation domains?*

This thesis has shown how abstraction-based timing analysis performs against the state of the art in both the computation and communication domains. In the computation domain, Chapter 3 evaluates the accuracy of the analysis against existing methods, while in the communication domain Chapter 6 provides a corresponding evaluation for TSN systems. Building on these results, Chapter 7 shows that the accuracy achieved by the analyses developed in Chapter 3 and Chapter 5 can be preserved while improving scalability. Taken together, these results demonstrate that the timing behaviour of real-time systems can be analysed accurately and in a scalable way across both domains, provided that the abstraction is designed carefully (i.e., keeping a trade-off between accuracy of the information saved in abstract states and the number of concrete states that are abstracted by a single abstract state). Across all studied settings, the analysis is most effective when it retains the information that is essential to system behaviour, while avoiding distinctions and exploration that do not affect the result.

RQ2: How can limited-preemptive real-time tasks with segment-level precedence constraints following a DAG structure and with self-suspension behaviour be analysed?

This thesis has shown, in Chapter 3, how abstraction-based timing analysis can be applied to limited-pre-emptive real-time tasks with complex internal behaviour. Specifically, it shows how such tasks can be modelled as multiple execution segments linked by DAG-structured precedence constraints, while also incorporating self-suspension behaviour and the delays it causes. These suspensions may arise, for example, when part of a task is offloaded elsewhere for execution, or when a task must wait for a communication or transfer to complete before resuming. In this way, the thesis presents the first unified methodology for analysing task models that combine limited pre-emption, non-linear segment dependencies, and self-suspensions.

RQ3: How can abstraction-based timing analysis, in particular the Schedule Abstraction Graph, be generalised to the domain of Time-Sensitive Networking (TSN)?

RQ3.1: Which principles of Schedule Abstraction Graph-based timing analysis can be reused, with suitable modelling changes, when moving from computation to communication?

RQ3.2: Which aspects of Time-Sensitive Networking require the development of fundamentally new modelling and analysis within the Schedule Abstraction Graph framework?

This thesis has shown how abstraction-based timing analysis, and in particular the Schedule Abstraction Graph, can be generalised from computation scheduling to the communication domain of Time-Sensitive Networking. Since the framework was originally developed for the analysis of jobs on processors, this shift to TSN required more than a direct change of modelling context. Instead, this meant determining which core principles of the framework could be preserved through appropriate modelling adaptations, and which features of TSN required entirely new modelling and analysis approaches, as investigated in Chapters 4, 5 and 6.

Regarding RQ3.1, the thesis showed that several key principles of Schedule Abstraction Graph-based timing analysis can be reused when moving from computation to communication. In particular, it identified similarities between scheduling in the two domains that allow core abstraction ideas to be reformulated for packet transmissions and communication behaviour.

Regarding RQ3.2, the thesis also showed that TSN introduces important new features that require dedicated modelling within the framework. These include packet interactions across links and switches, queueing behaviour, and traffic shaping mechanisms that do not arise in the same form in application-level computation scheduling explored by the SAG framework. In particular, the thesis developed methods to analyse TSN shaping mechanisms such as TAS and CBS in Chapter 5 and 6, thereby extending the Schedule Abstraction Graph framework to behaviours that had not previously been captured. Taken together, these results show how the framework can be generalised to support systematic latency analysis in TSN systems.

8.2 Key Insights

This thesis has provided several broader insights into the analysis of real-time systems.

First, accurate timing analysis depends not only on the analysis method itself, but also on how the system is modelled. Across the different settings studied in

this thesis, the choice of representation proved to be central in determining both the accuracy of the analysis and its computational cost.

Second, the main challenge in abstraction-based analysis is to identify which information is actually relevant for decision-making. A recurring insight throughout this work is that effective analysis depends on deciding what must remain explicit, what can be aggregated, what can be reused, and what can safely be omitted without affecting correctness.

Third, methods such as Schedule Abstraction Graphs are most effective when they are carefully specialised to the structure of the problem under analysis. The results of this thesis showed that extending such methods to richer task models and to TSN is possible, but only when the abstraction is redesigned to reflect the timing semantics of the domain.

Fourth, scalability is closely tied to structure. Improving analysis is not only a matter of reducing the number of explored scenarios, but also of improving how behaviour is represented, organised, and reused during exploration.

Taken together, these insights show that advancing real-time systems analysis requires more than applying existing methods to larger problems. It requires designing representations and abstractions that match the true timing structure of the systems being analysed.

8.3 Final Remarks

This thesis has shown that abstraction-based analysis can contribute meaningfully to the study of realistic real-time systems when the abstraction is designed to reflect the timing structure of the problem under analysis. By extending Schedule Abstraction Graphs to richer application-level task models, adapting them to TSN, and improving their structural efficiency, this work has demonstrated that accurate reachability-based analysis can remain useful beyond simpler and more classical settings.

A broader conclusion of this thesis is that progress in real-time systems analysis depends not only on developing stronger analysis methods, but also on understanding how to model systems in a way that it preserves decision-relevant behaviour without introducing unnecessary complexity. In that sense, the main contribution of this work is not only to extend a particular method, but also to show how careful abstraction design can support the analysis of increasingly expressive systems.

This also points naturally to future work. An important next step is to further unify the TSN-related analyses developed in this thesis into a more integrated framework. Another promising direction is to transfer insights gained from TSN

back to application-level real-time systems, especially in settings where queueing, regulation, or other structured interactions play an important role. More generally, further work on how abstractions are constructed, refined, and reused may help strengthen the role of abstraction-based techniques in the analysis of complex real-time systems.

Overall, the results of this thesis help make real-time systems analysis more accurate and better able to capture the complexity of modern systems.

LIST OF ABBREVIATIONS

AFDX	Avionics Full-Duplex Switched Ethernet
BCET	Best-Case Execution Time
BCRT	Best-Case Response Time
CBS	Credit-Based Shaper
CPA	Compositional Performance Analysis
CPU	Central Processing Unit
CW-EDF	Critical Window Earliest Deadline First
DAG	Directed Acyclic Graph
DBM	Difference Bound Matrix
EDD	Event-Driven Delay-induced
EDF	Earliest Deadline First
FIFO	First-In First-Out
FP	Fixed Priority
FPGA	Field-Programmable Gate Array
GCL	Gate Control List
GPU	Graphics Processing Unit
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers
IPG	Inter-Packet Gap
JLFP	Job Level Fixed Priority
LiDAR	Light Detection And Ranging
Mbps	Megabits per second
NetCal	Network Calculus
ROS2	Robot Operating System 2
RTA	Response Time Analysis
SAG	Schedule Abstraction Graph
TAS	Time-Aware Shaper

TLFP	Task Level Fixed Priority
TPU	Tensor Processing Unit
TSN	Time-Sensitive Networking
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time
WCRTA-EI	Worst-Case Response-Time Analysis based on Eligible Intervals

LIST OF SYMBOLS

Part I

Common Part I Notations

Notation	Meaning
$\mathcal{T}$	Finite set of tasks in the system / task set under analysis
τ	A task
τ_i	Task i in the task set
$\mathcal{J}$	Set of all jobs under analysis
j	A job
$a_j^{\min}$	Earliest arrival time of job j
$a_j^{\max}$	Latest arrival time of job j
$C_j^{\min}$	Minimum / best-case execution time of job j
$C_j^{\max}$	Maximum / worst-case execution time of job j
d_j	Absolute deadline of job j
π_j	Priority assigned to job j
Γ	Set of all execution scenarios
$\Gamma(s)$	Subset of execution scenarios compatible with abstract state s
s	Abstract state in the Schedule Abstraction Graph
s_0	Initial abstract state
$\mathcal{D}$	Set of jobs already dispatched in an abstract state
L^{cur}	Current layer of abstract states being explored
L^{next}	Next layer of abstract states produced during exploration

Notation	Meaning
BCRT_j	Best-case response-time bound of job j
WCRT_j	Worst-case response-time bound of job j
$f_j^{\min}$	Earliest possible finish time of job j
$f_j^{\max}$	Latest possible finish time of job j
m	Number of processors in the platform

Chapter 2: The Schedule Abstraction Graph

Notation	Meaning
$j_{i,k}$	The k-th job generated by task τ_i
T_{τ_i}	Period of task τ_i
O_{τ_i}	Offset of task τ_i, i.e. the earliest activation time
J_{τ_i}	Arrival jitter of task τ_i
$C_{\tau_i}^{\min}$	Minimum execution time of jobs generated by task τ_i
$C_{\tau_i}^{\max}$	Maximum execution time of jobs generated by task τ_i
D_{τ_i}	Relative deadline of task τ_i
π_{τ_i}	Priority parameter of task τ_i
$a_{j_{i,k}}$	Concrete arrival time of job $j_{i,k}$
$a_{j_{i,k}}^{\min}$	Earliest possible arrival time of job $j_{i,k}$
$a_{j_{i,k}}^{\max}$	Latest possible arrival time of job $j_{i,k}$
$C_{j_{i,k}}$	Concrete execution time of job $j_{i,k}$
$C_{j_{i,k}}^{\min}$	Minimum execution time of job $j_{i,k}$
$C_{j_{i,k}}^{\max}$	Maximum execution time of job $j_{i,k}$
$d_{j_{i,k}}$	Absolute deadline of job $j_{i,k}$
γ_j	Timing assignment of job j, consisting of an arrival time and an execution time
a_j	Arrival time component of scenario element γ_j
C_j	Execution time component of scenario element γ_j
$A^{\min}$	Earliest time at which the processor may become available
$A^{\max}$	Earliest time at which the processor is guaranteed to be available
$G = (V, E)$	Schedule Abstraction Graph with vertex set V and edge set E
V	Set of abstract states in the graph
E	Set of transitions between abstract states

Notation	Meaning
d_j	Deadline of job j
$A_l^{\min}$	Earliest time at which at least l processors may be available
$A_l^{\max}$	Earliest time at which at least l processors are guaranteed to be available

Chapter 3: Analysis of Event-Driven Delay-Induced Tasks

Notation	Meaning
n	Number of tasks in $\mathcal{T}$
$\tau = (V_\tau, E_\tau)$	EDD task represented as a directed acyclic graph
V_τ	Set of vertices of task τ, representing execution segments
E_τ	Set of directed edges of task τ, representing precedence constraints
v	A vertex / execution segment of a task
(v, v')	Directed edge from segment v to segment v'
$C_{\tau,v}^{\min}$	Best-case execution time of segment v of task τ
$C_{\tau,v}^{\max}$	Worst-case execution time of segment v of task τ
$sus_{\tau,v,v'}^{\min}$	Minimum suspension / delay from segment v to segment v'
$sus_{\tau,v,v'}^{\max}$	Maximum suspension / delay from segment v to segment v'
$pred_j$	Set of immediate predecessor jobs of j
$succ_j$	Set of successor jobs of j
$sus^{\min}(j, j')$	Minimum suspension time between a predecessor job and j
$sus^{\max}(j, j')$	Maximum suspension time between a predecessor job and j
$sus(j, j')$	Concrete suspension time between a predecessor job and j
ρ_j	Concrete ready time of job j
a_j	Concrete arrival time of job j
f_j	Concrete finishing time of job j
γ_j	Concrete parameter assignment for job j
χ	Set of jobs certainly executing in abstract state s
$A_i^{\min}$	Earliest time at which at least i processors may be available
$A_i^{\max}$	Earliest time from which at least i processors are guaranteed to be available
σ_j	Concrete start time of job j
$\mathcal{R}$	Set of potentially ready jobs in an abstract state

Notation	Meaning
ρ_j^{min}	Lower bound on the ready time of job j
ρ_j^{max}	Upper bound on the ready time of job j
$\bar{\rho}_j^{hp}$	Upper bound on the time at which any higher-priority job certainly becomes ready
$\bar{\rho}_j^{any}$	Upper bound on the time at which any potentially ready job certainly becomes ready
σ_j^{min}	Lower bound on the start time of job j
σ_j^{max}	Upper bound on the start time of job j
$\tilde{s}$	Successor abstract state generated after dispatching a job
$\tilde{\mathcal{D}}$	Dispatched-job set in successor abstract state $\tilde{s}$
$\tilde{\chi}$	Certainly-running set in successor abstract state $\tilde{s}$
$\tilde{F}$	Set of stored finish-time intervals in successor abstract state $\tilde{s}$
$\tilde{A}$	Processor-availability intervals in successor abstract state $\tilde{s}$
$\tilde{A}_i^{min}$	Lower bound of the i-th availability interval in $\tilde{s}$
$\tilde{A}_i^{max}$	Upper bound of the i-th availability interval in $\tilde{s}$
s_1, s_2	Abstract states considered in the merging phase
s'	Merged abstract state
L_ℓ	Set of abstract states at layer ℓ of the graph
ℓ	Layer depth in the Schedule Abstraction Graph
p	Number of certainly running predecessors of a dispatched job
U'	Target total utilization used in the experiments
U'_τ	Artificial utilization assigned to task τ during task-set generation
T_τ	Period of task τ in the experiments
C_τ	Total worst-case execution time of task τ in the experiments
$susp_\tau$	Total worst-case suspension time of task τ in the experiments
β	Ratio controlling execution time versus suspension time in the experiments
S	Number of execution segments of a generated self-suspending task

Part II

Common Part II Notations

Notation	Meaning
$\mathcal{F}$	Set of communication flows
F	A communication flow
F_i	Flow i in the flow set
$\mathcal{P}$	Set of packets under analysis
$p_{i,j}$	A packet
$d_{p_{i,j}}$	Absolute deadline of packet $p_{i,j}$
$\pi_{p_{i,j}}$	Priority of packet $p_{i,j}$
R^{tx}	Transmission rate of the considered egress port
C_{IPG}	Inter-packet gap duration / corresponding transmission time
γ	Transmission scenario
Γ	Set of all transmission scenarios
$f_{p_{i,j}}$	Finish time of packet $p_{i,j}$
$\text{BCRT}p_{i,j}$	Best-case latency / response time of packet $p_{i,j}$
$\text{WCRT}p_{i,j}$	Worst-case latency / response time of packet $p_{i,j}$
$\mathcal{C}$	Set of traffic classes
$\mathcal{C}_a$	A traffic class
$\pi_{\mathcal{C}_a}$	Priority of traffic class $\mathcal{C}_a$
H	Hyperperiod of the periodic traffic system / observation window basis
$A^{\min}$	Earliest time the egress port may become available
$A^{\max}$	Earliest time the egress port is guaranteed to be available
L^{cur}	Current layer of abstract states during exploration
L^{next}	Next layer of abstract states during exploration
$tr_{\mathcal{C}_a}^{\max}$	Upper bound on the earliest time at which a not-yet-transmitted packet of a class is certainly in its FIFO queue

Chapter 4: Time-Sensitive Networking

Notation	Meaning
T_F	Period of flow F

Notation	Meaning
O_F	Offset of flow F, i.e. earliest release time of its first packet
J_F	Arrival jitter of flow F
$l_F^{\min}$	Minimum packet length of flow F
$l_F^{\max}$	Maximum packet length of flow F
D_F	Relative end-to-end deadline of flow F
π_F	Priority of flow F
$a_{p_{i,j}}^{\min}$	Earliest arrival time of packet $p_{i,j}$ at the considered switch
$a_{p_{i,j}}^{\max}$	Latest arrival time of packet $p_{i,j}$ at the considered switch
$C_{p_{i,j}}^{\min}$	Minimum transmission time of packet $p_{i,j}$
$C_{p_{i,j}}^{\max}$	Maximum transmission time of packet $p_{i,j}$
$l_F^{\min}$	Minimum packet length generated by flow F
$l_F^{\max}$	Maximum packet length generated by flow F
$l_{p_{i,j}}^{\min}$	Minimum length of packet $p_{i,j}$
$l_{p_{i,j}}^{\max}$	Maximum length of packet $p_{i,j}$
$C_{p_{i,j}}$	Concrete transmission time of packet $p_{i,j}$
m	Number of ingress ports in the switch architecture figure
n	Number of egress ports in the switch architecture figure
o	Number of transmission queues per egress port / traffic classes

Chapter 5: Analysis of Time-Aware Shapers on a TSN Switch

Notation	Meaning
$\mathcal{F}$	Finite set of communication flows transmitted through the considered egress port
T_i	Period of flow F_i
D_i	End-to-end deadline of flow F_i
$a_{p_{i,j}}^{\min}$	Earliest arrival time of packet $p_{i,j}$
$a_{p_{i,j}}^{\max}$	Latest arrival time of packet $p_{i,j}$
$C_{p_{i,j}}^{\min}$	Minimum transmission time of packet $p_{i,j}$
$C_{p_{i,j}}^{\max}$	Maximum transmission time of packet $p_{i,j}$
$a_{p_{i,j}}$	Concrete arrival time of packet $p_{i,j}$ in a transmission scenario
$C_{p_{i,j}}$	Concrete transmission duration of packet $p_{i,j}$
$a_{p_{i,j}}^{\min}$	Minimum arrival time bound of packet $p_{i,j}$

Notation	Meaning
$a^{\max}_{p_{i,j}}$	Maximum arrival time bound of packet $p_{i,j}$
$C^{\min}_{p_{i,j}}$	Minimum transmission-time bound of packet $p_{i,j}$
$C^{\max}_{p_{i,j}}$	Maximum transmission-time bound of packet $p_{i,j}$
$R_{p_{i,j}}$	Latency / response time of packet $p_{i,j}$
$\mathcal{P}_{\mathcal{C}_a}$	Set of packets belonging to traffic class $\mathcal{C}_a$
$\mathcal{G}_{\mathcal{C}_a}$	Set of time intervals during which the gate of class $\mathcal{C}_a$ is open
$go_l^{\mathcal{C}_a}$	Opening time of the l-th gate-open interval of class $\mathcal{C}_a$
$gc_l^{\mathcal{C}_a}$	Closing time of the l-th gate-open interval of class $\mathcal{C}_a$
$n_{\mathcal{C}_a}$	Number of gate-open intervals of class $\mathcal{C}_a$ within the observation window
$\mathcal{D}$	Set of packets already transmitted in an abstract state
s	Abstract state in the Schedule Abstraction Graph
s_0	Initial abstract state $(\emptyset, [0, 0])$
$\mathcal{E}_{p_{i,j}}$	Set of eligible transmission intervals of packet $p_{i,j}$
$\mathcal{E}^l_{p_{i,j}}$	The l-th interval in the set $\mathcal{E}_{p_{i,j}}$
$EFT_{p_{i,j}}$	Earliest finish time of packet $p_{i,j}$
$LFT_{p_{i,j}}$	Latest finish time of packet $p_{i,j}$
$top_{\mathcal{C}_a}$	Set of packets that may be at the top of the FIFO queue of class $\mathcal{C}_a$
$\hat{t}_{\mathcal{C}_a}(s)$	Earliest time when the egress is certainly available and there is certainly a packet at the top of class $\mathcal{C}_a$'s queue
$\mathcal{G}^{\text{pos}}_{p_{i,j}}$	Set of time intervals where packet $p_{i,j}$ may not be able to start transmission because Rule R2 and/or R3 may be violated
$gc^{\text{last}}_{p_{i,j}}$	Last time instant before $\hat{t}_{\mathcal{C}_a}(s)$ at which $p_{i,j}$ may not be able to start transmission
$go^{\text{next}}_{p_{i,j}}$	First time instant after $gc^{\text{last}}_{p_{i,j}}$ at which $p_{i,j}$ may start transmission
$t^{\text{ready}}_{p_{i,j}}$	Earliest time when packet $p_{i,j}$ is certainly ready and the egress is certainly available
$t^{\text{lb}}_{p_{i,j}}$	Lower bound on the start time of packet $p_{i,j}$
$t^{\text{ub}}_{p_{i,j}}$	Upper bound on the start time of packet $p_{i,j}$
$\mathcal{G}^{\text{cert}}_{p_{i,j}}$	Set of time intervals in which packet $p_{i,j}$ can certainly not start transmission
$\mathcal{I}^{\text{hp}}_{\mathcal{C}_b}$	Set of time intervals in which there is certainly a ready packet of higher-priority class $\mathcal{C}_b$

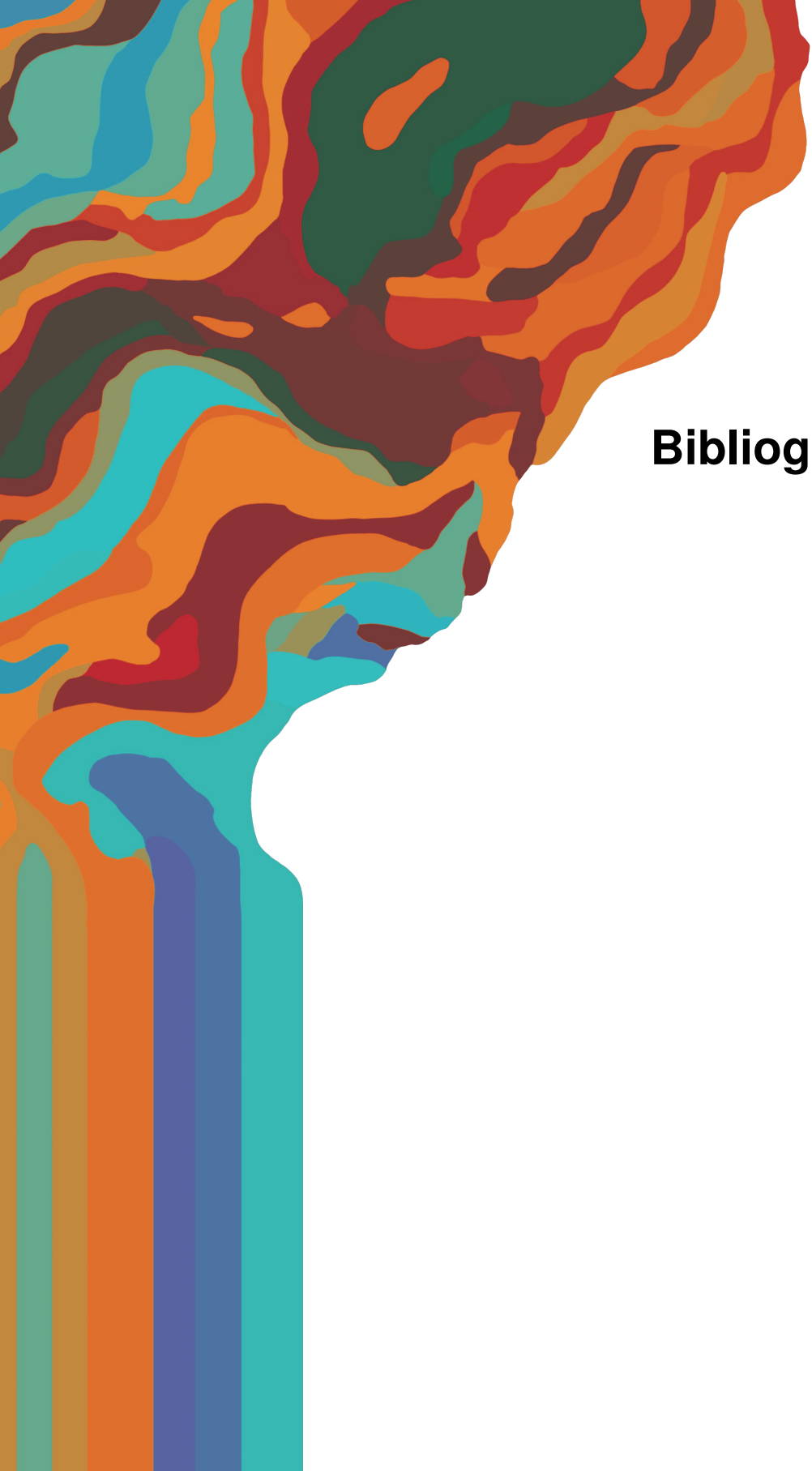
Notation	Meaning
$tr_{\mathcal{C}_b}^{\max}$	Upper bound on the earliest time at which a not-yet-transmitted packet of class $\mathcal{C}_b$ is certainly in its FIFO queue
$top_{\mathcal{C}_b}$	Set of packets that may be at the top of the FIFO queue of class $\mathcal{C}_b$
$\mathcal{E}_{p_{i,j}}^{k,\text{start}}$	Start of the considered eligible transmission interval of $p_{i,j}$
$\mathcal{E}_{p_{i,j}}^{k,\text{end}}$	End of the considered eligible transmission interval of $p_{i,j}$
$gc_{p_{i,j}}^{\text{next}}$	First closing time of $p_{i,j}$'s gate after $p_{i,j}$ starts transmission
U_a	Bandwidth utilization reserved for traffic class $\mathcal{C}_a$
T_a	Period of the gate schedule of traffic class $\mathcal{C}_a$
O_a	Total duration for which the gate of class $\mathcal{C}_a$ is open within one period T_a
f	Number of communication flows per traffic class in the experiments

Chapter 6: Analysis of Credit-Based Shapers

Notation	Meaning
C_a	Traffic class a
π_{C_a}	Priority of traffic class C_a
$Q_{p,a}$	Queue of class C_a at egress port p
$\alpha_{p,a}^{+}$	Idle slope of class C_a at egress port p
$\alpha_{p,a}^{-}$	Send slope of class C_a at egress port p
α_{a}^{+}	Idle slope of class C_a on the considered egress port
α_{a}^{-}	Send slope of class C_a on the considered egress port
$\mathcal{F}$	Set of communication flows, used as shorthand for $\mathcal{F}$
$\mathcal{P}$	Set of all packets under analysis
$p_{i,j}$	The j-th packet generated by flow F_i
$a_{p_{i,j}}$	Concrete arrival time of a packet
$a_{p_{i,j}}^{\min}$	Earliest possible arrival time of a packet
$a_{p_{i,j}}^{\max}$	Latest possible arrival time of a packet
$d_{p_{i,j}}$	Absolute deadline of a packet
$\ell_{p_{i,j}}^{\min}$	Minimum packet length
$\ell_{p_{i,j}}^{\max}$	Maximum packet length
$C_{p_{i,j}}^{\min}$	Minimum transmission duration of a packet
$\hat{C}_{p_{i,j}}^{\min}$	Minimum transmission duration of a packet including IPG

Notation	Meaning
$\hat{C}^{\max}_{p_{i,j}}$	Maximum transmission duration of a packet including IPG
$f_{p_{i,j}}(\gamma)$	Finish time of packet $p_{i,j}$ in transmission scenario γ
s	Abstract state m in the Schedule Abstraction Graph
s'	Newly generated successor abstract state
$\mathcal{T}$	Set of packets already transmitted in an abstract state
ϵ_{C_a}	Set of possible finish-time/credit pairs for the most recently transmitted packet of class C_a
(t, c)	A time–credit pair
ϵ_{C_a}	Shorthand for ϵ_{C_a}
ϵ_1	Illustrative instance of a class credit-time region used in an example
ι_1	Illustrative instance of the projected non-negative-credit time set used in an example
top_{C_a}	Set of packets that may be at the top of the FIFO queue of class C_a
$tr^{\min}_{C_a}$	Lower bound on the earliest time at which a not-yet-transmitted packet of class C_a may have entered its FIFO queue
$zc_{C_a}(t, c)$	Function returning the earliest time at or after t when the credit becomes non-negative, starting from credit c at time t
ι_{C_a}	Set of time instants at which the credit of class C_a becomes non-negative
$t^{+,\min}_{C_a}$	Earliest time at which the credit of class C_a may become non-negative
$t^{+,\max}_{C_a}$	Latest time at which the credit of class C_a becomes non-negative
$t^{\mathrm{lb}}_{p_{i,j}}$	Lower bound on the start time of transmission of a packet or class
t^{eet}	Upper bound on the time by which some class is certainly eligible to transmit
$t^{\mathsf{high}}_{C_a}$	Upper bound on the time by which a higher-priority class than C_a becomes eligible
$t^{+,\mathsf{max}}_{C_b}$	Latest time at which the credit of class C_b becomes non-negative
$\sigma^{\min}_{p_{i,j}}$	Lower bound on the start time of an eligible packet
$\sigma^{\max}_{p_{i,j}}$	Upper bound on the start time of an eligible packet
$f^{\min}_{p_{i,j}}$	Earliest finish time bound of an eligible packet
$f^{\max}_{p_{i,j}}$	Latest finish time bound of an eligible packet

Notation	Meaning
$g_{C_a}(t, c, a_{p_{i,j}}, \sigma)$	Credit accumulation function for class C_a from state (t, c) until start time σ, assuming packet arrival at time $a_{p_{i,j}}$
$h_{C_a}(t, c, C_{p_{i,j}})$	Credit update function for class C_a during transmission of packet $p_{i,j}$
σ	Concrete start time of transmission
U_a	Idle-slope share / bandwidth share assigned to traffic class $\mathcal{C}_a$ in the experiments



Bibliography

[1] Muhammad Adnan et al. “Model for worst case delay analysis of an AFDX network using timed automata”. In: *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. 2010, pp. 1–4. DOI: 10.1109/ETFA.2010.5641124.

[2] *Apollo Cyber RT*. URL: https://cyber-rt.readthedocs.io/en/latest/.

[3] Federico Aromolo et al. “Event-driven delay-induced tasks: Model, analysis, and applications”. In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2021, pp. 53–65.

[4] *Artefact of ”Compositional Verification of Embedded Real-Time Systems”*. Version 1.0. 2024. URL: https://gitlab.math.univ-paris-diderot.fr/foughali/fhz%5C_aeic-jsa.

[5] Neil Audsley et al. “Applying new scheduling theory to static priority preemptive scheduling”. In: *Software Engineering Journal* 8.5 (1993), pp. 284–292.

[6] Theodore P Baker and Michele Cirinei. “Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks”. In: *International Conference On Principles Of DIstributed Systems*. Springer. 2007, pp. 62–75.

[7] S.K. Baruah, A.K. Mok, and L.E. Rosier. “Preemptively scheduling hard-real-time sporadic tasks on one processor”. In: *[1990] Proceedings 11th Real-Time Systems Symposium*. 1990, pp. 182–190. DOI: 10.1109/REAL.1990.128746.

[8] Lucia Lo Bello et al. “Schedulability analysis of Time-Sensitive Networks with scheduled traffic and preemption support”. In: *Journal of Parallel and Distributed Computing* 144 (2020), pp. 153–171.

[9] Enrico Bini and Giorgio C. Buttazzo. “Measuring the Performance of Schedulability Tests”. In: *Real-Time Systems* 30.1-2 (2005), pp. 129–154. DOI: 10.1007/s11241-005-0507-9. URL: https://doi.org/10.1007/s11241-005-0507-9.

[10] Unmesh D Bordoloi et al. “Schedulability analysis of Ethernet AVB switches”. In: *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2014, pp. 1–10.

[11] Marc Boyer et al. “Integrating end-system frame scheduling for more accurate AFDX timing analysis”. In: *ERTS 2014 proceedings*. Toulouse, France, Feb. 2014. URL: https://hal.science/hal-02272276.

[12] Daniel Bujosa, Mohammad Ashjaei, and Saad Mubeen. “CONAN-TSN: An Integrated Toolchain for CONfiguration and ANalysis of TSN Networks”. In: *2025 IEEE 30th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2025, pp. 1–8. DOI: 10.1109/ETFA65518.2025.11205713.

[13] Daniel Bujosa et al. “An Improved Worst-Case Response Time Analysis for AVB Traffic in Time-Sensitive Networks”. In: *2024 IEEE Real-Time Systems Symposium (RTSS)*. 2024, pp. 135–147. DOI: 10.1109/RTSS62706.2024.00021.

[14] Artem Burmyakov, Enrico Bini, and Chang-Gun Lee. “Towards a tractable exact test for global multiprocessor fixed priority scheduling”. In: *IEEE Transactions on Computers* 71.11 (2022), pp. 2955–2967.

[15] Artem Burmyakov, Enrico Bini, and Eduardo Tovar. “An exact schedulability test for global FP using state space pruning”. In: *Proceedings of the 23rd international conference on real time and networks systems*. 2015, pp. 225–234.

[16] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. “Limited preemptive scheduling for real-time systems. a survey”. In: *IEEE transactions on Industrial Informatics* 9.1 (2012), pp. 3–15.

[17] Jingyue Cao et al. “Independent yet tight WCRT analysis for individual priority classes in ethernet AVB”. In: *Proc. 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 55–64.

[18] Daniel Casini et al. “Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions”. In: *RTSS*. IEEE Computer Society, 2018, pp. 421–433.

[19] Jian-Jia Chen et al. “Many Suspensions, Many Problems: A Review of Self-Suspending Tasks in Real-Time Systems”. In: *Real-Time Systems* (2018). https://link.springer.com/article/10.1007/s11241-018-9316-9.

[20] Robert I. Davis et al. “Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised”. In: *Real-Time Systems* 35.3 (2007), pp. 239–272. DOI: 10.1007/s11241-007-9012-7.

[21] Joan Adrià Ruiz De Azua and Marc Boyer. “Complete modelling of AVB in network calculus framework”. In: *Proc. 22nd International Conference on Real-Time Networks and Systems*. 2014, pp. 55–64.

[22] Jonas Diemer, Jonas Rox, and Rolf Ernst. “Modeling of Ethernet AVB networks for worst-case timing analysis”. In: *IFAC Proceedings Volumes* 45.2 (2012), pp. 848–853.

[23] Jonas Diemer, Daniel Thiele, and Rolf Ernst. “Formal worst-case timing analysis of Ethernet topologies with strict-priority and AVB switching”. In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2012, pp. 1–10.

[24] Paul Emberson, Roger Stafford, and Robert I Davis. “Techniques for the synthesis of multiprocessor tasksets”. In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*. 2010, pp. 6–11.

[25] Paul Emberson, Roger Stafford, and Robert I Davis. “Techniques for the synthesis of multiprocessor tasksets”. In: *WATERS*. 2010, pp. 6–11.

[26] Mohammed Foughali, Pierre-Emmanuel Hladik, and Alexander Zuepke. “Compositional verification of embedded real-time systems”. In: *Journal of Systems Architecture* 142 (2023), p. 102928.

[27] Laurent George, Nicolas Rivierre, and Marco Spuri. “Preemptive and non-preemptive real-time uniprocessor scheduling”. PhD thesis. Inria, 1996.

[28] Laurent Georges, Paul Mühlethaler, and Nicolas Rivierre. “A few results on non-preemptive real time scheduling”. PhD thesis. INRIA, 2000.

[29] Pourya Gohari, Jeroen Voeten, and Mitra Nasri. “Reachability-Based Response-Time Analysis of Preemptive Tasks Under Global Scheduling”. In: *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*. Ed. by Rodolfo Pellizzoni. Vol. 298. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 3:1–3:24. ISBN: 978-3-95977-324-9. DOI: 10.4230/LIPIcs.ECRTS.2024.3. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2024.3.

[30] Pourya Gohari et al. “Response-time Analysis of Conditional DAG Tasks under Limited-Preemptive Global Scheduling”. In: *Proceedings of the Real-Time Networks and Systems Conference (RTNS)*. 2025.

[31] Nan Guan et al. “Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking”. In: *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer. 2007, pp. 263–272.

[32] Arne Hamann et al. “Waters industrial challenge 2017”. In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2017.

[33] Jonas Hansen et al. “Exact Schedulability Analysis for Limited-Preemptive Parallel Applications Using Timed Automata in UPPAAL”. In: *2025 Design, Automation Test in Europe Conference (DATE)*. 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992936.

[34] IEEE. *IEEE 802.1AS-2025: IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications*. https://https://standards.ieee.org/ieee/802.1AS/3956/. 2025.

[35] IEEE. *IEEE 802.1Qav-2009: IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams*. https://standards.ieee.org/ieee/802.1Qav/4401/. Accessed: 2026-04-12. 2009.

[36] IEEE. *IEEE 802.1Qbv-2015: IEEE Standard for Local and metropolitan area networks—Bridges and Bridged Networks—Amendment 25: Enhancements for Scheduled Traffic*. https://https://standards.ieee.org/ieee/802.1Qbv/6068/. IEEE 802.1Qbv-2015. 2015.

[37] “IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks”. In: *IEEE Std 802.1Q-2022 (Revision of IEEE Std 802.1Q-2018)* (2022), pp. 1–2163.

[38] Kevin Jeffay, Donald F Stanat, and Charles U Martel. “On non-preemptive scheduling of periodic and sporadic tasks”. In: *IEEE real-time systems symposium*. US: IEEE. 1991, pp. 129–139.

[39] Xu Jiang et al. “Real-time scheduling and analysis of processing chains on multi-threaded executor in ros 2”. In: *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2022, pp. 27–39.

[40] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. “Real world automotive benchmarks for free”. In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2015.

[41] Xiaoting Li and Laurent George. “Deterministic delay analysis of AVB switched Ethernet networks using an extended trajectory approach”. In: *Real-Time Systems* 53.1 (2017), pp. 121–186.

[42] Xiaoting Li, Jean-Luc Scharbarg, and Christian Fraboul. “Partially Synchronizing Periodic Flows with Offsets Improves Worst-Case End-to-End Delay Analysis of Switched Ethernet”. In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 228–242. ISBN: 978-3-642-16558-0.

[43] Xiaoting Li et al. “Existing offset assignments are near optimal for an industrial AFDX network”. In: *SIGBED Rev.* 8.4 (Dec. 2011), pp. 49–54. DOI: 10.1145/2095256.2095263. URL: https://doi.org/10.1145/2095256.2095263.

[44] Alessandra Melani et al. “Response-time analysis of conditional DAG tasks in multiprocessor systems”. In: *2015 27th Euromicro Conference on Real-Time Systems*. IEEE. 2015, pp. 211–221.

[45] Mitra Nasri and Björn B. Brandenburg. “Offline Equivalence: A Non-preemptive Scheduling Technique for Resource-Constrained Embedded Real-Time Systems (Outstanding Paper)”. In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2017, pp. 75–86. DOI: 10.1109/RTAS.2017.34.

[46] Mitra Nasri and Bjorn B. Brandenburg. “An Exact and Sustainable Analysis of Non-preemptive Scheduling”. In: *2017 IEEE Real-Time Systems Symposium (RTSS)*. 2017, pp. 12–23. DOI: 10.1109/RTSS.2017.00009.

[47] Mitra Nasri and Gerhard Fohler. “Non-work-conserving non-preemptive scheduling: motivations, challenges, and potential solutions”. In: *ECRTS*. 2016, pp. 165–175.

[48] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. “A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling”. In: *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Ed. by Sebastian Altmeyer. Vol. 106. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 9:1–9:23. ISBN: 978-3-95977-075-0. DOI: 10.4230/LIPIcs.ECRTS.2018.9. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2018.9.

[49] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. “Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling”. In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Ed. by Sophie Quinton. Vol. 133. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 21:1–21:23. ISBN: 978-3-95977-110-8. DOI: 10.4230/LIPIcs.ECRTS.2019.21. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2019.21.

[50] Geoffrey Nelissen, Joan Marcè i Igual, and Mitra Nasri. “Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks”. In: *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Ed. by Martina Maggio. Vol. 231. Leibniz International Proceedings in Informatics (LIPIcs).

Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 12:1–12:22. ISBN: 978-3-95977-239-6. DOI: 10.4230/LIPIcs.ECRTS.2022.12. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2022.12.

[51] Suhail Nogd et al. “Response-Time Analysis for Non-Preemptive Global Scheduling with FIFO Spin Locks”. In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. 2020, pp. 115–127. DOI: 10.1109/RTSS49844.2020.00021.

[52] Rene Queck. “Analysis of Ethernet AVB for automotive networks using Network Calculus”. In: *2012 IEEE International Conference on Vehicular Electronics and Safety (ICVES 2012)*. IEEE. 2012, pp. 61–67.

[53] S. Ranjha, P. Gohari, G. Nelissen, et al. “Partial-Order Reduction in Reachability-Based Response-Time Analyses of Limited-Preemptive DAG Tasks”. In: *Real-Time Systems* 59 (2023), pp. 201–255. DOI: 10.1007/s11241-023-09398-x. URL: https://doi.org/10.1007/s11241-023-09398-x.

[54] Sayra Ranjha, Geoffrey Nelissen, and Mitra Nasri. “Partial-Order Reduction for Schedule-Abstraction-based Response-Time Analyses of Non-Preemptive Tasks”. In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2022, pp. 121–132. DOI: 10.1109/RTAS54340.2022.00018.

[55] *SAG analysis github*. Version 2.2.0. 2024. URL: https://github.com/gnelissen/np-schedulability-analysis.

[56] Aakash Soni et al. “Integrating Offset in Worst Case Delay Analysis of Switched Ethernet Network With Deficit Round Robbin”. In: *Proceedings of ETFA 2018*. Turin, Italy, Sept. 2018, pp. 353–359. URL: https://hal.science/hal-02494127.

[57] Srinidhi Srinivasan et al. “Analysis of TSN Credit-Based Shapers Using the Schedule Abstraction Graph Methodology”. Under review.

[58] Srinidhi Srinivasan, Mario Günzel, and Geoffrey Nelissen. “Response-Time Analysis for Limited-Preemptive Self-Suspending and Event-Driven Delay-Induced Tasks”. In: *2024 IEEE Real-Time Systems Symposium (RTSS)*. 2024, pp. 29–42. DOI: 10.1109/RTSS62706.2024.00013.

[59] Srinidhi Srinivasan, Geoffrey Nelissen, and Reinder Bril. “Work-in-Progress: Analysis of TSN Time-Aware Shapers using Schedule Abstraction Graphs”. In: Dec. 2021, pp. 508–511. DOI: 10.1109/RTSS52674.2021.00052.

[60] Srinidhi Srinivasan et al. “Analysis of TSN Time-Aware Shapers Using Schedule Abstraction Graphs”. In: *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*. Vol. 298. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 16:1–16:24. DOI: 10.4230/LIPIcs.ECRTS.2024.16.

[61] Youcheng Sun and Giuseppe Lipari. “A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling”. In: *Real-Time Systems* 52.3 (2016), pp. 323–355.

[62] Daniel Thiele, Rolf Ernst, and Jonas Diemer. “Formal worst-case timing analysis of Ethernet TSN’s time-aware and peristaltic shapers”. In: *2015 IEEE Vehicular Networking Conference (VNC)*. IEEE. 2015, pp. 251–258.

[63] Ken W Tindell, Alan Burns, and Andy J. Wellings. “An extendible approach for analyzing fixed priority hard real-time tasks”. In: *Real-Time Systems* 6.2 (1994), pp. 133–151.

[64] Micaela Verucchi, Ignacio Sañudo Olmedo, and Marko Bertogna. “A survey on real-time DAG scheduling, revisiting the Global-Partitioned Infinity War”. In: *Real-Time Systems* 59.3 (2023), pp. 479–530.

[65] Beyazit Yalcinkaya, Mitra Nasri, and Björn B. Brandenburg. “An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks”. In: *DATE*. IEEE, 2019, pp. 1228–1233.

[66] Luxi Zhao et al. “Latency analysis of multiple classes of AVB traffic in TSN with standard credit behavior using network calculus”. In: *IEEE Transactions on Industrial Electronics* 68.10 (2020), pp. 10291–10302.

[67] Yimi Zhao et al. “Work-in-Progress: Generating Counter-Examples to Schedulability Using the Schedule Abstraction”. In: *2023 IEEE Real-Time Systems Symposium (RTSS)*. 2023, pp. 459–462. DOI: 10.1109/RTSS59052.2023.00052.

IPA DISSERTATION SERIES

Titles in the IPA Dissertation Series since 2023

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification*. Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification*. Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software*. Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI*. Faculty of Electrical Engineering,

Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond*. Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic*. Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction*. Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software*. Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting*. Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement*. Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies*. Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi*. Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

P. Karkhanis. *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways*. Faculty of Mathematics and Computer Science, TU/e. 2024-15

N.W. Cassee. *Sentiment in Software Engineering*. Faculty of Mathematics and Computer Science, TU/e. 2024-16

H. van Antwerpen. *Declarative Name Binding for Type System Specifications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01

I.N. Mulder. *Proof Automation for Fine-Grained Concurrent Separation Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2025-02

T.S. Badings. *Robust Verification of Stochastic Systems: Guarantees in the Presence of Uncertainty*. Faculty of Science, Mathematics and Computer Science, RU. 2025-03

A.M. Mir. *Machine Learning-assisted Software Analysis*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-04

L.T. Vinkhuijzen. *Data Structures for Quantum Circuit Verification and How To Compare Them*. Faculty of Mathematics and Natural Sciences, UL. 2025-05

D. van der Wal. *What is the Point? Single-Input-Change Testing a EULYNX Controller*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-06

A. Rosset. *Uniform Monad Presentations and Graph Quasitoposes*. Faculty of Sciences, Department of Computer Science, VU. 2025-07

L. Guo. *Higher-Order Termination with Logical Constraints*. Faculty of Science, Mathematics and Computer Science, RU. 2025-08

D.A. Manrique Negrin. *A Model Orchestra in Digital Twins: A Model-Driven Approach to Integration and Orchestration*. Faculty of Mathematics and Computer Science, TU/e. 2025-09

C.A. Esterhuyse. *Specification-Centric Multi-Agent Systems*. Faculty of Science, UvA. 2025-10

H.M. Muctadir. *Consistency Matters: Building Consistent Digital Twin Virtual Entities*. Faculty of Mathematics and Computer Science, TU/e. 2025-11

A. Stramaglia. *Model Checking Machine-Control Applications*. Faculty of Mathematics and Computer Science, TU/e. 2025-12

M. Saeedi Nikoo. *Supporting business process management: clone detection and recommendation techniques*. Faculty of Mathematics and Computer Science, TU/e. 2025-13

R.B. Rubbens. *Bridging the Implementation Gap: Advances in Model-Based Concurrent Program Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-14

F.I. van der Berg. *DMC Model Checker: Delta-Driven Variable-Length Next-State Generation via Recursive Compression*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-15

D.G. Sprokholt. *Correct Translation between Weak Memory Model Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-16

B. Kohlen. *Verified & Fast Probabilistic Verification: Stepwise Refinement of Probabilistic Model Checking into an IEEE 754 Floating-Point Implementation*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-01

D. Frolich. *Exploratory Language Development*. Faculty of Science, UvA. 2026-02

L.B. van den Haak. *Verifying Optimised Parallel Code*. Faculty of Mathematics and Computer Science, TU/e. 2026-03

A. Papotti. *Human Acceptance of Machine Generated Code Security Analysis*. Faculty of Sciences, Department of Computer Science, VU. 2026-04

N. Doorn. *O bug, where art thou? Improving software testing education*. Faculty of Science, OU. 2026-05

C.R. van der Rest. *Reusable Programming Language Components*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-06

T.T.P. Franken. *A Parallel Programming Perspective – Designing AuDaLa and AuDaLogic*. Faculty of Mathematics and Computer Science, TU/e. 2026-07

T.J. Smeding. *Reverse-mode Automatic Differentiation for Functional Array Languages*. Faculty of Science, UU. 2026-08

S.A. Rukmono. *Towards Automated Software Architecture Explanation: An AI-enabled method and industrial evaluation*.

Faculty of Mathematics and Computer Science, TU/e. 2026-09

R.T.C. Turkenburg. *Comparing and Constructing (Co)Inductive Predicates*. Faculty of Science, Mathematics and Computer Science, RU. 2026-10

P.B.H. Tasche. *Deductive Verification Techniques for Embedded Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-11

W.B. Mbaka. *An Empirical Investigation Into Human Judgments of Security Threats*. Faculty of Sciences, Department of Computer Science, VU. 2026-12

S. Srinivasan. *Taming Uncertainty, Enabling Predictability: Systematic Timing Analysis of Real-Time Systems Across Computation and Communication*. Faculty of Mathematics and Computer Science, TU/e. 2026-13

ACKNOWLEDGEMENTS

I am not the best with words, especially not words of gratitude or affirmation. I have always found it easier to show gratitude in other ways, through time, presence, and actions. But writing this acknowledgement has given me an opportunity to pause, think, and reflect. Putting gratitude into words is harder than simply feeling it, but it has also made me understand more deeply how much people have shaped this life I have built in the Netherlands, from my master's all the way until now.

This thesis is not just the result of a PhD. It is also the result of the people in this life, and the connections I have made along the way. I do not think I fully understood the choices that brought me here when I first made them. But here we are, and I could not be more grateful to close this chapter surrounded by people who have shown up for me in ways I still sometimes struggle to understand.

First and foremost, I would like to thank my supervisors, Nirvana and Geoffrey, who have had to put up with the ups and downs of my PhD struggles most closely.

Nirvana, the first thing that comes to mind when I think of you is that I do not understand how one person can do so much, and do it all with so much care. I like to believe that I am good at understanding people and how they function, but with you, I simply do not understand how you do all that you do. I am endlessly fascinated.

What I admire most is the way you make people feel. You make us feel like we are part of something larger than just a research group. You care so deeply, and it is not care in a passive sense. Your care comes with willpower, force, and action. When someone comes to you with a concern, they know they will be supported in one way or another. To be that present, versatile, and generous with people is something I deeply admire. I hope I get more opportunities to spend time with you, understand you more, and learn from you more, because I genuinely do not understand how your brain works, and I mean that with complete awe. Your level

of care is something I will always be grateful for.

Geoffrey, you are the one who has had to deal with me for the longest. I met you when I first came to the Netherlands, when I was 21, and now I will be almost 30. In many ways, you have seen me grow through almost the entirety of my life here.

What strikes me most about you is your ability to read people. I have walked into your office so many times without saying anything, and somehow you already knew. You knew when something was wrong, when something was going well, when I needed to talk, or when I was trying very hard to pretend that everything was fine. In the moment, especially on bad days, I would think, how can I not get past this man with anything? But looking back, I am so deeply grateful for it.

Your office has always been a space where I could say anything. Whether it was about something I was struggling with, something about research, something about teaching, or something completely unrelated, I knew I would always find a good ear. You would listen, try to understand, and make space for whatever I brought in with me. The moments I cherish the most are the ones where I could simply come and brainstorm with you, tease your brain, and exercise mine as much as I possibly could. Those have been some of the best moments of doing a PhD and doing research with you. I hope I can always come and pick your brain about something, because it is something I genuinely cherish and am deeply grateful for.

I know there were moments where we probably did not think we would get here. But we are here. And this would not have happened without the both of you, Geoffrey and Nirvana.

Mitra, you were one of the first people I knew in the Netherlands, and one of the first people I worked with. Your charm and charisma as a teacher were what first drew me to you, and I am ever so grateful that they did. I am still so impressed by the way you teach, by the way you can take something in your head and make it understandable to anyone. If it were not for you, I would not have done a PhD. In many ways, you were the catalyst for all of this.

But apart from you as a teacher and professor, I have also come to appreciate all the different facets of your life. I have always gravitated towards people who do many different things, and do them well. You are one of those people. You are amazing at what you do at work, but also at so much outside of it. Your optimism, joy, and infectious enthusiasm have had such an impact on the people around you. I hope you are always able to tap into that, not just for the people around you, but also for yourself.

Jolande, you sit in the middle of it all. Every time I visited one of the professors' offices, especially after coming back from a heavy meeting, I would pass by you. And what a wonderful person to walk past. You are such a bundle of joy. You

cheer people up in so many ways, with candy and sweets, with your enthusiasm, with your willingness to listen and help, and with your deep investment in all of us doing well. You are the person everyone notices is missing the moment you are gone for a few days. Even when I was not directly talking to you, I loved simply hearing your laughter trickle through the corridor. The joy that brings is something special.

Richard, what I would like to thank you for most is your willingness to help. Your technical expertise is appreciated by everyone, but it is your generosity with that expertise that makes such a difference. You are always ready to give your time, find a solution, and make things easier. Thank you for all the times you helped me in ways I do not even fully understand, and for making so many things work behind the scenes.

The people who have probably had to spend the most time with me are, of course, the people who sat in my office. Gizem, you have been by my side, literally, through most of this journey. I would not have wanted anyone else sitting next to me. You have been such a constant support and such a constant ear. Every time I had a tough meeting or a difficult day, I could come back to the office, see you, and dump all my thoughts and feelings. You received them all, and I am so grateful for that. Vasilis and Saeed, you have always pulled my leg, played around, and tortured me the way one would torture a sibling. While I would probably never say this to your faces, I am grateful for all the laughter that brought me. Fatih, thank you for saying the most random things and somehow always catching me off guard.

To the people who got some respite from me by not sharing an office with me, but still had to put up with me often enough, I am so grateful to have had you around. Michele, thank you for always being ready to spend time, socialise, go out for a drink when it was really needed, lend an ear, and cook amazing food for us. Iram, thank you for all the times I have stayed at your place, eaten your food, and gone home with even more food from you. Where I come from, we bond over food and take care of people through food. You have done that for me so many times, and I am deeply grateful. Ali, thank you for taking us to the best places and for sharing your hobbies with us. Riz, thank you for your immense love of language, and for all the conversations about history, society, and everything in between. Alex, thank you for sharing all your journeys with us. Shengyuan, thank you for always catching people off guard in the most unexpected and innocent ways. Devilliers, thank you for bringing your full competitive spirit to every game and making all of it so much more entertaining. Nan, thank you for helping me integrate into the group in my earlier days. Olav, thank you for being such a cheerful presence and for being so nice to work with. Yimi, I am so glad that what started through work turned into such a lovely friendship. Your infectious

happiness pulled me out of some very difficult times, more than you perhaps realise.

Mario, Jonas, and Veronica, thank you for coming all the way to work in our research group, and especially for working on real-time systems. I had always hoped for more people in the group to work on this topic, and I could not have been happier that it was the three of you.

Let me not only talk about the people in the office. While it may have seemed, at times, that my entire life was just work, it did extend beyond that.

Prernna, thank you for always welcoming me into your home, for giving me an ear when I needed to talk, and for taking me out and making sure I forgot my problems for a while. Supriya, thank you for feeding me through quite a bit of my master's, and for being the kind of friend with whom there is no pretence. Even if a lot of time passes, I know we will simply pick up from where we left off. Both of you have been such a big support. Just knowing that you exist right across the border has given me strength in ways I cannot fully explain.

And then, of course, Benjamin. Where do I even begin with you? If there is one person I have learned a lot from, whose viewpoints I deeply enjoy and trust, it is you. But the biggest thing I am grateful to you for is something that may sound simple, but to me is anything but simple: you have helped me relax.

I am a very highly strung person. I do not know how to let my guard down easily. I do not know how to switch off. And you are one of the very few people I trust enough to think, okay, I can turn my brain off now. I can relax. I will be okay. If I have been able to work in a way that is more sustainable, and more importantly, live in a way that is more sustainable, it is in large part because of you.

I am also grateful for the way you think. In the big picture, we often see things similarly, but the ways in which we get there can be quite different. I love seeing the world as you see it, and I love sharing how I see the world with you. You have given me more confidence in the way I see the world and in the way I see myself in it. That is a very big gift for anyone to give.

And with you also came something I had deeply missed here, a sense of close family. To your family, thank you for taking me in as one of your own. For every event, every celebration, every important moment, you have made sure I was part of it. Petra, thank you for the effortless way in which you allow me to simply be. I admire the way you seem to do insanely difficult things with such ease. Mari, thank you for always trying to communicate with me, through my broken Dutch and your broken English. Somehow, we still get the point across. Willemeike, thank you for the gifts, the generosity, and the care behind them. Jeroen, thank you for being curious about what I do and for wanting to understand my work.

And to the little ones, Valentijn and Olivier, thank you for the joy you bring in

your own ways. Valentijn, thank you for taking me as part of the family from the very beginning, in that effortless way only children can. I have loved your curious questions and the conversations they lead to. Olivier, you are still such a baby, running around and learning to say your first words, and somehow one of those words is my name. When you say “Nidhi buiten” because you want me to play outside with you, it absolutely melts my heart. To the whole family, thank you for the immense generosity with which you have welcomed me.

Last but not least, of course, I come to my family back in India.

It goes without saying that this PhD is as much theirs as it is mine. It is hard to write about family without falling into clichés, because of course I am grateful for your support, and of course I am grateful for the strength it must have taken to send me away and let me build a life so far from home. But beyond the clichés, there are a few things I want to say more specifically.

Appa, thank you for never leaving me wanting for anything. Anything I wanted to do, I knew you would be part of it. Your support has always been quiet, but steady. You let me do what I needed to do, and somehow, you would make it happen.

One of the things I have always felt safe in is your strong sense of morals. You are headstrong in what you believe is right, and there was a safety in growing up around that kind of certainty. You also never trusted me blindly, but you trusted me fairly. You asked questions, made me justify my decisions, and pushed me to argue for them. But once I could explain myself in a way that made sense, you let me have my way. That trust in my judgment has made me stronger and more independent. Your resourcefulness, your quiet support, and your trust have been some of the most positive forces in my life.

Amma, as I grow older, I am falling into yet another cliché: I am very much like you in so many different ways. I see it in the way I worry about people. I see it in humility, in the care, and in the instinct to put others first. I hope you see one day that the qualities you have are a strength in you, just as much as you see them as a strength in me. One of the things I respect and thank you for the most is your humility. Sometimes, that humility comes with making yourself smaller so that others can take up more space. And while I wish you would put yourself first sometimes, I also understand it more and more because I see parts of that in myself now.

But more than anything, I admire your endurance. You have endured so much, and yet you keep going, quietly and steadily. I have learned from that more than I can say. This PhD has demanded quite a bit of endurance from me, and I sincerely get that from you and you should be proud of yourself for that. I know I have inherited something from you, and I am deeply grateful for it.

To both of you, one of the biggest things I want to thank you for is the way you

chose to be parents. You broke away from many of the ideas of what parenting may have meant to you growing up, and gave us something different. You always allowed me to question you. You never imposed yourselves simply as authority figures in the house. Our opinions were valid, and we were allowed to question, argue, and have a back and forth. If I had not developed that personality at home, I would not have been able to come here and have that same back and forth with my professors, colleagues, and the people around me. Thank you for allowing dialogue.

And finally, I come to my brother, to whom this entire thesis is fully dedicated.

I do not even know how to begin talking about you. I carry all the versions of you in my head at once, from the little baby boy you were to the person you are now. Sometimes, you still appear in my mind as the small child I loved spending time with and watching grow. And then sometimes, you say something so wise, so far beyond your years, that I find myself thinking, how did that happen? When did he grow up so much?

Through all these versions of you, you have always been a source of learning for me. When you were young, I learned from watching your curiosity. As you grew older, the way you saw the world continued to catch me by surprise. You say things that make me think, yes, that is true, but I would never have seen it that way. You are also one of the few people, perhaps the only person, to whom I can say absolutely anything and know that I will be understood. There is a freedom in that which I do not take lightly.

When I left for the Netherlands, you were still so young. In my head, you were still a baby, even if technically you were a young boy. You were shorter than me then, and now you tower over me. I thought I would want you to stay small forever, but I have loved watching you become who you are. Your emotional intelligence, the way you understand people, the way you understand me, I look up to you in more ways than I can properly put into words.

I would also like to extend my gratitude to my relatives and friends, near and far. Even if I have not mentioned each of you by name, please know that your presence has mattered. Through calls, messages, visits, shared meals, celebrations, and simple moments of companionship, you have helped me stay connected to the people and places that shaped me. Thank you for your support, your encouragement, and for continuing to be part of my life despite the distance and the passing years.

Thank you all for allowing me and accepting me as I was, even when it was difficult for me to do so myself. Thank you for supporting me when it was difficult for me to support myself. Thank you for putting up with me, for going through all of this with me, and for walking with me all the way here.

I hope we have many more steps left to go.

LIST OF PUBLICATIONS

1. Srinidhi Srinivasan, Geoffrey Nelissen, and Reinder Bril. “Work-in-Progress: Analysis of TSN Time-Aware Shapers using Schedule Abstraction Graphs”. In: Dec. 2021, pp. 508–511. DOI: 10.1109/RTSS52674.2021.00052

2. Srinidhi Srinivasan et al. “Analysis of TSN Time-Aware Shapers Using Schedule Abstraction Graphs”. In: *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*. Vol. 298. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 16:1–16:24. DOI: 10.4230/LIPIcs.ECRTS.2024.16

3. Srinidhi Srinivasan, Mario Günzel, and Geoffrey Nelissen. “Response-Time Analysis for Limited-Preemptive Self-Suspending and Event-Driven Delay-Induced Tasks”. In: *2024 IEEE Real-Time Systems Symposium (RTSS)*. 2024, pp. 29–42. DOI: 10.1109/RTSS62706.2024.00013

4. Yimi Zhao et al. “Work-in-Progress: Generating Counter-Examples to Schedulability Using the Schedule Abstraction”. In: *2023 IEEE Real-Time Systems Symposium (RTSS)*. 2023, pp. 459–462. DOI: 10.1109/RTSS59052.2023.00052

5. Jonas Hansen et al. “Exact Schedulability Analysis for Limited-Preemptive Parallel Applications Using Timed Automata in UPPAAL”. in: *2025 Design, Automation Test in Europe Conference (DATE)*. 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992936

6. Srinidhi Srinivasan et al. “Analysis of TSN Credit-Based Shapers Using the Schedule Abstraction Graph Methodology”. Under review

ABOUT THE AUTHOR



Srinidhi Srinivasan was born on 23 November 1996 in Chennai, India. From 2018 to 2020, she pursued a Master's degree in Embedded Systems at Delft University of Technology in Delft, the Netherlands. In 2020, she began her PhD research at Eindhoven University of Technology in Eindhoven, the Netherlands, within the Interconnected Resource-aware Intelligent Systems (IRIS) research group. The results of this research are presented in this dissertation. She is currently employed as a teacher at Eindhoven University of Technology, also within the IRIS research group.

