Modelling and analysis of multi-scale streaming applications

Citation for published version (APA):
Seyedalizadeh Ara, S. (2019). *Modelling and analysis of multi-scale streaming applications*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven.

Document status and date:
Published: 21/03/2019

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Modelling and Analysis of Multi-Scale Streaming Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit
Eindhoven, op gezag van de Rector Magnificus prof.dr.ir. F.P.T.
Baaijens, voor een commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op vrijdag 21 maart 2019 om
16:00 uur

door

Seyedhadi Seyedalizadeh Ara

geboren te Tabriz, Iran

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

voorzitter:	prof.dr.ir. J.H. Blom	
1e promotor:	prof.dr.ir. T. Basten	
copromotor:	dr.ir. M.C.W. Geilen	
leden:	prof.dr. E.A. Lee	(University of California at Berkeley)
	prof.dr. B. Heidergott	(Vrije Universiteit Amsterdam)
	prof.dr.ir. C.H. van Berkel	
	prof.dr.ir. J.P.M. Voeten	

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

Modelling and Analysis of Multi-Scale Streaming Applications

Hadi Alizadeh Ara

The work in this thesis was supported by the ARTEMIS joint undertaking through the ALMARVI project (621439) and ITEA 3 Project 14014 ASSUME.

Copyright © Seyedhadi Seyedalizadeh Ara 2018.
All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Cover design by Joli Seelen

Printed by Proefschriftmaken, The Netherlands

A catalogue record is available from the Eindhoven University of Technology Library ISBN: 978-90-386-4721-0.

To my beloved brother, Hessam

Summary

Modelling and Analysis of Multi-Scale Streaming Applications

Embedded systems are computing systems that provide a number of specific functionalities within an electronic device such as a smartphone. Some embedded systems run programs that process continuous streams of data received from several sources such as sensors, a network or other devices. For instance, video applications on smartphones process the data received from the camera or online video streaming services. These programs are referred to as streaming applications. In domains such as multi-media processing, streaming applications have strict real-time performance requirements, often specified in terms of minimum throughput and maximum latency. For cost and power reasons, multiple streaming applications in a system may be realized on a multi-processor system, and share its heterogeneous computing elements such as processors and memory. To ensure that every individual application will have a guaranteed performance, the applications are designed based on a model of computation (MoC) that accurately captures their timing behaviour in the system. In a design iteration, an application that is modelled by an analyzable MoC, can be analytically evaluated for its real-time performance. Synchronous dataflow (SDF) is an analyzable MoC that is widely used to model and design streaming applications. In particular, these models are used to find optimal processor and memory shares for the applications to have executions with the required performance.

Today's competitive market on embedded systems has led to the emergence of more and more complicated streaming applications. For instance, new digital cameras utilize advanced filtering applications to produce high quality images. Complex applications are characterized by their dynamic execution behaviour. These applications include operations that change their computation loads and/or data dependencies, depending on their operating modes or input data. In a multi-scale application, the operations with changing behaviours act at multiple scales of data granularity. For instance, a filtering

application may have dynamic operations at frame, line and pixel levels. SDF can only provide a conservative abstraction for dynamic applications, as it does not have a means to express dynamism. Scenario-Aware Dataflow (SADF) is an extension of SDF that can express dynamism. In this model, scenarios describe SDF behaviours that correspond to particular modes or input data types. The dynamism is expressed via (non-) deterministic scenario transitions. In SADF, the set of all meaningful application behaviours is modelled as a set of scenario sequences, which can be represented using a formal language. The full range of analysis methods for SADF models is not developed yet, and the existing ones do not scale for multi-scale applications. This thesis provides an approach to model multi-scale applications using SADF. In addition, it provides a number of scalable analysis methods for SADF models of multi-scale applications.

We observe that the timing behaviour of multi-scale applications follows a periodic pattern of small, static behaviours that is composed of a few behaviours associated with the coarser grain operations and many repetitions of behaviours associated with finer grain operations. We perceive each distinctive small behaviour as a scenario. Consequently, the behaviour of the application is described as a periodic sequence of scenarios, some of which are repeated many times. Such behaviours can be modelled by an SADF. To compactly represent the repeated (sub-) sequences we introduce a representation for SADF, where the language of the sequences is represented as a regular expression with an explicit repetition construct.

A modular compositional model can facilitate the modelling process of complex applications by allowing individual component models to be composed in a recursive manner to generate a hierarchical model of a complex application. A complex application, such as a multi-resolution filter is structured as parallel pipelines of multi-scale components, each with a possibly different periodic pattern. Composing models with periodic patterns results in a model with a periodic pattern with a common hyper-period. This common periodic pattern is hard to derive by hand. We propose an efficient algorithmic method that given the periodic scenario sequences of the components by regular expressions, generates a sequence of composite scenarios in a compact regular expression representation.

For SADF with expression representation, we provide a compositional approach for exact, worst-case throughput and maximum latency analyses. The proposed throughput analysis shows an enhanced scalability for multi-scale dataflow models compared to the state of the art. Scalability improvement is achieved by the proposed expression representation to recognize the repeated patterns and the fact that the throughput analysis on repeated scenarios can be done in logarithmic time in the number of repetitions instead of linear time as in the state of the art methods. The proposed latency analysis, which is similarly scalable, is the first exact latency method for SADF models. The latency is recursively computed by decomposing the regular expression and computing the latency of sub-expressions in a bottom up approach.

The execution traces of applications can be used to understand the behaviour of the applications and find the opportunities for performance improvement or detect anomalies. Re-using elements of the latency analysis, we provide a technique to quickly generate execution traces of a particular scenario in the pattern without performing a detailed simulation of the sequence that is followed by that scenario. This is very helpful in case of multi-scale applications, as their periodic sequence is often very long.

Distributing the available storage space on an embedded platform to the buffers in a streaming application is a complicated design challenge. The capacities of buffers in an application affect the throughput of the application. Since storage space is a scarce resource on embedded platforms, optimal points in the space of throughput-storage space trade-offs should be found. We provide the first throughput-buffering trade-off analysis for applications modelled as SADF. We use our scalable throughput analysis in a guided design space exploration to obtain the trade-offs. At every exploration step, the exploration prunes the exploration space without losing any optimal points. Consequently, the analysis terminates in a reasonably short time.

Processor sharing is used to reduce cost and energy. Budget scheduling is a common strategy to share the processing power of a platform among real-time applications since it bounds the interference between the timing behaviour of individual applications. As the final contribution in this thesis, we provide a means to determine conservative, but tight timing bounds for scenarios of an SADF that are mapped onto a shared multi-processor platform.

In conclusion, this thesis presents a number of contributions that can be used within an automated framework to realize multi-scale applications on multi-processor systems.

Contents

1 Introduction 1
- 1.1 Real-Time Streaming Applications . . . 1
- 1.2 Design Challenges . . . 3
  - 1.2.1 Concurrency . . . 4
  - 1.2.2 Timing Predictability . . . 6
  - 1.2.3 Multiple Applications . . . 8
  - 1.2.4 Dynamism . . . 9
  - 1.2.5 Multi-Scale Behaviour . . . 11
  - 1.2.6 Composite Behaviour . . . 12
- 1.3 Problem Definition . . . 13
- 1.4 Contributions . . . 14
- 1.5 Thesis Overview . . . 17

2 Preliminaries 19
- 2.1 Notations . . . 19
- 2.2 (max,+) Algebra . . . 20
- 2.3 Synchronous Dataflow . . . 21
- 2.4 Synchronous Dataflow Scenarios . . . 23
- 2.5 Regular Languages . . . 27
  - 2.5.1 Finite State Machines . . . 27
  - 2.5.2 Regular Expressions . . . 28
- 2.6 Scenario-Aware Dataflow . . . 29
- 2.7 (max,+) Characterization of SDF Scenarios . . . 33
  - 2.7.1 The State Equation . . . 34
  - 2.7.2 The Output Equation . . . 36
- 2.8 FSM-SADF Throughput Analysis . . . 37
- 2.9 SADF Buffer Requirements . . . 40
- 2.10 Open SADF . . . 41

3 Modelling Multi-Scale Applications 45
3.1 Modelling a Convolution Filter 46
3.2 Modelling an H.263 Decoder 52
3.3 Generating Execution Traces 54

4 Scalable Timing Analysis 57
4.1 Throughput Analysis 57
4.2 Latency Analysis 67
4.3 Evaluation 78
4.4 Related Work 82
4.5 Conclusion 84

5 Compositional Modelling 85
5.1 Introduction 85
5.2 Producer-Consumer Composition 87
5.3 Feedback Composition 97
5.4 Experimental Results 99
5.5 Related Work 100
5.6 Conclusion 101

6 Throughput-Buffering Trade-Off Analysis 103
6.1 The Main Idea 104
6.2 The Extended Matrix Representation 106
6.3 The Critical Buffers of an FSM-SADF 110
6.4 Design Space Exploration 111
6.5 Evaluation 117
6.6 Related Work 119
6.7 Conclusion 120

7 Timing Bounds on the WC Execution 121
7.1 Introduction 122
7.2 Worst-Case Timing Bounds 124
  7.2.1 Worst-Case Response Times 124
  7.2.2 The State Matrix With Worst-Case Response Times . . 126
  7.2.3 Accumulated Worst-Case Response Times 129
  7.2.4 Symbolic Simulation With Accumulated Worst-Case Response Times 130
7.3 Tighter Worst-Case Timing Bounds 132
7.4 Evaluation 137
7.5 Related Work 139

7.6 Conclusion 140

8 Concluding Remarks 141

8.1 Conclusions 141

8.2 Future Work 143

Acronyms 155

Acknowledgments 157

Curriculum Vitae 159

Publication List 160

Chapter 1

Introduction

1.1 Real-Time Streaming Applications

Embedded systems are information processing systems embedded into enclosing products [1]. Embedded systems have a large-scale market that includes multi-media, automotive and health-care. Smartphones, smart TVs, digital cameras and game consoles are examples of embedded systems in the multimedia domain. In the automotive domain, embedded systems include a wide range of systems from controllers of individual parts such as the engine and the transmission system to complicated automatic control features such as cruise and traction control systems.

Embedded systems use peripherals such as sensors and actuators to interact with their environment. A digital camera uses an image sensor to capture images, a light sensor to detect the brightness of the environment, control buttons to get user inputs and a microphone to record audio. In addition, it has mechanisms to move the lens and control the shutter speed, and a display to show picture previews. Embedded systems of this kind can also be classified as cyber-physical systems, as they constantly interact with physical processes. Timing management between the cyber and physical parts is one of the major challenges in these systems. For instance, recording a video using a digital camera with enabled automatic focus feature requires calculations to detect the contrast of the frames at specific, often short time intervals, and a special motor which is able to adjust the lens elements shortly after the end of the detection intervals to facilitate automatic focus.

Embedded systems use their processing capabilities to run various algorithms that are needed to use or control their peripherals. In a digital camera, the light rays pass through the lens, directed into a colour filter and finally hit the image sensor. The image sensor produces an array of analog signals called colour filter array. The signals contain the colour information of every pixel in the sensor. This signal array is first transformed into a digital stream of pixel data using an A/D converter. A number of image processing algorithms are needed to correct or compensate for artifacts that are caused by physical phenomena in the optics or colour filter limitations, e.g., lens distortion compensation and colour interpolation. Then, different algorithms are needed to extract some features to be used in the automatic focus control and image enhancing, e.g., edge detection. A typical digital camera executes a high stack of algorithms to transform the stream of raw data generated from an image sensor to a stream of frames in a format that can be displayed on the screen or stored in an SD memory [2].

The algorithms used in embedded devices are implemented as embedded computer programs called streaming applications [3]. A digital camera implements its algorithms as a pipeline of various streaming applications that include image filtering applications, image enhancing applications and video tracking applications [4]. Smartphones offer wireless data communication through several standards such as cellular network standards, Bluetooth and Wifi. The program used to establish a connection through each of these communication protocols is a streaming application. Other streaming applications in smartphones include applications of online video and audio streaming services such as YouTube, Spotify etc. They execute algorithms that convert the compressed data received from their servers to audio or video content that is playable on smartphones.

Many streaming applications require real-time computing, which means that the processes on their input stream must be guaranteed to be within a specified time constraint [5]. Otherwise, the application may cause inconvenience to the user or in some cases it may not be useful anymore. For instance, game consoles constantly perform graphical processes depending on the inputs generated from the controller. Suppose a person is playing a graphics-intensive video game. If the processing rate drops below 25 frames per second (fps), he will experience frequent screen freezes, which often makes game play impossible. The rate at which the data streams are flowing in a streaming applications is called the throughput. Multi-media and communication applications are among streaming applications with a minimum throughput constraint [6]. In contemporary digital cameras, the user can opt for recording videos at 60fps. To achieve this frame rate, all streaming applications in the pipeline must be guaranteed to provide at least 60fps.

Processing data takes time. Consequently, streaming applications introduce a processing delay between the output and input streams. Digital cameras suffer from something called lag time. When you press the shutter button, the camera may seem to take a photo instantly; however, this is not actually what happens. There is a delay between the time when you press the button and when the photo is actually taken. Lag time is a result of what your camera does when you press the shutter button. The camera will first need to run control applications to focus the shot and then it will need to open the shutter to let the light in. Once the light comes in through the shutter and hits the image sensor, the image is taken. Then, it takes additional time before it becomes visible as a preview. All of these processes result in a lag time which could make you miss your shot. In other applications the delay might be caused because the application needs an initial amount of data before it can start processing and producing an output stream with a certain rate. For instance, video tracking and object recognition applications need to buffer a certain number of frames before they can start functioning. This is because object recognition and tracking algorithms executed in such applications require computations that span over a window of several frames [7]. The initial waiting time for an application before getting the output stream with a specific throughput is called the latency.

For some applications the maximum latency is a requirement. Image-guided surgery is a surgical procedure where the surgeon uses tracked surgical instruments in conjunction with real-time x-ray images from the anatomy of the patient to perform safer and less invasive operations compared to the traditional methods. The surgery unit used in this type of surgery has a video application that transforms the data produced by the x-ray detector to the image frames shown on the screen. To maintain the hand-eye coordination of the surgeon, the maximum latency of the frames is bound to 150ms at a frame rate of 60fps [8]. The latency is a crucial constraint also in any feedback control system such as automatic focus systems in digital cameras [9] or cruise control systems in modern cars [10].

1.2 Design Challenges

The performance of a streaming application, i.e., its throughput and latency, depends on how it is implemented. Every implementation has a timing behaviour that represents how the execution of the application progresses over time, for instance, at what time intervals its various functions are being executed or at what time instants it accesses system resources such as memory

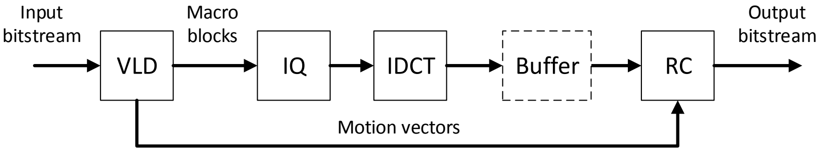


Figure 1.1: Structure of an H.263 decoder application.

and processors. The timing behaviour is affected by characteristics of both the application and the underlying computer system [5]. Obtaining a design that guarantees a certain timing performance is a challenging task, since the timing behaviour is affected by many factors, some of which may be non-deterministic. In this section we discuss some of these challenges and their state of the art solutions.

1.2.1 Concurrency

Computing systems with multiple processors allow system designers to exploit the inherent task parallelism in concurrent applications to achieve the required performance [11]. We illustrate the task parallelism by an H.263 decoder application, which will be used in the remainder as a running example. H.263 is a video compression standard designed as a compressed format for efficient video transmission in video-conferencing. An H.263 decoder is an application that transforms the encoded video stream after it is transmitted (Figure 1.1) [12]. The decoder accepts a video stream of encoded frames as input. The frames require the following tasks: Variable Length Decoding (VLD), Inverse Quantization (IQ), Inverse Discrete Cosine Transformation (IDCT) and Reconstruction (RC). The VLD task decodes the frames into small blocks of data items called macro blocks, where every macro block contains six smaller blocks of data. It also generates a motion vector for each macro block, which defines the relative motion of a macro block from one frame to another. The IQ and IDCT kernels process the blocks. When all blocks in a frame are processed, the RC executes to reconstruct the decoded frame from the blocks. The RC kernel also performs motion compensation on the frames using the motion vectors of all macro blocks in the frame.

Usually the decoding (VLD), the block computations (IQ and IDCT) and the reconstruction (RC) tasks are each mapped to a separate processor to exploit the concurrency. For instance, when the RC is executing on the blocks of the current frame, the remaining tasks can be executing on the blocks of the subsequent frames. Concurrent execution can also happen within a single frame. For instance, when the IQ starts operating on the blocks generated from a VLD execution on a macro block from a frame, the VLD can start decoding another macro block from the same frame, simultaneously. As another advantage, each task can be mapped to a processor that is fit for the task. For example, the IDCT in the H.263 decoder application is usually processed using vector processors [12].

The challenges in a concurrent implementation are rooted in the concept of concurrency, i.e., when more than one thing is happening at a time. Suppose in an H.263 decoder IDCT and RC are running on separate processors and use a buffer, where IDCT could write its output data, i.e., the processed blocks, and RC could read the blocks and reconstruct the frames. Things can easily go wrong. For instance, when RC starts reading from the buffer before IDCT writes all blocks of the first frame on the buffer, or when IDCT completes processing a new block and overwrites one of the old blocks, while RC has not read that old block yet. As easy as it may sound, synchronization and communication between tasks can be challenging, especially in applications with varying, possibly complicated data dependencies [13]. Careless concurrent implementations are prone to concurrency issues such as deadlocks and race conditions. A deadlock occurs when multiple tasks are waiting for each other, but never proceed. In a race condition, the output of an application becomes unexpectedly dependent on the execution order of its concurrently running tasks [14]. Deadlocks and race conditions should be avoided at all costs since they lead to a type of non-deterministic behaviour that violates performance guarantees required by real-time applications.

Models of Computation (MoCs) can be used to obtain a correct behaviour of the applications by providing a formal reasoning about their concurrent behaviour. Kahn Process Networks (KPNs) [15] are one of the popular MoCs because of its useful properties. It models applications by a collection of individual processes (tasks) that are communicating through unbounded First-In-First-Out (FIFO) channels. KPNs enforce a set of communication rules in the network to be able to mathematically prove the property of determinism, the property that the application output is deterministic regardless of the execution order of the (concurrent) processes in the network. A KPN rule states that processes read and write atomic data items (also called tokens), from and to channels. Another rule states that writing to a channel is non-blocking, i.e. it always succeeds and does not stall the process, whereas reading from a channel is blocking, i.e. a process that reads from an empty channel stalls and can only proceed when the channel contains sufficient tokens. For instance, in the H.263 decoder, this rule prohibits the start of the RC task before all the processed blocks of a frame are accumulated in the buffer.

KPN is an example of many MoCs proposed for concurrent computation. Each model occupies a different spot in the space of expressivity and analysability. For instance, in KPNs, the sizes of the buffers cannot be determined at design time in general, and therefore they require a run-time scheduler that schedules the processes, manages the buffers and avoids deadlocks [16, 17]. In the next sub-section, we discuss that by further restricting the behaviour of the network in some MoCs, for instance by adding more communication rules, properties such as deadlock freedom and boundedness of buffers can be proved for the network at design time. Furthermore, their timing behaviour then becomes predictable.

1.2.2 Timing Predictability

An application with a guaranteed performance requires a computer system that is predictable with respect to its timing requirements. In such a system it is possible to define a reasonable maximum response time (timing bound) for every action that is happening in the system, such as the execution of computational tasks, data communications, external interrupts, memory accesses, etc. This enables bounds on performance, such as upper bounds on latency or lower bounds on throughput, to be provided. However, such a system ties the designers' hands in use of many mechanisms in general-purpose computing that provide a good average performance in an easy programming model, but make the system's performance less predictable. For instance, snooping cashing [18] dynamically manages cache coherence for a good average performance, but at the cost of less predictable delays since the time required for memory access depends on the state of several caches, which is hard to predict. Real-Time Operating Systems (RTOSs) in general computing provide scheduling mechanisms for tasks, but they also abstract the detailed behaviour of a task, i.e., how it accesses memory, and its flow of control [5]. The detailed behaviour of tasks can influence the execution time of the task and consequently, the system schedule that is managed by the RTOS. When realizing a real-time application, the designer should use or provide mechanisms to ensure a predictable execution of the application, and he cannot rely on any available abstractions or mechanisms that introduce unpredictable timing behaviour.

All mechanisms used in a predictable system such as resource managers and schedulers should be formally verified for performance robustness. Simi-

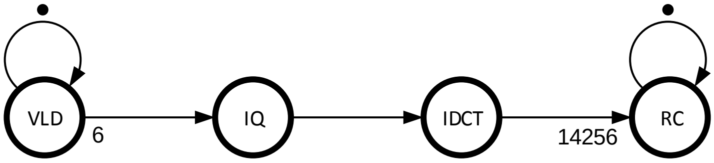


Figure 1.2: SDFG of an H.263 decoder application.

lar to the formal verification of a correct concurrent computation, performance verification approaches require performance models of the application and the underlying system [19]. Synchronous Dataflow (SDF) [20] is a Model of Computation (MoC) that is widely used to model and design concurrent streaming applications [6] [21] [22]. SDF is supported with analysis methods that can be used to verify the throughput and the latency of an application that behaves according to an SDF model.

SDF describes the application by a directed graph. Figure 1.2 shows a Synchronous Dataflow Graph (SDFG) of an H.263 decoder. In this graph, nodes depict actors. Actors represent the application tasks and actor firings correspond to task executions. Directed edges in the graph represent channels. Channels represent dependencies between actor firings to ensure a correct execution according to the data dependencies or control decisions. The dependencies are enforced by production and consumption of entities called tokens. When an actor starts firing, it consumes tokens from its input channels, and when it completes the execution, it produces tokens on its output channels, depicted as black dots. As a result, an actor firing cannot start before having received enough tokens on all of its input channels. All firings of an actor consume and produce constant numbers of tokens per channel, called the consumption and production rates. The rates are given as edge annotations in the graph (rates of 1 are not shown to avoid cluttering). For instance, when actor VLD fires, it produces 6 tokens on the channel from VLD to IQ. When SDF is used as a timing performance model, the actors are assigned execution times.

Compared to KPNs, SDF has a limited expressivity. This is because all firings of an actor in SDF are the same in terms of actor rates and execution time, where as executions of a KPN process might be different from one another. In exchange for the limited expressivity, SDF provides properties that are very useful in the design of real-time streaming applications. Timing predictability is one of these properties. In an SDFG given worst-case execution times for every actor in the graph, the worst-case throughput and maximum end-to-end latency of the graph can be obtained at design time, together with a static periodic schedule for every actor firing. SDF can also be analysed for deadlocks and boundedness of its buffers at design time. Using a more complete analysis such as a throughput-buffering trade-off analysis, all Pareto (optimal) points in the space of throughput and buffer sizes can be obtained for an SDF [6]. Besides being an analytical performance analysis model, SDF can also be used as a concurrent programming language that can be compiled to generate a concurrent program code [23].

1.2.3 Multiple Applications

Often multiple streaming applications in a system are realized on a single computer system and share its computing elements such as processors and memory for cost and power reasons [6]. This complicates the design process, because applications may influence each other's timing behaviour, for instance, when an application claims a shared resource and prevents other applications from having an access to it when they need to. To solve this problem, embedded system designers use resource arbitration policies that bound this influence, to be able to ensure performance robustness [6]. Even then, a challenging question is how much resource should be allocated to every application to satisfy its constraints and whether the resource usage can be optimized.

The system memory is used to realize buffers for the applications. The capacities of the buffers influence the throughput of the application by altering the task schedules. For instance, consider the H.263 decoder (Figure 1.1), operating on frames of size 352×288 pixels (also called CIF size). Every frame contains 2376 macro blocks. The buffer between IDCT and RC must have enough capacity to store 2376 macro blocks, otherwise the application cannot execute, i.e., it deadlocks. This is because IQ will not start executing after the buffer becomes full and consequently, the decoding of the first frame never completes. If the buffer can store exactly 2376 macro blocks, the IDCT and RC always execute sequentially even if they are running on different processors. This is due to the fact that IDCT will not start processing the blocks of the next frame until all blocks in the buffer are reconstructed into a frame and the buffer is emptied. By increasing the capacity of the buffer by one block, the IDCT on the first block of the next frame can run in parallel with RC execution in the current frame. By further increasing the capacity, more IDCT executions can be executed in parallel with RC, as RC takes hundreds of times as much time to complete as a single execution of IDCT. Increasing the capacity block by block, will increase the throughput, assuming that this buffer is the limiting factor for the throughput (i.e., when other tasks have shorter execution times compared to RC and other buffers are large enough). If we keep increasing the capacity, at some point the RC on the next frame will be fully parallelized with IDCT executions. From this moment, increasing the capacity of this buffer will not affect the throughput as the application has reached its maximum throughput.

The capacities of the buffers should be be optimized for the required throughput, since the buffers are implemented using costly and power consuming memory blocks. Moreover, to have an application with a predictable performance in a multi-processor system, every buffer is implemented such that it has an exclusive access to a number of memory blocks. This may quickly fill up the memory if the buffers are sized with generosity. The significance of the buffer minimization problem is higher when there are multiple applications running on the same system, because the over-allocation of memory reduces the number of applications that can run on the system. Applications that are modelled by SDF graphs can be analysed for optimal buffer requirements given a throughput constraint. This allows the designers to realize multiple applications on a multi-processor system with a minimum memory usage [6].

A similar challenge holds for processor sharing. Budget scheduling is a common strategy to share the processing time of processors among real-time applications, since it bounds the interference between the timing behaviour of individual applications [24]. A budget scheduler allocates time budgets to applications in every processor to run their tasks. A higher budget results in a faster execution, and possibly a higher throughput and lower latency. A challenge is to determine budgets for every application such that the constraints of all applications are satisfied. Conservative throughput bounds can be obtained for an SDF graph in which the actors are bound to multiple processors [25].

1.2.4 Dynamism

Applications may be subject to dynamic behaviours. For instance, input frames in an image recognition application may be taken from various scene types in different weather conditions. Moreover, the frames may contain a different number of objects every time [26]. A similar scenario holds for a video game running on a game console. The video game is supposed to run smoothly with different combinations of physics and actions happening at the same time. An application with a guaranteed performance should be able to deliver the required performance for any possible combination of these varying conditions. Varying conditions create a dynamic timing behaviour that should be carefully studied to ensure performance robustness.

Dynamic applications are often characterized as tasks that change their execution time and/or data dependencies at run-time [27]. For instance, video tracking applications include tasks with execution times that are determined by the number of tracked objects in the input video [28]. In a stereo audio decoding application, the decoding task alternates between decoding the data from left and right audio channels [27]. This results in a timing behaviour that changes at run-time either deterministically (as in the stereo audio decoder) or non-deterministically (as in the video tracking application).

Dynamism complicates the design process because verifying the behaviour of the application in any possible combination of varying conditions leads to an explosion in test cases. An easy solution to achieve a guaranteed performance in dynamic applications is to rely on static worst-case assumptions, where every task is assumed to have a worst-case execution time and data dependencies [25]. However, this approach results in pessimistic performance estimations, and pessimistic estimations cause in turn over-allocation of the resources. For instance, SDF can provide a good abstraction only for static applications, as it does not have a means to express dynamism (the actor rates and actor execution times are constant).

Cyclo-Static Dataflow (CSDF) [27] is an extension of SDF that enables the design of streaming applications with periodically varying task behaviours. In this model the actor rates and execution times can periodically change. For instance, a stereo audio application that alternates between right and left channel processing can be modelled with CSDF. Throughput [29], end-to-end latency [30], and resource sharing analysis techniques [29] are generalized for applications modelled as CSDF.

For some dynamic applications, changing behaviours occur not only on the task level but also in the structure. Moreover they might be non-deterministic. As an example of such dynamic applications, consider an MPEG-4 Simple Profile decoder [31]. Similar to the H.263 decoder, the MPEG-4 decoder has a VLD that decompresses the encoded input frames into a stream of macro blocks. In this decoder however, the frames are classified in I-frames and P-frames. Decoding of I and P-frames is partly different as they are encoded differently. I-frames are encoded independently from previous frames, but P-frames need a reference to a previous frame as they exploit temporal correlation in the form of motion vectors between subsequent frames. Therefore, the decomposed macro blocks from a P-frame are decoded and then made into a decoded frame using both MC (motion compensation) and RC (reconstruct). I-frame macro blocks lacking motion vectors are reconstructed using RC only. MPEG-4 is a dynamic application with behaviour that changes in non-deterministic patterns based on the input frame type and the number of motion vectors present. Observe that the structure of the application changes when decoding I and P-frames. For example, MC is not needed for I-frames while it is a part of the decoding process for P-frames.

Such dynamic applications, require a design that can recognize the dynamism and provide implementations that can deal with different conditions in a different way, while guaranteeing a robust performance. CSDF cannot express this type of dynamism. Scenario-Aware Dataflow (SADF) [32] is an extension of SDF and CSDF that can handle enough dynamism to model applications such as the MPEG-4 decoder, while still being amenable to analysis. In this model, scenarios represent modes of operation. For instance, MPEG-4 has naturally two modes of operation, namely I-frame and P-frame modes. In the P-frame mode, moreover, actor reading and writing rates and their execution times change between executions due to different numbers of motion vectors present in the data. In SADF, modes are described by (parameterized) SDF models. SADF provides additional semantics that captures parametric rates and execution times and switching between scenarios. In SADF scenario transitions can be defined deterministically, non-deterministically or even probabilistic.

1.2.5 Multi-Scale Behaviour

Applications may include tasks operating at multiple scales of data granularity. For example, an H.263 decoder has tasks acting at the frame level (RC), macro block level (VLD) and block level (IQ and IDCT). In general, dealing with such applications is challenging because their behaviour involves many task executions that correspond to lower level operations. For instance, the behaviour of the H.263 decoder involves 14256 executions of IQ and IDCT each, even for small frames of size 352×288 pixels. The H.263 decoder is an example of a multi-scale application for which the behaviour under analysis involves different sub-behaviours that occur at multiple granularity levels. This can be recognized in many other realistic examples, such as applications that include memory transactions with large blocks of memory that are communicated through a network-on-chip that operates at the level of much smaller individual flits [33].

Large numbers of task executions often cause scalability problems in the design process, for instance when estimating their performance, scheduling their tasks or optimizing the resource usage. Consider again the SDFG of an H.263 decoder shown in Figure 1.2. Observe that the minimum and maximum actor rates in the graph are a few orders of magnitude different from each other. This is often a sign that the graph is modelling a multi-scale behaviour. The existing throughput analysis techniques on the SDFG of the H.263 decoder need to execute the actor firings involved in at least one frame-level (coarsest level) behaviour of the H.263 decoder. For 352 × 288 frames, the frame-level behaviour includes more than 30000 actor firings. This potentially causes scalability issues for throughput analysis techniques.

Multi-scale applications cause serious scalability issues for resource optimization algorithms. For instance, a buffer-sizing approach on CSDF models of multi-scale image processing applications such as a multi-resolution convolution filter encounters scalability issues when applied to large image sizes [8]. This is partly due to the fact that the throughput analysis that is repeatedly performed while exploring the throughput buffer size space is adversely affected by the multi-scale nature of the application. Another orthogonal reason is that considering multiple granularity levels at the same time causes design space explosion, as the number of design choices at the finer levels grow very quickly.

An easy solution to such scalability problem is to consider a combination of the finer level behaviours in a multi-scale application as one coarse behaviour. For instance, in the H.263 decoder, we could consider the 14256 IDCT executions as one coarse execution of the IDCT task. However, this may result in very pessimistic assumptions and cause over-allocation of the resources. For instance, if we take this approach when finding the optimal buffer sizes for the H.263 decoder, we could only explore the buffer capacities that can hold integer multiples of 14256 macro blocks. This results in over-allocation of system memory, because allocating 14256 blocks does not allow for pipelining and 2 × 14256 is much more than needed.

1.2.6 Composite Behaviour

A common practice in the domain of streaming applications is to design complex applications by composing multiple pre-designed components. A complex image processing application that transforms the image sensor data to an image format in a digital camera, is a good example for this practice. The image data flows through a pipeline of various image filtering, enhancing and compression modules before it is stored in memory. In such a pipeline, every component uses the output of the previous component in the pipeline and produces output that is used as input in the next component in the pipeline. Such a structure creates an application with a composite behaviour that involves complicated data dependencies caused by the interaction between different components.

Understanding the timing behaviour of such an application and deriving an implementation that has a robust performance is hard due to complicated data dependencies. Usually each component is separately designed and verified for its performance. Then, the main challenge is to ensure that separately designed components can also deliver the performance in combination. This can be done by composing the component models in a recursive fashion to create a hierarchical model of the application for a performance analysis.

Compositionality of SDF models has been addressed in a number of works (e.g. [34] and [35]) by introducing the concept of hierarchical SDF graphs. A hierarchical SDF graph contains composite SDF actors. In a composite actor, an SDFG that receives tokens as input and produces tokens as outputs is abstracted by an ordinary (atomic) SDF actor. Such abstraction is not always compositional, i.e., it may lead to deadlock and/or rate inconsistencies [35]. Tripakis et al. [36] introduced a compositional abstraction for SDF actors, called deterministic SDF with shared FIFOs profiles. The introduced shared FIFO semantics allows for modular analysis, compilation and code generation for applications modelled as hierarchical SDF graphs. An approach to compose SADF models is recently addressed by Skelin et al. [37].

1.3 Problem Definition

As explained in Section 1.2, to tackle the challenges incurred in designing real-time streaming applications, the applications are designed based on a MoC. The model enables an iterative design process where the application is first represented at a necessary abstraction level. Then model elaborations iteratively refine the initial design and include the necessary details to implement the application such that it satisfies the required performance. The ability to efficiently construct models combined with associated analysis, verification and synthesis techniques makes model-based design a successful design method. Model-based design results in a complete solution by enabling concurrent implementation, automatic refinement and resource optimization [38].

SADF is a MoC that shows enough expressivity and analysability properties to be used in the design of real-time streaming applications. This model allows concurrent implementations to be verified for correctness, deadlock freedom and buffer boundedness in design time. An application that is designed based on an SADF model can also be verified for its timing properties. Moreover, it provides a means to express (non-deterministic) dynamism, while being able to guarantee functional correctness and performance robustness.

Applications with both dynamic and multi-scale behaviours are common in the embedded systems domain and they are often composed to form more complex applications. SADF can be used to express applications with such properties. However, the analysis techniques for SADF models are not fully developed. For instance, latency analysis is not worked out for this MoC, neither is a buffer sizing approach. Moreover, the existing throughput analysis for SADF and the method for composing them, do not scale for multi-scale applications. The lack of basic analysis methods prevents applications with such properties to be designed using a model-based approach.

This thesis addresses the problem of modelling and analysis of streaming applications with dynamic, multi-scale and composite behaviours. The modelling solution aims at finding a procedure to obtain SADF models for multi-scale applications and a method to compose them in a hierarchical manner to ease the procedure of modelling applications with composite behaviours. The analysis covers the most important performance metrics for streaming applications, i.e., the throughput and latency, besides obtaining throughput-buffering trade-offs. Since the analyses are repetitively used in model-based design methods, they should be fast enough such that the design methods terminate in a reasonable time. Moreover, they should be exact to avoid resource over-allocations.

1.4 Contributions

We make the following contributions in this thesis.

Contribution 1. *Modelling multi-scale applications by SADF*

We observe that the timing behaviour of multi-scale applications follows a periodic pattern of small (involving only a few task executions), static behaviours that is composed of a few behaviours associated with the coarser grain operations and many repetitions of behaviours associated with finer grain operations. We perceive each distinctive, small behaviour as a scenario. Consequently, the behaviour of the application is described as a periodic sequence of scenarios, some of which are repeated many times. Such behaviours can be modelled by an SADF. To compactly represent the repeated (sub-) sequences we introduce a representation for SADF, where the language of the sequences is represented as a regular expression with an explicit repetition construct instead of the traditional finite state machine. This modelling approach was first published in the following article.

H. Alizadeh Ara, A. Behrouzian, M. Hendriks, M. Geilen, D. Goswami, and T. Basten. Scalable analysis for multi-scale dataflow models. *Transactions on*

Embedded Computing Systems, 17(4), 2018.

Contribution 2. *Providing an approach to quickly generate the execution traces for SADF models of multi-scale applications*

The execution traces of applications can be used to understand the behaviour of the applications and find opportunities for performance improvement or detect anomalies. Re-using elements of the latency analysis, we provide a technique to quickly generate execution traces of a particular scenario within a pattern without performing a detailed simulation of the sequence that is followed leading up to that scenario. This is very helpful in case of multi-scale applications, as their periodic sequence is often very long. This trace generation method was published in the following paper.

H. Alizadeh Ara, A. Behrouzian, M. Geilen, M. Hendriks, D. Goswami, and T. Basten. Analysis and visualization of execution traces of dataflow applications. In *Proceedings of the 2nd Embedded computing and Architecture (IDEA) Workshop on Integrating Dataflow*, pages 19–20. ESR-2017-01, 2016.

Contribution 3. *Providing scalable throughput and latency analyses for multi-scale dataflow models*

For SADF with regular expression representation, we provide a compositional approach for exact, worst-case throughput and maximum latency analyses. The proposed throughput analysis shows an enhanced scalability for multi-scale dataflow models compared to the state of the art. Scalability improvement is achieved by the proposed regular expression representation to recognize the repeated patterns and the fact that the throughput analysis on repeated scenarios can be done in logarithmic time in the number of repetitions instead of linear time as in the state of the art methods. The proposed latency analysis, which is similarly scalable, is the first exact latency method for SADF models. The latency is recursively computed by decomposing the regular expression and computing the latency of sub-expressions in a bottom up approach. The throughput and latency analysis was published in the following article.

H. Alizadeh Ara, A. Behrouzian, M. Hendriks, M. Geilen, D. Goswami, and T. Basten. Scalable analysis for multi-scale dataflow models. *Transactions on Embedded Computing Systems*, 17(4), 2018

Contribution 4. *Composing dataflow models of cyclo-static applications*

A modular compositional model can facilitate the modelling process of complex applications by allowing individual component models to be composed in a recursive manner to generate a hierarchical model of a complex application. A complex application, such as a multi-resolution filter is structured as parallel pipelines of multi-scale components, each with a possibly different periodic pattern. Composing models with periodic patterns results in a model with a periodic pattern with a common hyper-period. This common periodic pattern is hard to derive by hand. We propose an efficient algorithmic method that given the periodic scenario sequences of the components by regular expressions, generates a sequence of composite scenarios in a compact regular expression representation. This composition approach was published in the following paper.

H. Alizadeh Ara, M. Geilen, A. Behrouzian, T. Basten, and D. Goswami. Compositional dataflow modelling for cyclo-static applications. In *Proceedings of the 21st Euromicro Conference on Digital System Design (DSD)*, pages 121–129. IEEE, 2018

Contribution 5. *Providing a throughput-buffering trade-off analysis for scenario-aware dataflow models*

As discussed in Section 1.2.3, the capacities of buffers in an application affect the throughput of the application. Since storage space is a scarce resource in computer systems, optimal points in the space of throughput-storage-space trade-offs should be found. We provide the first throughput-buffering trade-off analysis for applications modelled as SADF. We use our scalable throughput analysis in a guided design-space exploration to obtain the trade-offs. At every exploration step, the exploration prunes the exploration space without losing any optimal points. This is done by an approach that at every step identifies the buffers which do not affect the throughput anymore. The exploration on these buffers is stopped and, consequently, the analysis is terminated in a reasonably short time. This trade-off analysis method was published in the following paper.

H. Alizadeh Ara, M. Geilen, A. Behrouzian, and T. Basten. Throughput-buffering trade-off analysis for scenario-aware dataflow models. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 265–275. ACM, 2018

Contribution 6. *Obtaining tight timing bounds for dataflow applications mapped onto shared resources*

As mentioned in Section 1.2.3 budget scheduling is a common method to share the processors among several applications. To obtain the required processor budgets for every application, we need analysis techniques to estimate the performance of the application after the budget is allocated. Tighter estimations result in a better resource allocation. The final contribution of this thesis provides an analysis method to determine conservative but tight timing bounds for the scenarios of an SADF that is mapped onto a shared multiprocessor system. We obtain tighter bounds with respect to the state of the art by finding less conservative, but still guaranteed estimations on the response times of tasks in the application. This method of obtaining timing bounds was published in the following paper.

H. Alizadeh Ara, M. Geilen, T. Basten, A. Behrouzian, M. Hendriks, and D. Goswami. Tight temporal bounds for dataflow applications mapped onto shared resources. In *Proceedings of the 11th Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2016

1.5 Thesis Overview

This thesis is organized as follows. The next chapter provides the notations and preliminaries required to follow this thesis. Chapter 3 explains the modelling procedure for multi-scale applications. This chapters shows how a multi-scale application is modelled as an SADF, the language of which is represented by a regular expression. In this chapter, we also provide a means to quickly generate the execution traces of such SADF models. Chapter 4 provides the scalable throughput and latency analysis for regular expression-based SADF models of multi-scale applications. In Chapter 5, we propose an algorithmic method to compose the SADF models of cyclo-static multi-scale applications into an SADF with a compact regular expression, enabling scalable analyses of such applications. In Chapter 6 we discuss our throughput-buffering tradeoff analysis for SADF models. Chapter 7 provides tight timing bounds for scenarios of SADF models. Chapter 8 concludes the thesis.

Chapter 2

Preliminaries

In this chapter, we first introduce some notation and give the necessary background on the $(\max, +)$ algebra. We proceed with providing the details of the SDF model, elaborating on scenarios of SDF models and showing how scenario sequences can be represented by formal languages. Afterwards, we provide the definition of the SADF model, its throughput and latency. We explain the $(\max, +)$ characterization of SDF scenarios. This sets the stage for discussing the existing throughput analysis for SADF models. After that, we show how buffer requirements of applications are modelled in SADF models. Finally, we introduce open SADF models, as they are essential to model compositional dataflow behaviours.

2.1 Notations

Throughout the thesis the symbols $\mathbb{R}$, $\mathbb{R}^{\geq 0}$ and $\mathbb{N}$ denote the sets of real numbers, non-negative real numbers and natural numbers, respectively. Symbol $\mathbb{N}_0$ denotes the union of the natural numbers and the number 0 and, $\mathbb{R}_{-\infty}$ the union of the set of real numbers with $-\infty$. The set of all vectors with size n, the elements of which belong to $\mathbb{R}_{-\infty}$ is denoted by $\mathbb{R}^n_{-\infty}$. Symbol $\mathbb{R}^{n \times m}_{-\infty}$ denotes the set of all matrices with n rows and m columns with elements that belong to $\mathbb{R}_{-\infty}$. We use lower-case letters for scalars, bold upper-case letters for matrices and bold lower-case letters for vectors. A bar accent ($\bar{s}$) is used to denote finite or infinite sequences. Superscript $\cdot^T$ on vectors and matrices denotes the transposition operator.

2.2 (max,+) Algebra

$(\max,+)$ algebra [44, 45] supports two commutative and associative binary operators, namely *maximum* ($\oplus$) and *addition* ($\otimes$): $x \oplus y = \max\{x, y\}$ and $x \otimes y = x + y$, where $x, y \in \mathbb{R}_{-\infty}$. In $(\max,+)$ algebra, $x \oplus -\infty = -\infty \oplus x = x$ and, $x \otimes -\infty = -\infty \otimes x = -\infty$.

Let vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ belong to $\mathbb{R}^n_{-\infty}$ and, $\boldsymbol{A}$ and $\boldsymbol{B}$ be matrices that belong to $\mathbb{R}^{m\times n}_{-\infty}$. Let $[\boldsymbol{x}]_i$ and $[\boldsymbol{y}]_i$ denote the i^{th} element of $\boldsymbol{x}$ and $\boldsymbol{y}$ respectively. The *inner product* of $\boldsymbol{x}$ and $\boldsymbol{y}$ is defined as $\boldsymbol{x}^T\boldsymbol{y} = ([\boldsymbol{x}]_1 \otimes [\boldsymbol{y}]_1) \oplus ([\boldsymbol{x}]_2 \otimes [\boldsymbol{y}]_2) \oplus \cdots \oplus ([\boldsymbol{x}]_n \otimes [\boldsymbol{y}]_n)$. Operator $\oplus$, on vectors with the same size, operates elementwise. Matrix multiplications and additions are defined using the above inner product and addition of vectors, analogous to ordinary linear algebra. Let $\boldsymbol{C}$ be a matrix of an appropriate size. For readability, we write $\boldsymbol{Ax}$ and $\boldsymbol{AC}$ instead of $\boldsymbol{A} \otimes \boldsymbol{x}$ and $\boldsymbol{A} \otimes \boldsymbol{C}$.

Let $[\boldsymbol{A}]_{ij}$ and $[\boldsymbol{B}]_{ij}$ denote the element on row i, column j of their corresponding matrices. We write $\boldsymbol{A} \leq \boldsymbol{B}$ to denote that for every i and j, $[\boldsymbol{A}]_{ij} \leq [\boldsymbol{B}]_{ij}$ (considering that $-\infty \leq x$ for any $x \in \mathbb{R}_{-\infty}$). If $\boldsymbol{A} \leq \boldsymbol{B}$, it can be proven that $\boldsymbol{AC} \leq \boldsymbol{BC}$, $\boldsymbol{CA} \leq \boldsymbol{CB}$ and $\boldsymbol{A} \oplus \boldsymbol{C} \leq \boldsymbol{B} \oplus \boldsymbol{C}$, for any matrix $\boldsymbol{C}$. Subtraction of a scalar $c \in \mathbb{R}$ from a matrix $\boldsymbol{A}$, i.e. $\boldsymbol{A} - c$ or $\boldsymbol{A} \otimes (-c)$, is to subtract c from all elements of $\boldsymbol{A}$. The *norm* of a matrix $\|\boldsymbol{A}\|$ or a vector $\|\boldsymbol{x}\|$ is equal to the maximum entry in the matrix or the vector. $\mathcal{I}$ denotes an *identity matrix* in $(\max,+)$ algebra. For any matrix $\boldsymbol{A}$, $\mathcal{I}\boldsymbol{A} = \boldsymbol{A}\mathcal{I} = \boldsymbol{A}$. An identity matrix is square, the diagonal elements are 0 and the other elements are $-\infty$.

Consider a square matrix $\boldsymbol{M} \in \mathbb{R}^{n\times n}_{-\infty}$. The scalar e is an *eigenvalue* of $\boldsymbol{M}$, if there exists a vector $\boldsymbol{x} \in \mathbb{R}^n_{-\infty}$ with $\|\boldsymbol{x}\| > -\infty$ such that $\boldsymbol{Mx} = e\boldsymbol{x}$. The *adjacency graph* (also called the precedence graph [44]) of matrix $\boldsymbol{M}$ is a weighted directed graph with n nodes labelled $1, 2, \cdots, n$. For every entry $[\boldsymbol{M}]_{ij} \neq -\infty$, the adjacency graph contains an edge from node j to node i with the weight of $[\boldsymbol{M}]_{ij}$. The *cycle mean* of a cycle in this graph is defined as the sum of the edge weights of the cycle, divided by the number of edges in the cycle. The largest eigenvalue of a square matrix equals the Maximum Cycle Mean (MCM) among all cycles of its adjacency graph. *Critical cycles* of the adjacency graph are defined as the cycles with the maximum cycle mean.

Symbol $\bigoplus$ denotes the summation quantifier in $(\max,+)$ algebra. Summation over an empty set equals $-\infty$. Let $n \in \mathbb{N}_0$. $\boldsymbol{M}^n$ denotes raising $\boldsymbol{M}$ to the power of n. $\boldsymbol{M}^0 = \mathcal{I}$. The *star closure* of a square matrix $\boldsymbol{M}$ is a matrix $\boldsymbol{M}^*$ such that $\boldsymbol{M}^* = \bigoplus_{k\geq 0} \boldsymbol{M}^k$. $\boldsymbol{M}^*$ exists if and only if $\boldsymbol{M}$ has no positive eigenvalues.

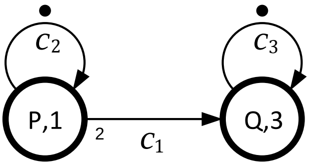


Figure 2.1: An example SDFG.

2.3 Synchronous Dataflow

Synchronous dataflow [20] describes an application by a directed graph. Figure 2.1 shows an example SDFG. In this graph, nodes depict *actors.* Actors represent individual tasks. In SDF, *actor firings* correspond to task executions. The letters inside the nodes are used to refer to actors. Directed edges in the graph represent *channels* and they are referenced by the annotations below them. Channels model dependencies be tween actor firings caused by, for instance, data dependencies or control decisions. For example, channel c_1 models a firing dependency of Q on P. The classical SDF is un-timed, i.e., actor firings do not take time.

To perform timing analysis we use a natural extension of SDF [46] where actor firings take some time, which is referred to as the *execution time.* We assume there exists a known upper bound for the execution time of every actor. This is a valid assumption in case the application is realized on a computer architecture that provides timing predictability, such as PRET [47] or CompSOC [48]. These architectures use predictable components such as predictable processor arbiters and predictable memory controllers. Consequently, they can ensure that within a certain time bound the actor will complete its execution. This upper bound is called the *worst-case execution time* of the actor. In Figure 2.1, the numbers inside the nodes show the worst-case execution times of the actors. For all examples in the thesis, we assume that the given execution times are the worst-case execution times.

The dependencies between actor firings are captured by a mechanism that allows actor firings to *produce* and *consume* some entities on the channels called *tokens.* Channels may initially contain tokens, referred to as *initial tokens.* In Figure 2.1, initial tokens are shown by black dots on top of the channels. When an actor starts firing, it consumes tokens from its input channels, and when its execution time expires, it produces tokens on its output channels. All firings of an actor consume and produce constant numbers of tokens per channel,

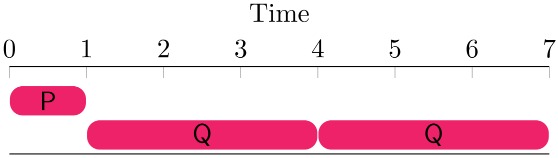


Figure 2.2: An execution trace of the example SDFG.

called the consumption and production *rates*. The rates are given as edge annotations in the graph (rates of 1 are not shown to avoid cluttering). For instance, every firing of actor P consumes one token from channel c_2 and after one time unit, it produces one token on channel c_2 and two tokens on channel c_1. An actor can start firing only if it is *enabled*. In an enabled actor, the token numbers on all input channels are equal to or greater than the consumption rates of the actor on the corresponding channels.

An *execution* of an SDFG is a sequence of actor firings which may overlap in time. An SDFG execution models an execution of the application. In a *self-timed* execution, actors start firing as soon as they are enabled. The timing behaviour of an SDFG can be studied by simulating an execution that corresponds to at least one full execution cycle of the application [49]. During the simulation, tokens are assigned *time-stamps* that show their availability times. These time-stamps can be stored and used for timing analysis after the simulation.

Consider Figure 2.1 with initial tokens that are available from time 0. These tokens are assigned the time-stamp 0. We illustrate an execution that involves one firing of P and two firings of Q. Figure 2.2 shows a trace of this execution. P is enabled at time 0, because it needs only the initial token available on c_2, but Q is not enabled because it needs at least one token on c_1, which is not there. In a self-timed execution, at time 0, P consumes the token on channel c_2 and starts firing. This firing ends at time 1 and produces a token on c_2 and two tokens on c_1, all with time-stamp 1. Now Q is enabled. The first firing of Q consumes a token produced by the firing of P on c_1 and the initial token on c_3. This firing starts at time 1, i.e., immediately after P produces tokens on c_1. When the first firing of Q ends after three time units, it produces a token on channel c_3 with time-stamp 4. This token, together with the token remaining on channel c_1, are consumed by the second firing of Q at time 4. At time 7, the second firing of Q ends and leaves a token on channel c_3 with time-stamp 7. In our particular example, actors fired one after another because of the token dependencies. However, in general, in a self-timed execution, multiple actor firings of the same actor and/or different actors can happen at the same time. For instance, if we allow another execution of P, it would start at time 1, immediately after the first firing of P completes and produces a token on c_2. This firing would be in parallel with the first firing of Q.

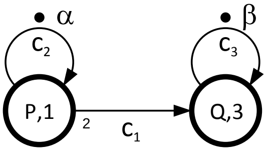
Observe that in Figure 2.1, after the completion of the second firing of Q, the number of tokens on the channels is the same as before the execution. An *iteration* of an SDFG is the smallest, non-empty set of actor firings that does not change the number of tokens on any of the channels. Performance metrics for an SDFG are often defined per iteration. For instance, the throughput of an SDFG is defined as the average number of iterations executed per time unit in a self-timed execution [49, 50].

An SDFG is formally defined by a tuple (A, C). The set A contains a finite number of actors. An SDFG defines an execution time $e(a) \in \mathbb{R}^{\geq 0}$ for every actor $a \in A$. The set $C \subseteq A \times A$ is the set of channels. Let $c \in C$ be a channel from an actor $a_1 \in A$ to an actor $a_2 \in A$ (a_1 and a_2 can be the same actor). Channel c is an output channel of a_1 and an input channel of a_2. Each of actors a_1 and a_2 associate a number from $\mathbb{N}_0$ to channel c, namely the output rate $Out_{a_1}(c)$ and input rate $In_{a_2}(c)$, respectively. Channel c contains a non-negative number $It(c)$ of initial tokens.

Since in SDF the rates and execution times of actors are fixed, this model can be a good abstraction only for applications with a relatively static execution behaviour. SDF abstractions of dynamic applications may become pessimistic, because they accumulate the worst-case rates and execution times in all varying behaviours in one graph, which may not be realistic. In the next section we discuss an extension to SDF that refines the behaviour of a dynamic application into several SDFGs and provides a less conservative abstraction.

2.4 Synchronous Dataflow Scenarios

Scenarios are a means to model dynamism in the behaviour of applications [51]. They describe different, deterministic modes of operation in an application. An application may continue on one mode or switch to another in a non-deterministic manner. Recall the MPEG-4 decoder application discussed in Section 1.2.4. In this application, the decoding of I-frames and P-frames is done differently, resulting in two different modes of operation. Each mode of operation executes a set of tasks. A task may execute in both modes (e.g. VLD), possibly with different data dependencies and execution times, or may not execute in one mode or another. For example, MC actor does not execute for an I-frame scenario.



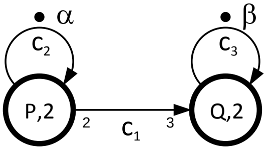
Reps: $[\mathsf{P},\mathsf{Q}] \mapsto [1,2]$

ITs: $c_2(\alpha), c_3(\beta)$

FTs: $c_2(\alpha), c_3(\beta)$

Reward: 2 (firings of Q)

Figure 2.3: Scenario ϕ.



Reps: $[\mathsf{P},\mathsf{Q}] \mapsto [3,2]$

ITs: $c_2(\alpha), c_3(\beta)$

FTs: $c_2(\alpha), c_3(\beta)$

Reward: 2 (firings of Q)

Figure 2.4: Scenario ψ.

As the MPEG-4 application is quite complicated for our illustration purposes, we proceed with a simpler example. Let's consider an example application that involves two tasks with dynamic behaviours, described by two SDF scenarios, namely ϕ and ψ. An SDF scenario specifies a finite non-empty set of actor firings in an SDFG. Scenarios are represented by a *repetition vector* that shows the exact number of times (repetitions) that each actor in the graph fires. Figure 2.3 shows an example SDF scenario ϕ. The table on the right side of this figure shows the repetition vector for the scenario. This scenario includes one execution of P and two executions of Q, which we denote as follows $[\mathsf{P},\mathsf{Q}] \mapsto [1,2]$. Scenario ϕ corresponds to one iteration of its SDFG. In general, a scenario may correspond to a partial iteration or also more than one iteration. For instance, it is possible to define a scenario that includes only one execution of P, i.e. $[\mathsf{P},\mathsf{Q}] \mapsto [1,0]$.

Executing a scenario is to execute all actor firings defined by the scenario. A scenario execution leaves a number of tokens on the channels, called *final tokens*. Initial and final tokens in a scenario are labelled. In Figure 2.3, the initial tokens are labelled with annotations next to them. For instance, α labels the initial token on channel c_2. In the table of Figure 2.3, Initial Tokens (ITs) and Final Tokens (FTs) show the locations and labels of all initial and final tokens. For instance, $c_3(\beta)$ in the row that corresponds to FTs means that the final token that will appear on channel c_2 after an execution is labelled β. ITs and IFs are a means to convey (timing and dependency) information from an executed scenario to the scenario that will be executed next. For

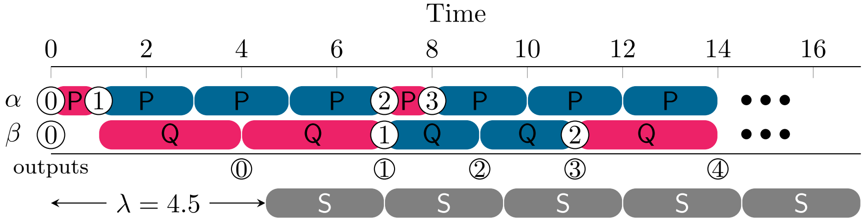


Figure 2.5: The execution trace of the scenario sequence $(\phi\psi)^\omega$.

instance, assume scenario ψ (shown in Figure 2.4) will execute after scenario ϕ. The final tokens in ϕ take the place of the initial tokens with the same labels in ψ. If the ITs and IFs in a transition from scenario a to scenario b do not match, it is said that the transition is *inconsistent*, and consequently, the model is invalid. This also holds for self-transitions, in which the scenario does not change. ITs and IFs should match within the scenario for which a self-transition is possible.

Consider an infinite scenario sequence in which ψ and ϕ execute alternatingly, starting with ϕ. Expression $(\phi\psi)^\omega$ represents this sequence (ω denotes indefinite repetition). Figure 2.5 shows the execution trace of actors for this scenario sequence. The initial tokens in ϕ are shown with nodes labelled 0 along the lanes annotated with α and β on the left. We assume the initial tokens in ϕ are available from time 0. After the execution of ϕ, the final tokens α and β are produced at times 1 and 7, respectively (nodes labelled with 1). These tokens are consumed in ψ as initial tokens, and the final tokens α and β in ψ are produced at times 7 and 11, respectively. Note that even though the scenarios are executed sequentially, their actors are executed in parallel.

The time-stamps of the initial tokens in scenarios of a scenario sequence carry a notion of *state* for the timing behaviour of the application. In fact these time-stamps summarize the timing behaviour from the starting point, i.e., from the time the application starts executing, and they are the only timing information needed from the past to proceed with the execution of the upcoming scenarios. The time-stamps of the initial tokens are often collected in a column vector called the *state vector* (also called schedule [31]). We denote the state vector by $\boldsymbol{\gamma}$. Every time a scenario executes, the state vector changes from an initial state to a final state. Let's consider $\boldsymbol{\gamma}_i$ ($i > 0$) as the state vector after the execution of the i^{th} scenario in the sequence and $\boldsymbol{\gamma}_0$ as the initial state vector. For the execution in Figure 2.5 we have $\boldsymbol{\gamma}_0 = [\ 0 \quad 0\ ]^T$, $\boldsymbol{\gamma}_1 = [\ 1 \quad 7\ ]^T$ and $\boldsymbol{\gamma}_2 = [\ 7 \quad 11\ ]^T$. The elements in the first and the second row of the state vector show the time-stamps of α and β, respectively. The state vector can be analyzed for performance. For instance, the growth rate of the state vector during the execution of a sequence determines the throughput of the application.

A scenario execution in the model, corresponds to a certain amount of real-world progress. The progress can be defined depending on what we are interested to analyse. For instance, in the MPEG-4 decoder application, if we are interested to analyse the number of decoded frames per time unit, the unit of progress can be defined as the decoding of one frame. To imply the amount of progress earned in executing a scenario, we assign scenarios with real-valued *rewards*. In the MPEG-4 decoder model provided by Geilen et al. [31], both I and P-frame scenarios have a reward of 1, as the actor firings included in those scenarios correspond to the task executions required for decoding one frame. For some scenarios, we may be interested in the number of firings of a specific actor, because that actor produces *output* every time it fires. In such a case, the repetition of that actor in the scenario can serve as the reward of the scenario. For instance in scenarios ϕ and ψ, we are interested in the number of firings of actor Q. Therefore, these scenarios both have the reward of 2. In Figure 2.5, the outputs produced by the firings of actor Q are shown by the nodes along the outputs lane. The outputs are numbered for referencing.

In some applications we may want to specify a restriction of the scenario transitions that are allowed to take place, to exclude inconsistent transitions or any other unrealistic behaviours. For instance, in the MPEG-4 decoder we may want to exclude sequences of frame types cannot occur because of encoding restrictions, such as any sequence starting with a P-frame type (P-frames need a reference to the previous frame which is non-existent if the P-frame is the first frame). This leaves us with a set of possible scenario sequences the application may follow. At the same time, we may allow for non-deterministic scenario transitions, leading to an infinite number of possible scenario sequences. For instance, in the MPEG-4 decoder, we may allow non-deterministic transitions between I and P-frame types. The scenario sequences are infinitely long when used to model streaming applications such as the MPEG-4 decoder, because the stream of frames is unbounded. Therefore, in general we deal with infinite sets of infinite scenario sequences that need to be analysed for the worst-case behaviour to guarantee a minimum performance.

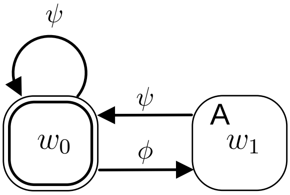


Figure 2.6: An example FSM.

2.5 Regular Languages

In the theory of languages [52], an *alphabet* is a finite, non-empty set of letters. A sequence of letters makes a *word*. The *empty word* does not have any letters. Words can be of finite or infinite length. A set of words forms a *language*. The language that does not contain any words is called the *empty language*. A language that can be represented as a finite state automaton is called a *regular language*. We use regular languages to describe finite or infinite sets of infinite scenario sequences. The letters correspond to scenarios and the words correspond to scenario sequences. We use the following two ways to represent a regular language.

2.5.1 Finite State Machines

A Finite State Machine (FSM), also called finite state automaton, is a tuple (W, w_0, η), with a set W of states, an initial state $w_0 \in W$ and a labelled transition relation $\eta \subseteq W \times S \times W$, where S is an alphabet, in our case, the set of letters that represent scenarios. Figure 2.6 shows a graphical representation of an FSM. This graph has two nodes, representing two states, namely states w_0 and w_1. Directed edges show state transitions of η. State transitions are labelled with letters. An *infinite run* in an FSM is an infinite sequence of transitions, starting from the initial state w_0. The initial state is shown with an extra ring in the node. An infinite run of an FSM corresponds to an infinite word that the transitions followed by the run are labelled with. For instance, a run that corresponds to taking the transition from w_0 to w_1 and back to w_0 alternatingly, forms the word $(\phi\psi)^\omega$.

An infinite word is said to be *recognized* by an FSM, iff there exists an infinite run in the FSM that can form that word. FSMs may have *acceptance conditions* in the form of a set $W_a \subseteq W$ of accepting states. Acceptance conditions, when present, place additional constraints on which words are recognized. The acceptance condition applied in this thesis is the one from Büchi Automata [53]. We say that a run of an FSM is *accepted* if some accepting state is visited along the run infinitely often. For instance, assume in Figure 2.6, w_1 is the only accepting state. The accepting state is labelled with letter A. The run that forms the word ψ^ω is not accepted as it never visits w_1, whereas the run that forms the word $(\phi\psi)^\omega$ is accepted, because it visits w_1 infinitely often. FSMs in the thesis have no acceptance conditions unless mentioned otherwise. If an FSM has no acceptance conditions, then all infinite runs recognized by the FSM are accepted. An FSM represents or recognizes the language that includes all words accepted by the FSM. We use $\mathscr{L}(M)$ to denote the language represented by an FSM M. As commonly done in non-deterministic FSMs, for convenience, ϵ is used to label empty transitions. Empty transitions change the state without adding a letter to words. An ϵ-free FSM does not have any ϵ transitions.

2.5.2 Regular Expressions

A *regular expression* is an algebraic formula that can represent a regular language. We first introduce regular expressions that represent sets of finite words, as a means to define regular expressions that represent sets of infinite words. We define a regular expression ρ of finite words over a finite alphabet S by the following syntax.

$$\rho ::= \emptyset \mid \epsilon \mid s \mid \rho_1\rho_2 \mid \rho_1 + \rho_2 \mid \rho^n \mid \rho^*$$

where $s \in S$, and $n \in \mathbb{N}$ is a natural number. A regular expression ρ can be the empty expression $\emptyset$ which represents the empty language, the empty string ϵ, a single letter $s \in S$, a sequential composition (also called concatenation) of two regular expressions, a choice between two regular expressions, a regular expression that sequentially composes ρ with itself for n times (e.g. $\rho^2 = \rho\rho$), and a Kleene iteration $*$ of a regular expression ρ, which is defined as $\rho^* = \epsilon + \rho + \rho^2 + \rho^3 + \cdots$. Although it does not add expressiveness, we add the syntax ρ^n to the commonly used syntax to be able to compactly represent the sequential composition of n times ρ. As an example, $\phi\psi^{10}$ represents the language that includes only the finite word that consists of letter ϕ followed by ten repetitions of the letter ψ. $\mathscr{L}(\rho)$ is used to denote the language defined by ρ.

ω-regular expressions formalize regular languages that describe sets of infinite words. We use the following syntax for ω-regular expressions σ, where ρ

is a regular expression describing a language of finite words.

$$\sigma ::= \rho^{\omega} \mid \rho\sigma \mid \sigma_1 + \sigma_2$$

The expression ρ^{ω} defines a regular language by indefinite concatenation of finite words from the language of a regular expression ρ. A concatenation $\rho\sigma$ denotes the sequential composition of a regular expression ρ and an ω-regular expression σ. Finally, + denotes a choice between two ω-regular expressions. We use $\mathscr{L}(\sigma)$ to denote the language recognized by σ. A language can be represented by both an ω-regular expression and an FSM. We say that an ω-regular expression and an FSM are *equivalent* if they represent the same language [54]. ω-regular expressions and FSMs with Büchi acceptance conditions define the same class of languages.

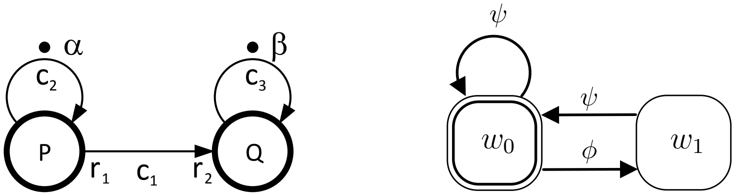
We use the term regular expressions generally if it is clear from the context that we are dealing with finite or infinite words. Otherwise, we use the term ordinary regular expressions with regular expressions that describe languages of finite words and ω-regular expressions with regular expressions that describe languages of infinite words.

2.6 Scenario-Aware Dataflow

Scenario-aware dataflow [32] is a dataflow model composed of a set of SDF scenarios, and a set of possible scenario sequences. Figure 2.7 shows an SADF with two scenarios, namely ϕ and ψ, and a set of possible scenario sequences represented by an FSM. Figure 2.7a shows the scenario graphs of these scenarios and Table 2.7c shows their specifications. The structure of the scenario graphs are the same, but the actor execution times and some rates are different for every graph.

In this section, we provide formal definitions for SADF and its two important performance metrics: the throughput and latency. The definition of SADF is adapted from the definition provided by Geilen et al. [55]. They use an FSM representation of the language of scenario sequences in their SADF definition. We use the language itself instead of a specific representation of it, since we use different representations of the language, namely FSMs and regular expressions.

An SADF is defined by a tuple (S, g, r, i, f, o, L). The set S contains a finite number of scenarios. Every scenario $s \in S$ has an associated SDFG $g(s)$. The function $r(s)$ maps every actor of $g(s)$ to a non-negative number that corresponds to its repetition count. The graph $g(s)$ has a number $i(s) \in \mathbb{N}$ of labelled initial tokens distributed over its edges. After scenario s executes, it



(a) Scenario graphs. (b) FSM.

Scenarios	ϕ	ψ
ETs	$e(P) = 1, e(Q) = 3$	$e(P) = 2, e(Q) = 2$
Rates	$r_1 = 2, r_2 = 1$	$r_1 = 2, r_2 = 3$
Reward	2 (firings of Q)	2 (firings of Q)
Reps	$[\mathsf{P}, \mathsf{Q}] \mapsto [1, 2]$	$[\mathsf{P}, \mathsf{Q}] \mapsto [3, 2]$
ITs	$c_2(\alpha), c_3(\beta)$	$c_2(\alpha), c_3(\beta)$
FTs	$c_2(\alpha), c_3(\beta)$	$c_2(\alpha), c_3(\beta)$

(c) Scenario specifications.

Figure 2.7: An SADF with two scenarios.

leaves a number $f(s) \in \mathbb{N}$ of labelled final tokens on some edges of $g(s)$. Note that scenario specifications such as the label and location of the initial and final tokens are assumed to be known and captured in g for every scenario $s \in S$. We made the maps $i(s)$ and $f(s)$ explicit in the definition to help the readability in developing formulas in the remainder of the thesis. The function $o(s)$ maps every scenario to its reward, which is the number of outputs produced by an actor specified in the scenario.

The language L describes a set of infinite scenario sequences. When the language of an SADF is described by an FSM we refer to it as a Finite-State-Machine Scenario-Aware Dataflow (FSM-SADF). For instance, the SADF shown in Figure 2.7 is an FSM-SADF. The FSM-SADF introduced by Geilen et al. [55] does not consider acceptance conditions for the FSM. Our definition, which is based on regular languages, allows more fine-grained specifications of the set of scenario sequences than the FSM-based definition of Geilen et al. [55], because some regular languages can only be translated to FSMs that have acceptance conditions.

The throughput can be defined for a given infinite scenario sequence of an

SADF. Let $\bar{s} = s_1 s_2 s_3 \cdots$ denote a sequence of scenarios s_n. We quantify the throughput of a given scenario sequence of an SADF by the average number of outputs produced per time unit during the execution of that sequence as follows.

$$Thr(\bar{s}) = \liminf_{i \to \infty} \frac{\Sigma_{n=1}^{i} o(s_n)}{\|\boldsymbol{\gamma}_i\|} \tag{2.1}$$

In this equation, the denominator of the fraction represents the progress of time, measured as the norm of the state vector $\boldsymbol{\gamma}_i$, and the nominator is the total number of outputs produced until and including scenario s_i. Taking the average over an infinite sequence $\bar{a}$ of numbers is typically defined using $\lim_{n\to\infty} a_n$. Such limit does not necessarily exist for the sequence $\bar{a}_i = \Sigma_{n=1}^{i} o(s_n)/\,\|\boldsymbol{\gamma}_i\|$, since the sequence of scenarios can be such that $\bar{a}$ always fluctuates between infimum and supremum values. We use the operator $\liminf_{i\to\infty}$ to be conservative. Using the definition above, the throughput of the sequence $(\phi\psi)^\omega$ is 0.4. This means actor Q fires and produces an output 4 times every 10 time units on average. This can be observed from the execution trace in Figure 2.5. The firings of actor Q during the period 1 to 11 exactly repeat during the period 11 to 21 and so on, and during every period 4 outputs are produced). The worst-case throughput of an SADF F is obtained among the set of all possible sequences represented by the language L, as follows.

$$Thr(F) = \min_{\bar{s} \in L} Thr(\bar{s}) \tag{2.2}$$

In Section 2.8 we discuss a throughput analysis based on this definition, provided by Geilen et al. [55]. The throughput of the example shown in Figure 2.7 is 2/6, since the sequence ψ^ω which is recognized by the FSM shown in the figure, turns out to be a sequence with the minimum throughput among all other recognized sequences. Three firings of actor P takes 6 time units in every execution of ψ, during which 2 outputs are produced.

Eq. 2.2 requires that $Thr(\bar{s})$ is the worst-case throughput of scenario sequence $\bar{s}$, considering that the actual execution time of actors may be less that their worst-case execution time. Geilen et al. [55] and Wiggers et al. [24] show that in a self-timed execution, if a dataflow actor executes in less time than the worst case, then the throughput will be no worse than if it had used its entire worst-case execution time. This property is known as the *monotonicity* property of dataflow models. In Chapter 6, we use the monotonicity property of SADF models to provide an efficient approach for the SADF buffer sizing problem. Monotonicity of a particular implementation of an application with bounded resources depends on the implemented resource arbitration policy. For instance, actor executions on a processor with TDMA arbitration is monotonic, whereas under round-robin arbitration it is not monotonic.

Next we define the latency. The outputs produced by an steaming application are often buffered and used in a periodic fashion. For instance, the frames decoded by an H.263 decoder are displayed to the user in a certain frame rate. We are often interested to know the minimum waiting time before the outputs produced by an application can be used with a certain rate. To define the latency of a scenario sequence, we that imagine the outputs that are produced during the execution of a scenario sequence are consumed by an external actor S that is *periodically scheduled* with period μ. Every firing of this actor consumes one output. For the external actor to have a feasible schedule, an output should be produced no later than the start time of the external actor for all firings. If the throughput of the application is lower than $1/\mu$, such a periodic schedule will be eventually disrupted, as the outputs will be produced later than the start times of actor S. If the throughput is not lower than $1/\mu$, there exists a feasible periodic schedule for the external actor. That is, there exists a waiting time λ such that for all k it holds that $p_k \leq \lambda + k\mu$, where p_k denotes the time of k^{th} output produced during the scenario sequence. Figure 2.5 shows such a periodic schedule for an actor S that consumes the outputs produced during the sequence $(\phi\psi)^\omega$. In this figure, the waiting time λ is 4.5 and the period of the tasks is equal to $1/0.4 = 2.5$. Note that a shorter waiting time is not feasible in this example, since the second firing of S would then not find its output token in time.

We define the latency of a scenario sequence relative to a given period μ as the shortest waiting time of the periodically scheduled external actor that consumes outputs with period μ. This means the latency is defined as the smallest λ such that $p_k \leq \lambda + k\mu$ holds for all k. Given an initial state $\boldsymbol{\gamma}_0$, we can determine the production times p_k of all outputs in a scenario sequence. The latency of a scenario sequence $\bar{s}$ with an initial state vector $\boldsymbol{\gamma}_0$, relative to period μ is defined as follows.

$$\lambda(\bar{s}, \boldsymbol{\gamma}_0, \mu) = \max_{k \geq 0} p_k - \mu k \tag{2.3}$$

According to Figure 2.5, given initial state $\boldsymbol{\gamma}_0 = [\ 0 \quad 0\ ]^T$, the latency of $(\phi\psi)^\omega$ relative to period 2.5 is 4.5. Considering the production times of outputs and the firing start times of the external actor S, the first start time of S cannot happen earlier than 4.5. Since the second firing of S requires output number one which is produced at 7 the earliest.

Similar to the throughput case, the worst-case latency over the language

L is defined as follows.

$$\lambda(L, \gamma_0, \mu) = \max_{\bar{s} \in L} \lambda(\bar{s}, \gamma_0, \mu) \tag{2.4}$$

Geilen et al. [31] provide a similar latency definition except that according to their definition, there is no notion of outputs produced by actor firings, and instead, the execution of any scenario is regarded as production of one output.

2.7 (max,+) Characterization of SDF Scenarios

The timing behaviour of any actor firing in an SDF scenario can be characterized by the timing relation between the time-stamps of the tokens produced by the actor firing, and the time-stamps of the initial tokens. This relation is naturally captured by maximization and addition operations. For instance, consider the firing of actor P in scenario ϕ. Let *symbols* t_α and t_β respectively denote the time-stamps of the initial tokens α and β, and t denote the time-stamps of the tokens produced by the firing of P. According to the scenario graph, in a self-timed execution, P starts firing as soon as α becomes available. This firing completes and produces tokens, 1 time unit after the start. Therefore, $t = t_\alpha + 1$. Alternatively, we could write this relation in $(\max, +)$ algebra as a linear equation $t = (1 \otimes t_\alpha) \oplus (-\infty \otimes t_\beta)$. The expression $-\infty \otimes t_\beta = -\infty$ implies the fact that t is not affected by t_β. This way of representing the time-stamp of a token, i.e. with a linear $(\max, +)$ equation in terms of variables t_α and t_β, is called the *symbolic time-stamp* representation.

Symbolic time-stamp t can be represented also by a vector inner product $t = \boldsymbol{g}^T \boldsymbol{t}$, where $\boldsymbol{g}$ contains suitable constants and $\boldsymbol{t}$ contains the symbols that denote the time-stamps of the initial tokens. In this equation, $\boldsymbol{g}^T$ is referred to as the *symbolic time-stamp vector* and $\boldsymbol{t}$ is referred to as the *symbolic state vector*. For instance, the symbolic time-stamp of tokens produced by the firing of P can be represented as $t = [\ 1 \quad -\infty\ ][\ t_\alpha \quad t_\beta\ ]^T$, where $[\ 1 \quad -\infty\ ]$ is the symbolic time-stamp vector assigned to all tokens produced by the firing of P, and $[\ t_\alpha \quad t_\beta\ ]^T$ is the symbolic state vector. Note that the symbolic state-vector is the same in the symbolic representation of any token in a scenario, and it does not carry any timing information. The essential timing information of every token is captured by its symbolic time-stamp vector.

This section shows how scenario executions can be characterized by $(\max, +)$ algebra equations using the symbolic time-stamp representation. Every scenario is characterized by two linear matrix equations, namely the state equation and the output equation (similar to state-space equations in linear system theory [56]). In a scenario sequence, the state equations of the scenarios determine the state vector after the execution of the scenario and the output equations of the scenarios relate the production times of outputs to the state vector. In the remainder of this section we discuss these two equations and show how they can be obtained for a scenario.

2.7.1 The State Equation

The relation between the time-stamps of the final tokens and the time-stamps of the initial tokens in the execution of a scenario can be represented by a linear matrix equation in $(\max, +)$ algebra. For a scenario s, if we collect the time-stamps of the initial tokens in vector $\boldsymbol{t}$ and the time-stamps of the final tokens in vector $\boldsymbol{t}'$, this relation can be expressed as the following equation [31].

$$\boldsymbol{t}' = \boldsymbol{G}_s \boldsymbol{t} \tag{2.5}$$

Eq. 2.5 is called the *state equation.* We refer to $\boldsymbol{G}_s$ as the *state matrix* (also called scenario matrix) of s. For instance, in scenario ϕ and ψ, we can collect the time-stamps of the initial tokens in vector $\boldsymbol{t} = [\ t_\alpha \quad t_\beta\ ]^T$ and the time-stamps of the final tokens in vector $\boldsymbol{t}' = [\ t'_\alpha \quad t'_\beta\ ]^T$ and describe scenarios ϕ and ψ by the following matrices.

$$\boldsymbol{G}_\phi = \begin{bmatrix} 1 & -\infty \\ 7 & 6 \end{bmatrix}, \quad \boldsymbol{G}_\psi = \begin{bmatrix} 6 & -\infty \\ 8 & 4 \end{bmatrix}$$

An entry t at column k and row m in $\boldsymbol{G}_s$ specifies that there is *at least* a time difference of t time units between the time-stamp of token k before the execution of s and the time-stamp of token m after s is executed. For instance the entry 7 on the first column of the second row of $\boldsymbol{G}_\phi$ implies that the time-stamp of final token β is at least 7 time units later than the time-stamp of the initial token α. This is a valid statement since the dependency path from α to β goes through one firing of P (with execution time of 1) and two firings of Q (with execution time of 3); therefore there is at least $1 + 2 \times 3 = 7$ time units difference between the availability of the initial token α and the production of the final token β.

In a scenario sequence, we can use the state matrices of the scenarios to determine the state vector of any scenario in the sequence. For instance, consider the sequence $(\phi\psi)^\omega$. Given $\boldsymbol{\gamma}_0 = [\ 0 \quad 0\ ]^T$, we can find the state vector $\boldsymbol{\gamma}_1$ and $\boldsymbol{\gamma}_2$ as follows.

$$\boldsymbol{\gamma}_1 = \boldsymbol{G}_{\phi}\boldsymbol{\gamma}_0 = \begin{bmatrix} 1 & -\infty \\ 7 & 6 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \otimes 0 \oplus -\infty \otimes 0 \\ 7 \otimes 0 \oplus 6 \otimes 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 7 \end{bmatrix}$$

$$\boldsymbol{\gamma}_2 = \boldsymbol{G}_{\psi}\boldsymbol{\gamma}_1 = \begin{bmatrix} 6 & -\infty \\ 8 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 7 \end{bmatrix} = \begin{bmatrix} 6 \otimes 1 \oplus -\infty \otimes 7 \\ 8 \otimes 1 \oplus 4 \otimes 7 \end{bmatrix} = \begin{bmatrix} 7 \\ 11 \end{bmatrix}$$

In general, for any sequence $\bar{s} = s_1 s_2 \cdots$, we can compute the state vector $\boldsymbol{\gamma}_i$ as follows.

$$\boldsymbol{\gamma}_i = \boldsymbol{G}_{s_i}\boldsymbol{\gamma}_{i-1} \tag{2.6}$$

A systematic way of obtaining the state matrix of a scenario can be found in a work of Geilen [31]. The author uses a *symbolic simulation* of the scenario (as opposed to the concrete simulation in Section 2.3) to obtain the state matrix. In the symbolic simulation, tokens are represented by their symbolic time-stamps using the symbolic time-stamp vectors. We explain how this matrix can be obtained for scenario ϕ using the symbolic simulation. Consider the symbolic state vector $\boldsymbol{t} = [\ t_\alpha \quad t_\beta\ ]^T$. Using this symbolic state vector, the time-stamp of the initial tokens α and β can be represented by time-stamp vectors $[\ 0 \quad -\infty\ ]$ and $[\ -\infty \quad 0\ ]$, respectively (e.g. $t_\alpha = [\ 0 \quad -\infty\ ][\ t_\alpha \quad t_\beta\ ]^T$). After P completes firing, the two tokens produced on channel c_1 and the final token α are assigned the time-stamp vector $[\ 0 \quad -\infty\ ] + 1 = [\ 1 \quad -\infty\ ]$, as they are produced 1 time unit after the initial token α becomes available. Observe explicitly that addition to a symbolic time-stamp is represented by the addition to the symbolic time-stamp vector. The first firing of Q consumes a token from c_1 and initial token β, and starts at $[\ 1 \quad -\infty\ ] \oplus [\ -\infty \quad 0\ ] = [\ 1 \quad 0\ ]$. Observe explicitly that max of symbolic time-stamps is represented by the max of the symbolic time-stamp vectors. It completes and produces a token on channel c_3 with time-stamp $[\ 1 \quad 0\ ] + 3 = [\ 4 \quad 3\ ]$. The second firing of Q, consumes another token from c_1 and the token on channel c_3, and produces the final token β with time-stamp $([\ 1 \quad -\infty\ ] \oplus [\ 4 \quad 3\ ]) + 3 = [\ 4 \quad 3\ ] + 3 = [\ 7 \quad 6\ ]$. If we collect the time-stamp vectors of the final tokens α and β, i.e., $[\ 1 \quad -\infty\ ]$ and $[\ 7 \quad 6\ ]$, in the first and second row of a matrix, respectively, we obtain $\boldsymbol{G}_\phi$.

It is worthwhile to mention that the state matrix can also be defined for SDF and CSDF models. The state matrix of an SDF or a CSDF captures the essential timing information in one iteration of the graph. This matrix can be directly used for timing analysis. For instance, the eigenvalue of the state matrix of an SDF or a CSDF shows the average time required to execute an iteration. This time is called the *cycle time*, which equals the reciprocal of the throughput of the graph.

2.7.2 The Output Equation

In SDF scenarios we may need to capture the production times of outputs, for instance to compute the latency. The output production times might be directly accessible through the state vectors. For example in Figure 2.5 it is observed that the production of every other output is synchronized with the production of the final token β, i.e. it is reflected in the second element of the state vector. However, this is not always the case. In general, given a time-stamp vector $\boldsymbol{t}$ of the initial tokens, the output production times in a scenario s can be computed in a vector $\boldsymbol{p}$ using the following matrix equation (similar to the one introduced by Skelin et al. [57]).

$$\boldsymbol{p} = \boldsymbol{H}_s \boldsymbol{t} \tag{2.7}$$

Eq. 2.7 is called the *output equation.* We refer to $\boldsymbol{H}_s$ as the *output matrix* of the scenario s. It has as many rows as there are outputs produced in scenario s. Each row of this matrix expresses the relation between the time-stamps of the initial tokens and the production time of an output in s as the inner product of each row and the time-stamp vector of initial tokens. We assume the rows are ordered such that the first row corresponds to the first output (consequently $\boldsymbol{p}$ has the same order). For scenarios ϕ and ψ, the output matrices are as follows.

$$\boldsymbol{H}_\phi = \begin{bmatrix} 4 & 3 \\ 7 & 6 \end{bmatrix} \quad \boldsymbol{H}_\psi = \begin{bmatrix} 4 & 6 \\ 8 & 4 \end{bmatrix}$$

In a scenario sequence, we can use the output matrices of the scenarios to compute the production times of outputs during any scenario in the sequence. For example, let p_0, p_1, p_2 and p_3 denote the production times of the first four outputs produced during the execution of the sequence $(\phi\psi)^\omega$. Using $\boldsymbol{H}_\phi$ and $\boldsymbol{H}_\psi$, the output production times can be computed as follows.

$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \boldsymbol{H}_\phi \boldsymbol{\gamma}_0 = \begin{bmatrix} 4 \\ 7 \end{bmatrix}, \begin{bmatrix} p_2 \\ p_3 \end{bmatrix} = \boldsymbol{H}_\psi \boldsymbol{\gamma}_1 = \boldsymbol{H}_\psi \boldsymbol{G}_\phi \boldsymbol{\gamma}_0 = \begin{bmatrix} 9 \\ 11 \end{bmatrix}$$

The computed production times can be confirmed by noticing the times at which the outputs are produced in Figure 2.5. We can obtain the output matrix of a scenario, similar to its state matrix. The output matrix is created by vertically augmenting the symbolic time-stamp vectors that correspond to production of outputs. For instance, in scenario ϕ, symbolic time-stamp vectors $\begin{bmatrix} 4 & 3 \end{bmatrix}$ and $\begin{bmatrix} 7 & 6 \end{bmatrix}$ correspond to production of outputs. By vertically augmenting these vectors we obtain $\boldsymbol{H}_\phi$.

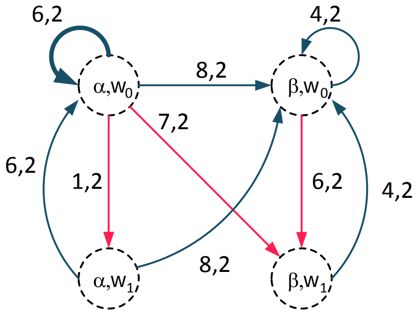


Figure 2.8: $(\max, +)$ automaton of the SADF shown in Figure 2.7.

2.8 FSM-SADF Throughput Analysis

This section discusses the FSM-SADF throughput analysis provided by Geilen et al. [55]. The analysis is based on the throughput definition given by Eq. 2.2. To determine the throughput, we need to find a worst-case scenario sequence among all possible scenario sequences, a sequence that determines the minimum of the throughput values. Recall that in the definition of throughput for scenario sequences (Eq. 2.1), the progress of time is measured as the norm of the state vector γ_i, which corresponds to an element with the maximum value. According to Eq. 2.6, every element of γ_i, is determined by some element of γ_{i-1} and *delay* by the corresponding dependency in the state matrix. This element in γ_{i-1} can in turn be traced back to a single element in γ_{i-2} and finally back to γ_0. This means that the relation between time progress and scenario sequences can be studied on individual elements of state vector and their individual dependencies expressed by the entries in the state matrices.

Figure 2.8 shows a graph which captures these dependencies for the example of Figure 2.7. This graph is called the *analysis* $(\max, +)$ *Automaton (MPA)*. The nodes in this graph represent the elements of the state vector in each of the states of the FSM. The nodes on the left and right sides are associated with the state vector elements corresponding to α and β, and the nodes on top and bottom sides are associated with the FSM states w_0 and w_1, respectively. For every FSM edge labelled s, we take the state matrix $\boldsymbol{G}_s$, and for every non $-\infty$ element in $\boldsymbol{G}_s$ we draw an edge between the corresponding state vector elements and label it with the value of that element (i.e. the delay) and with the reward of s (e.g., label $7, 2$ has a delay of 7 and reward

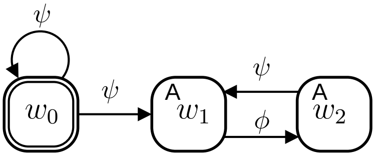


Figure 2.9: An example FSM with acceptance conditions.

of 2). The edges that correspond to scenarios ϕ an ψ are shown with red and blue colours for clarity.

A formal definition of the MPA corresponding to an SADF is provided in the form of a graph $M(R, E)$ with nodes R and edges E as follows.

$$\begin{aligned}
&R = \{(w, i) \mid w \in W, 1 \leq i \leq n(w)\} \\
&E = \{((w_1, i), [\boldsymbol{G}(s)]_{i,j}, o(s), (w_2, j)) \mid (w_1, s, w_2) \in \eta, 1 \leq i \leq n(w_1), \\
&\quad 1 \leq j \leq n(w_2), [\boldsymbol{G}(s)]_{i,j} \neq -\infty\}
\end{aligned}$$

In this equation, $n(w)$ is the size of the state vector in state w. Cycles of the MPA show the cyclic dependencies between state vector elements in execution of scenario sequences. The *ratio* of a cycle equals the sum of the delays on the edges of the cycles, divided by the sum of the rewards on the edges of the cycle. A cycle with the maximum ratio corresponds to scenario sequences that determine the infimum of the throughput values. Therefore, the throughput can be obtained by a Maximum Cycle Ratio (MCR) analysis on the MPA. The MCR of the MPA of the example SADF is 6/2. Therefore the throughput is 2/6 rewards per time unit. Critical cycles of an MPA are the cycles with the maximum cycle ratio. The MPA in Figure 2.8 has one critical cycle, which is from node (α, w_0) to itself.

The throughput analysis discussed above is provided for FSM-SADF models, following the definition of Geilen et al. [55]. Hence, acceptance conditions are not considered for the FSM. One needs to be careful when using this analysis with our SADF definition, when using regular expressions instead of FSMs. A language may not necessarily have an FSM representation without considering acceptance conditions. The throughput analysis on SADF models with such languages can still be used without considering acceptance conditions. However, the analysis may be conservative in some cases. For instance, for the SADF shown in Figure 2.7, assume the language of the model is given by regular expression $\psi^*(\psi\phi)^\omega$ instead of the FSM shown in that figure. The FSM representation of the language of this regular expression is shown in Figure 2.9. Observe that w_1 and w_2 are the accepting states. This means that the language of the FSM are the sequences of the runs that visit w_1 and w_2 infinitely often. As a result, sequence ψ^ω does not belong to this language, because it corresponds to a run that never visits either of the accepting states. If we do not consider the acceptance condition of the FSM representation of the language of $\psi^*(\psi\phi)^\omega$, we also include sequences such as ψ^ω in the language. Since ψ^ω happens to be the sequence with the worst-case throughput among all sequences, the throughput analysis will compute the conservative throughput value of 2/6, whereas the exact throughput is 4/10 outputs per time unit, which corresponds to sequence $(\psi\phi)^\omega$.

In general, the throughput analysis on SADF models with languages that are represented by a concatenation $\rho\sigma$ of an ordinary regular expression ρ and an omega-regular expression σ, may be conservative. Regular expression ρ represents a language of finite words. The sequences of this language constitute the prefixes of the sequences in the language of $\rho\sigma$. Therefore, the sequences of ρ do not affect the throughput, as they are finite sequences, while the throughput is entirely determined by the infinite part from σ. An MPA that is constructed from the FSM of SADF models with such languages without considering acceptance conditions, includes cycles that are related to ρ; therefore the analysis becomes conservative if the included cycles contribute to the worst-case throughput.

Conservative throughput analysis for SADF models that have FSMs with acceptance conditions can be avoided by straightforward static analysis of the FSM. In an SADF, we can transform every FSM with acceptance conditions to an FSM without acceptance conditions which has the same throughput (not the same language). This can be done by removing the states that can only be visited a finite number of times in any recognized sequence. For instance, in Figure 2.9, if we remove state w_0 and then perform the throughput analysis, the analysis result would be 4/10 which is the exact throughput.

It is good to mention that the FSM-SADF throughput analysis can also be used to analyse the throughput of SDF and CSDF models. A scenario that represents exactly one iteration of an SDFG or a CSDF graph is formed and assigned the reward of 1. Then, an FSM-SADF model is created using this scenario and an FSM that repeats this scenario. The MCR analysis on the MPA of this FSM-SADF is the same as obtaining the eigenvalue of its one and only state matrix.

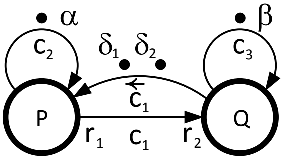


Figure 2.10: Modelling a buffer capacity on the scenario graphs of Figure 2.7.

2.9 SADF Buffer Requirements

In an SDFG, the channels that model data dependencies are called *buffer channels*. In the scenario graphs of Figure 2.7, channel c_1 is considered a buffer channel between a source actor P and a destination actor Q. SDF channels do not have control over the number of tokens that accumulates on them. A limited capacity buffer can be represented by modelling *back-pressure*, i.e., by blocking the firings of the source actor when the buffer does not have enough capacity to hold the tokens produced by source actor firings. To model back-pressure, it is common to draw a channel between the source and destination actors, in the direction that is opposite to the direction of the buffer channel [29]. We call these channels capacity control channels or *capacity channels* for short. Channel $\overleftarrow{c_1}$ in Figure 2.10 models a limited capacity buffer in the scenario graphs of Figure 2.7. The capacity channels have a number of tokens on them. The *capacity* of a buffer channel is the sum of the tokens on the buffer channel and the tokens on the capacity channel. In Figure 2.10, there is a buffer with 2 tokens capacity between P and Q. We consider a set $B \subseteq C$ of buffer channels with limited capacities for an SDFG $G(A, C)$. Let $b \in B$ be a buffer channel from an actor P to an actor Q ($P \neq Q$). To model limited capacity buffers, we need to add to G, a capacity channel $\overleftarrow{b}$ from Q to P, such that $Out_Q(\overleftarrow{b}) = In_Q(b)$ and $In_P(\overleftarrow{b}) = Out_P(b)$.

For an SADF, without loss of generality, we assume that a buffer with a certain capacity in one scenario graph is also a buffer with the same capacity in all other scenario graphs. To ensure that a buffer capacity that is occupied by a scenario cannot be used in an other scenario at the same time, the tokens that model the capacity of the buffers are assigned the same labels in all scenarios. For instance, δ_1 and δ_2 label the buffer capacities in Figure 2.10. We define a set U of *common buffers* for an SADF. Every common buffer $u \in U$ corresponds to a buffer channel in the scenario graph of every scenario $s \in S$. For an SADF, we define functions $b_s(u)$ to map every common buffer $u \in U$

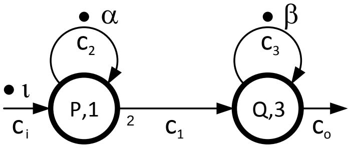
to the corresponding buffer channel in $g(s)$ of every scenario $s \in S$. Similarly, we define functions $c_s(u)$ that map every buffer $u \in U$ to the capacity channel of its corresponding buffer in $g(s)$. For the SADF in Figure 2.10, we define $U = \{u_{c_1}\}$. The common buffer u_{c_1} is mapped to buffer channel c_1 in both scenarios.

To realize an application, we need to allocate and distribute storage space to its buffers. To formalize the distribution of the storage space over the buffers in an SADF, we define *storage distributions*. Similar to the definition provided by Stuijk et al. [29], a storage distribution of an SADF on a set U of common buffers is a mapping $d : U \to \mathbb{N}_0$ that associates, with every buffer $u \in U$ the capacity of the buffer. The total allocated storage space, i.e. the *size* of a distribution, is defined as $|d| = \sum_{u \in U} d(u)$. For simplicity and without loss of generality, in this definition we assume that all tokens represent the same amount of storage space. This can be easily generalized by considering different weights for tokens in different buffers. We write $d_1 \preceq d_2$ if and only if for every $u \in U$, $d_1(u) \leq d_2(u)$. In the remainder, an SADF F that incorporates a storage distribution d is denoted by F_d. In F_d, for every $u \in U$, there exists $d(u) - It(b_s(u))$ tokens on channel $c_s(u)$ in every scenario graph $g(s)$. The SADF shown in Figure 2.10 incorporates a distribution d that allocates two tokens to u_{c_1}, i.e., $d(u_{c_1}) = 2$. We denote this distribution by $\langle u_{c_1} \rangle \mapsto \langle 2 \rangle$. The size of this distribution is 2 tokens.

Given a storage distribution, a scenario execution may *deadlock* due to insufficient capacities allocated to a buffer. In a scenario which is in a deadlock situation, no actor is able to fire anymore. For instance, in Figure 2.7, using distribution $d(u_{c_1}) = 2$ leads to a deadlock when executing scenario ψ, because in this scenario every firing of Q needs to consume 3 tokens from channel c_1, which can hold at most 2 tokens according to the given distribution. Hence, after the first firing of P, which fills the buffer c_1, neither P nor Q can fire, and consequently the scenario execution deadlocks.

2.10 Open SADF

In an SDFG, channels are defined between two actors. In some cases we may need to model execution dependencies between an actor and an external source. For instance, when we want to compose application models such that an actor in one application consumes the tokens produced by an actor in another application. For such cases, we define *open channels*. An open channel has only one end connected to an actor. An SDFG that contains at least one open channel is called an *open SDFG*. An *open scenario* is defined using an open



Reps: $[\mathsf{P},\mathsf{Q}] \mapsto [1,2]$
ITs: $c_2(\alpha), c_3(\beta)$
FTs: $c_2(\alpha), c_3(\beta)$
Reward: 2 (firings of Q)

Figure 2.11: Scenario ϕ^o.

SDFG. An *open SADF* includes at least one open scenario. Figure 2.11 shows an open SDF scenario, namely ϕ^o. Channel c_i is an open input channel and c_o is an open output channel in the scenario graph of ϕ^o. A token consumed from an input channel is called an *input token* and a token produced on an output channel is called an *output token*. Note that the number of outputs, i.e., the reward, is enough to compute the throughput of an open SADF. Nevertheless, we need the notion of input and output tokens to model interactions with external sources.

Open SDF scenarios can also be characterized by $(\max, +)$ algebra equations. For open SDF scenarios, the state and output equations include also the relations with the time-stamps of the input tokens. Let $\boldsymbol{\gamma}$ and $\boldsymbol{u}$ be vectors that contain the time-stamps of the initial and input tokens, respectively. After scenario s is executed, the time-stamps of final and output tokens are collected in vectors $\boldsymbol{\gamma}'$ and $\boldsymbol{y}$, respectively. The $(\max, +)$ characterization of a scenario s is described as follows [57].

$$\begin{bmatrix} \boldsymbol{\gamma}' \\ \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{G}_s & \boldsymbol{K}_s \\ \boldsymbol{H}_s & \boldsymbol{L}_s \end{bmatrix} \begin{bmatrix} \boldsymbol{\gamma} \\ \boldsymbol{u} \end{bmatrix} \tag{2.8}$$

$\boldsymbol{G}_s$, $\boldsymbol{H}_s$, $\boldsymbol{K}_s$ and $\boldsymbol{L}_s$ are *system matrices* that describe the timing dependency relations in scenario s.

System matrices of an open scenario can be computed using the symbolic simulation method provided in Section 2.7. For instance, consider scenario ϕ with the open SDFG shown in Figure 2.11. This scenario consumes 1 input token and produces 2 output tokens. Let ι label the input token and o_1 and o_2 label the output tokens, and t_ι, t_{o_1} and t_{o_2} denote the time-stamps of these tokens, respectively. For scenario ϕ^o, we can obtain the symbolic time-stamps using the vector $[\ t_\alpha \quad t_\beta \quad t_\iota\ ]^T$. For instance, the symbolic time-stamps of the tokens produced by firing of P can be obtained as $t = [\ 0 \quad -\infty \quad 0\ ][\ t_\alpha \quad t_\beta \quad t_\iota\ ]^T$. By symbolically simulating all actor firings in scenario ϕ and collecting the time-stamps of all final and output tokens we can obtain the following system of equations.

$$\begin{bmatrix} t'_\alpha \\ t'_\beta \\ t_{o_1} \\ t_{o_2} \end{bmatrix} = \left[\begin{array}{cc|c} 1 & -\infty & 1 \\ 7 & 6 & 7 \\ \hline 4 & 3 & 4 \\ 7 & 6 & 7 \end{array}\right] \begin{bmatrix} t_\alpha \\ t_\beta \\ t_\iota \end{bmatrix}$$

By letting $\boldsymbol{\gamma}' = \begin{bmatrix} t'_\alpha & t'_\beta \end{bmatrix}$, $\boldsymbol{\gamma} = \begin{bmatrix} t_\alpha & t_\beta \end{bmatrix}$, $\boldsymbol{u} = \begin{bmatrix} t_\iota \end{bmatrix}$ and $\boldsymbol{y} = \begin{bmatrix} t_{o_1} & t_{o_2} \end{bmatrix}$, the $(\max, +)$ matrices of scenario a are obtained as follows.

$$\boldsymbol{G}_{\phi^o} = \begin{bmatrix} 1 & -\infty \\ 7 & 6 \end{bmatrix} \boldsymbol{K}_{\phi^o} = \begin{bmatrix} 1 \\ 7 \end{bmatrix} \boldsymbol{H}_{\phi^o} = \begin{bmatrix} 4 & 3 \\ 7 & 6 \end{bmatrix} \boldsymbol{L}_{\phi^o} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

Observe that matrix $\boldsymbol{G}_{\phi^o}$ and $\boldsymbol{H}_{\phi^o}$ are the same as $\boldsymbol{G}_\phi$ and $\boldsymbol{H}_\phi$ defined in Section 2.7. We use the $(\max, +)$ characterization of open SDF scenarios when composing multiple SADF models.

Chapter 3

Modelling Multi-Scale Applications

As discussed in Section 1.2.5, dataflow models of multi-scale applications have actors acting at multiple granularity levels. For instance, a dataflow model of a video processing application with operations on frame, macro block and block level is multi-scale. The state of the art timing analysis methods [6, 58, 55] for both static and dynamic dataflow types aggregate the actor firings across all granularity levels into one, often large iteration (see Section 2.3 for the definition of SDF iteration). This iteration is repeated without exploiting the structure within such an iteration. This poses scalability issues to these analyses, because the behaviour of the large iteration contains a large number of actor firings, which need to be simulated for the analysis, e.g. the symbolic simulation in Section 2.7.1 required to extract state matrices of dataflow graphs for the throughput analysis.

In this chapter, we provide a modelling approach that takes a fresh perspective of what is happening inside the large iteration. We model the large iteration as a sequence of multiple small SDF scenarios. The state matrices of theses scenarios can be quickly generated as they contain only a small number of actor firings. Then, we show that the state matrix of the iteration, which is now modelled by a sequence of scenarios, can be compositionally constructed from the state matrices of its scenarios. Scenario sequences of multi-scale applications contain a lot of repetitions. Constructing the state matrices of a repeated sequence of scenarios amounts to raising matrices to the power of the number of repetitions, which scales logarithmically with the number of repe-

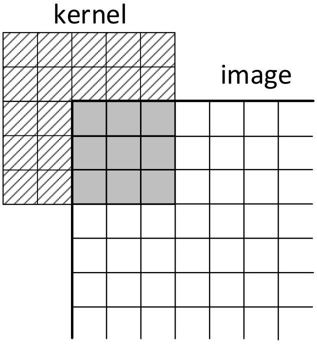


Figure 3.1: A 5x5 convolution filter.

titions, whereas constructing the same state matrix using a simulation, scales linearly in the number of actor firings in the repeated sequence. We show our approach for modelling dynamic and static applications using two practical examples respectively in Sections 3.1 and 3.2. As another advantage of representing multi-scale applications by sequences of scenarios, in Section 3.3, we present a method to compute the start and completion times of a selective set of actor firings to efficiently generate execution traces of multi-scale applications. This chapter is based on publications [39] and [40].

3.1 Modelling a Convolution Filter

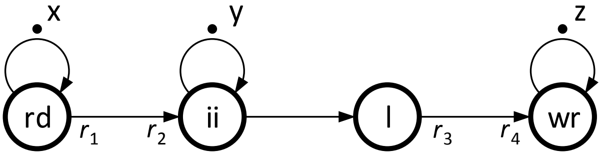
Convolution is one of the most important operations in signal and image processing. It is mostly used for feature extraction and is the core block of Convolutional Neural Networks (CNNs). Figure 3.1 shows a convolution filter that uses a kernel of 5 by 5 pixels and applies padding at the border of the frame. We assume that a single frame is an image with a width of W pixels and a height of H pixels. The video stream is delivered to the filter pixel-by-pixel, pixels being ordered frame-by-frame, line-by-line – within a frame, from top line to bottom line and from left pixel to right pixel. The filter produces filtered video with pixels in the same order.

The filter shows a specific pattern of data dependencies (it needs to collect the required input data for the convolution kernel) depending on the location of the kernel. Initially the centre of the kernel is located on the top left pixel of the frame (Figure 3.1) to produce the first output pixel. The kernel slides to the right, pixel by pixel, for a whole line and then starts from the left side of the next line. This continues until the centre of the kernel reaches the bottom right pixel of the frame. Then it continues with the next frame. When the kernel is on the initial location, it needs to have read 9 input pixels (kernel elements with gray colour in Figure 3.1) and use border padding for the elements that fall out of the image (kernel elements with diagonal pattern in Fig 3.1). Since the input data is ordered line-by-line, the filter needs to read 2 lines and 3 pixels from the input buffer before it can carry out the first convolution computation. Then the kernel slides one pixel to the right and it needs to read only one extra input pixel to do the second convolution computation. We summarize the behaviour of the filter in terms of reading input pixels and producing output pixels by the following pattern of phases.

1. **frame rush-in phase**: read pixels one by one without producing any output yet for 2 lines of the image, or $2W$ pixels;
2. **line rush-in phase**: read two pixels without producing output;
3. **line computation phase**: one pixel is read and one pixel is produced for a whole line minus two pixels: $W-2$ pixels;
4. **line rush-out phase**: two more pixels are produced without reading new input (using padding);
5. repeat phases 2–4 for $H-2$ lines;
6. **frame rush-out phase**: produce two more lines of output without new input (using padding): $2W$ pixels.

This phase pattern can be described by a repetitive sequence, composed of three different data dependency modes. Let *ri* (rush-in) denote a mode in which one input pixel is read but no output pixel is produced, *cm* (computation) be a mode that reads one input pixel and produces one output pixel and finally, *ro* (rush-out) denote a mode in which no input is read but one output is produced. Each of the phases in the pattern is composed of a constant repetition of one of theses modes. For instance the frame rush-in phase is composed of $2W$ repetitions of mode *ri*. This phase can be represented in a compact way by regular expression $(ri)^{2W}$. By representing the rest of the phases in a similar way, the repeated phase pattern can be described by the following regular expression.

$$\sigma_{conv} = \left((ri)^{2W} \left((ri)^2 (cm)^{W-2} (ro)^2 \right)^{H-2} (ro)^{2W} \right)^{\omega}$$



Scenario	*ri*	*cm*	*ro*
ETs	$e(\mathsf{rd}) = 2, e(\mathsf{ii}) = 2,$ $e(\mathsf{l}) = 1, e(\mathsf{wr}) = 0$	$e(\mathsf{rd}) = 2, e(\mathsf{ii}) = 4,$ $e(\mathsf{l}) = 5, e(\mathsf{wr}) = 2$	$e(\mathsf{rd}) = 0, e(\mathsf{ii}) = 3,$ $e(\mathsf{l}) = 4, e(\mathsf{wr}) = 2$
Rates	$r_1 = 1, r_2 = 1,$ $r_3 = 0, r_2 = 0$	$r_1 = 1, r_2 = 1,$ $r_3 = 1, r_4 = 1$	$r_1 = 0, r_2 = 0,$ $r_3 = 1, r_4 = 1$
Reward	0 (firings of wr)	1 (firings of wr)	1 (firings of wr)
Reps	$[\mathsf{rd}, \mathsf{ii}, \mathsf{l}, \mathsf{wr}] \mapsto$ $[1, 1, 1, 0]$	$[\mathsf{rd}, \mathsf{ii}, \mathsf{l}, \mathsf{wr}] \mapsto$ $[1, 1, 1, 1]$	$[\mathsf{rd}, \mathsf{ii}, \mathsf{l}, \mathsf{wr}] \mapsto$ $[0, 1, 1, 1]$
ITs	$c_1(x), c_2(y), c_3(z)$	$c_1(x), c_2(y), c_3(z)$	$c_1(x), c_2(y), c_3(z)$
FTs	$c_1(x), c_2(y), c_3(z)$	$c_1(x), c_2(y), c_3(z)$	$c_1(x), c_2(y), c_3(z)$

Figure 3.2: SDF scenarios of the convolution filter modes.

Since the modes in the convolution filter correspond to fixed data dependency relations, they can be described using SDF scenarios. We use the SDFG shown in Figure 3.2 to describe the modes in the filter. The rd actor models reading of the input pixels. The ii and l actors model the starting of the convolution computations or the *initiation interval* and computation *latency* respectively. The wr actor models the production and writing of the output pixels. The scenario specifications are shown for every mode in the table in Figure 3.2. The specific data dependencies in every mode are modelled by adjusting some channel rates. For instance, in mode *cm*, the channel from rd to ii has a rate of 1 on both ends. This ensures that the convolution computations in this mode start only after the reading is complete. The rates on this channel in scenario *ro* are both 0. This allows the convolution computations to start without reading a new pixel in mode *ro*. The repetition of actors in every scenario is designed to project the behaviour of the corresponding mode. For instance, mode *cm* involves one firing from every actor in the graph, as it represents reading an input, performing a computation and, writing an output. The execution times of actors are obtained from an FPGA implementation; they are reported in clock cycles.

Observe that x, y and z label the initial and final tokens on self-edges of actors rd, ii and wr, respectively. In the implementation we model in this example, input pixels are streamed sequentially. To ensure that the filter reads only one new pixel at a time during the execution of the filter sequence, we put a self-edge with one token on actor rd. The token on the self-edge has the same label both as initial token and final token, in all scenarios. In Section 2.4, we showed that during the execution of a sequence of scenarios, actors may execute in parallel. The only dependencies that determine when actor firings start are those expressed by consumption and production of tokens. Consequently, subsequent scenarios can be executed in parallel when there are no such dependencies. For instance, as shown in Figure 2.5, during the execution of sequence $(\phi\psi)^{\omega}$, actors of both ϕ and ψ scenarios are executing at the same time. The exception is when an actor in scenario ϕ or ψ is waiting for an initial token to become available, and the availability of this initial token is bound to the production of a final token in the preceding scenario which has the same label as the initial token. For instance, the first firing of actor P in every ψ scenario in the sequence starts once the final token α in the preceding ϕ scenario is produced, due to the dependency of actor P on the availability of initial token α. As the output pixels in the convolution filter are sequentially streamed, we use the same technique to prevent concurrent executions of actor wr by putting a self-edge with one token on this actor.

This model represents a pipelined implementation of a convolution computation on a single FPGA accelerator with a certain initiation interval and latency. The sequential nature of the execution due to the single resource is modelled in the scenario graphs. Similar to sequential reading and writing of pixels, we prohibit concurrent executions of the convolution computations, by putting a self-edge with one token on actor ii. If we wish to allow multiple convolution computations at a time, using multiple accelerators in FPGA implementations for instance, we can put more than one token on the self-edge of ii. For example, by putting n tokens on the self-edge of ii, we allow maximally n concurrent executions of ii. One could also allow an arbitrary number of concurrent executions by removing the edge altogether.

To obtain the state matrices of the scenarios, consider the symbolic state vector $\boldsymbol{\gamma}^{conv} = [\ t_x \quad t_y \quad t_z\ ]^T$, where t_x, t_y and t_z denote the time-stamps of tokens x, y and z, respectively. Using a symbolic simulation that involves 10 actor firings (the 4 firings included in mode cm plus the 3 firings included in each of the modes ri and ro), we can obtain the state matrices of all scenarios in the filter as follows.

$$\boldsymbol{G}_{ri} = \begin{bmatrix} 2 & -\infty & -\infty \\ 4 & 2 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix}, \quad \boldsymbol{G}_{cm} = \begin{bmatrix} 2 & -\infty & -\infty \\ 6 & 4 & -\infty \\ 13 & 11 & 2 \end{bmatrix},$$

$$\boldsymbol{G}_{ro} = \begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 3 & -\infty \\ -\infty & 9 & 2 \end{bmatrix}$$

Similar to the state matrices that correspond to the timing behaviour of modes, we can define a state matrix that corresponds to the timing behaviour of a pattern of modes, such as the frame pattern. For the frame pattern, such a matrix, called the frame-level state matrix, abstracts all actor firings during the production of every individual pixel of the frame, in the relation between the initial state vector and the state vector after the last scenario of the frame pattern is executed. Let $\boldsymbol{f}_i$ denote the state vector after the production of the i^{th} frame and $\boldsymbol{f}_0$ denote the initial state vector. We define this relation as follows.

$$\boldsymbol{f}_i = \boldsymbol{G}_{frame}\boldsymbol{f}_{i-1}$$

Now we show how to compute $\boldsymbol{G}_{frame}$. Let γ_i^{conv} denote the state vector after the execution of the i^{th} scenario in the sequence of the language represented by σ_{conv} and, γ_0^{conv} denote the initial state vector. By repeatedly using Eq. 2.5 on this sequence, we can obtain the following set of equations.

$$\begin{aligned}
\gamma_1^{conv} &= \boldsymbol{G}_{ri}\gamma_0^{conv} \\
\gamma_2^{conv} &= \boldsymbol{G}_{ri}\gamma_1^{conv} = \boldsymbol{G}_{ri}\boldsymbol{G}_{ri}\gamma_0^{conv} \\
&\vdots \\
\gamma_{2W}^{conv} &= \boldsymbol{G}_{ri}^{2W}\gamma_0^{conv} \\
\gamma_{2W+1}^{conv} &= \boldsymbol{G}_{ri}\boldsymbol{G}_{ri}^{2W}\gamma_0^{conv} \\
\gamma_{2W+2}^{conv} &= \boldsymbol{G}_{ri}^{2}\boldsymbol{G}_{ri}^{2W}\gamma_0^{conv} \\
\gamma_{2W+3}^{conv} &= \boldsymbol{G}_{cm}\boldsymbol{G}_{ri}^{2}\boldsymbol{G}_{ri}^{2W}\gamma_0^{conv} \\
&\vdots \\
\gamma_{2W+(W+2)(H-2)+2W}^{conv} &= \boldsymbol{G}_{ro}^{2W}\left(\boldsymbol{G}_{ro}^{2}\boldsymbol{G}_{cm}^{W-2}\boldsymbol{G}_{ri}^{2}\right)^{H-2}\boldsymbol{G}_{ri}^{2W}\gamma_0^{conv}
\end{aligned}$$

The last equation above computes the state vector after production of the first complete frame, i.e., $\boldsymbol{f}_1$. Since this pattern repeats for the next frames, we obtain

$$\boldsymbol{f}_i = \left(\boldsymbol{G}_{ro}^{2W} \left(\boldsymbol{G}_{ro}^{2}\boldsymbol{G}_{cm}^{W-2}\boldsymbol{G}_{ri}^{2}\right)^{H-2} \boldsymbol{G}_{ri}^{2W}\right) \boldsymbol{f}_{i-1}.$$

This equation shows that $\boldsymbol{G}_{frame}$ can be computed by multiplying the matrices of scenarios in the order that is the reverse of the appearance order of their corresponding modes in the frame pattern. Let's consider a small 9×11 frame ($W = 9, H = 11$) to be able to show the execution traces of the filter. $\boldsymbol{G}_{frame}$ is computed as follows.

$$\begin{aligned}\boldsymbol{G}_{frame} &= \boldsymbol{G}_{ro}^{2W} \left(\boldsymbol{G}_{ro}^{2}\boldsymbol{G}_{cm}^{W-2}\boldsymbol{G}_{ri}^{2}\right)^{H-2} \boldsymbol{G}_{ri}^{2W} = \boldsymbol{G}_{ro}^{18} \left(\boldsymbol{G}_{ro}^{2}\boldsymbol{G}_{cm}^{7}\boldsymbol{G}_{ri}^{2}\right)^{9} \boldsymbol{G}_{ri}^{18} \\ &= \begin{bmatrix} 198 & -\infty & -\infty \\ 434 & 432 & -\infty \\ 440 & 438 & 198 \end{bmatrix}\end{aligned}$$

$\boldsymbol{G}_{frame}$ can be used for analysis, similar to the state matrix of a CSDF or an SDF graph. For example, the frame-level throughput is obtained from the eigenvalue of this matrix. The eigenvalue of $\boldsymbol{G}_{frame}$ shows that the growth rate of the state vector during indefinite repetition of the frame pattern is 432 clock cycles per frame. Moreover, the critical cycle of the adjacency graph of this matrix reveals a critical dependency on the second column of the second row of $\boldsymbol{G}_{frame}$. This element corresponds to the time difference between the time-stamp of token y, before and after the production of a frame. This critical dependency is also observed in the execution trace of the filter in Figure 3.3. Note that except for the first ii firing, firings of ii start immediately after the previous ii firing completes. This means that firings of ii create a critical cycle. As shown in the first complete frame, with the first ii firing starting at 2 and the last firing completing at 434, we conclude that this critical cycle is 432 clock cycles long. Hence, the throughout is equal to 99/432 pixels per time unit or 1/432 frames per time unit.

The matrix $\boldsymbol{G}_{frame}$ can be efficiently computed. As we just saw, when a scenario (or a scenario sequence) corresponds to a matrix $\boldsymbol{G}$, then n repetitions of that scenario correspond to the matrix $\boldsymbol{G}^n$, which can be computed from $\boldsymbol{G}$, in $\mathcal{O}(\log n)$ time. When analysing the throughput of the convolution filter with a conversion to single-rate SDF [59], the state of the art CSDF [6, 60] and FSM-SADF [55] analyses, a simulation that goes through all variations in the phase pattern is needed to obtain the essential timing information (e.g. the frame-level state matrix). For a 2048×2048 frame, this simulation contains $1.67 \cdot 10^7$ firings (3 firings for every pixel in the rush-in and rush-out phases and

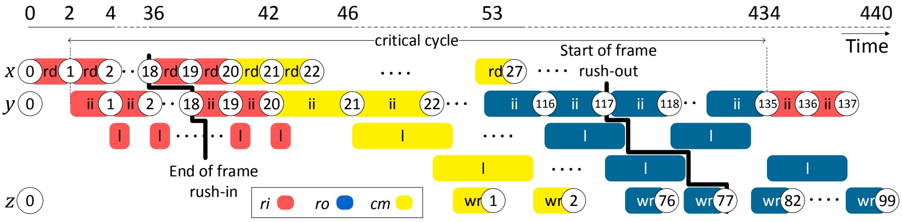
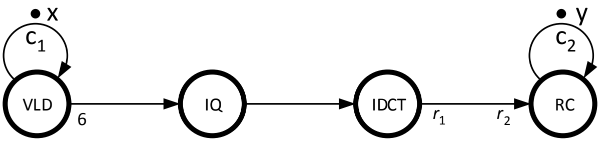


Figure 3.3: Execution of the convolution filter on a 9 × 11 frame.

4 firings for every pixel in the computation phase). The run-time of a timing analysis that evokes such a simulation on today's computers is in the order of seconds. However, this is still too much for trade-off analysis methods such as buffer sizing, since the timing analysis techniques might be called more than 300 thousand times during the analysis [29]. In contrast, using our approach, which includes only 10 actor firings and less than a 100 matrix multiplications, the same result is computed in less than 1 millisecond.

3.2 Modelling an H.263 Decoder

In Section 1.2.2 we provided an SDF model for an H.263 decoder, applied to video streams of small frames of size 352 × 288 pixels. An iteration of this graph contains a large number (30889) of actor firings, as the application is multi-scale. Similar to the convolution filter, we can decompose the iteration into a sequence of SDF scenarios by recognizing distinct sub-behaviours with fixed data dependencies inside the iteration. For instance, we can recognize the following two sub-behaviours. The first sub-behaviour is associated with decoding a single macro block. We describe this behaviour as a scenario called, macro block or *mb* scenario for short. Scenario *mb* contains one firing of VLD and 6 firings from each of the actors IQ and IDCT. The second behaviour is associated with the decoding of the last macro block in a frame which enables the reconstruction task. We describe this behaviour using a scenario called, decode and reconstruct or *drc* for short. Scenario *drc* includes one firing of VLD, 6 firings from each of the actors IQ and IDCT and one firing of RC. The scenario graphs and specifications of these two scenarios are given in Figure 3.4. In the scenario specifications, a reward of 1 is associated with the production of a complete frame, which is modelled by one firing of RC. The



Scenario	*mb*	*drc*
ETs	$e(\text{VLD}) = 44, e(\text{IQ}) = 93,$ $e(\text{IDCT}) = 78, e(\text{RC}) = 0$	$e(\text{VLD}) = 44, e(\text{IQ}) = 93,$ $e(\text{IDCT}) = 78, e(\text{RC}) = 43832$
Rates	$r_1 = 0, r_2 = 0,$	$r_1 = 1, r_2 = 1,$
Reward	0 (firings of RC)	1 (firings of RC)
Reps	[VLD,IQ,IDCT,RC] $\mapsto$ $[1, 6, 6, 0]$	[VLD,IQ,IDCT,RC] $\mapsto$ $[1, 6, 6, 1]$
ITs	$c_1(x), c_2(y)$	$c_1(x), c_2(y)$
FTs	$c_1(x), c_2(y)$	$c_1(x), c_2(y)$

Figure 3.4: SDF scenarios of the H.263 decoder sub-behaviours.

actor execution times are given in cycles of an ARM7 processor. We adapted the execution times to 352×288 frames from the results provided by Stuijk [6] for 176×144 frames using a linear extrapolation.

Using these two scenarios, the frame-level behaviour of the decoder can be formed as a sequence of scenarios which repeats scenario *mb* for 2375 times and then continues with scenario *drc*. The language that contains only this scenario sequence can be represented by the regular expression $((mb)^{2375}(drc))^{\omega}$. Considering a state vector $\boldsymbol{\gamma}^{H.263} = [\ t_x \quad t_y\ ]^T$, the state matrices of the scenarios can be obtained as follows (by simulating 27 actor firings, which is the sum of the actor repetitions in both scenarios).

$$\boldsymbol{G}_{mb} = \begin{bmatrix} 44 & -\infty \\ -\infty & 0 \end{bmatrix}, \quad \boldsymbol{G}_{drc} = \begin{bmatrix} 44 & -\infty \\ 44047 & 43832 \end{bmatrix}$$

Now we can find the state matrix of the H.263 decoder as follows.

$$\boldsymbol{G}_{H.263} = \boldsymbol{G}_{drc}\boldsymbol{G}_{mb}^{2375} = \begin{bmatrix} 44 & -\infty \\ 44047 & 43832 \end{bmatrix} \left(\begin{bmatrix} 44 & -\infty \\ -\infty & 0 \end{bmatrix} \right)^{2375}$$

$$= \begin{bmatrix} 104544 & -\infty \\ 148547 & 43832 \end{bmatrix}$$

The eigenvalue of this matrix reveals that the cycle time of the H.263 decoder is 104544, that is about 105 kilocycles per frame.

3.3 Generating Execution Traces

An execution trace of a dataflow behaviour is a visualization of the start and completion times of actor firings in that behaviour. In this and the previous chapter, we used the execution traces of scenario sequences to explain several concepts. The execution trace of a scenario sequence provides tangible information about a particular behaviour in the application, which not only helps with a better understanding of that particular behaviour, but also facilitates performance debugging and detecting anomalies via visual inspection.

Since behaviours of multi-scale applications involve many actor firings, simulating and storing the start and completion times of all actor firings in their behaviour becomes very time consuming. Besides, the execution trace of some parts of the execution may not be informative, for instance because they contain a lot of repeated behaviours. In fact, with multi-scale applications we are often interested in viewing a particular part of the execution. For example, in the convolution filter, we may be interested in viewing the execution trace of actor firings around the production of the first or last output pixel in a frame.

In the previous two sections we showed that we can represent the behaviour of multi-scale applications by sequences of small scenarios expressed as regular expressions, and consequently, gain scalability in the timing analysis methods. In this section, we exploit this representation and provide an approach to efficiently compute the start and completion times of actor firings for which we wish to generate their execution traces, at any selected part of their behaviour, without having to simulate all actor firings that lead up to those actor firings.

In Section 2.7, we showed how the availability times of tokens in a scenario can be represented by symbolic time-stamps in the form of $t = \boldsymbol{g}^T\boldsymbol{t}$. In the same way, we can represent the start and completion times of actor firings. For instance, consider scenario *cm* in the convolution filer example. Actor rd starts firing as soon as the initial token x becomes available. The start time of this firing can be expressed as $s_{\mathsf{rd}} = [\ 0 \quad -\infty \quad -\infty\ ]\boldsymbol{\gamma}^{conv}$. Similarly, the completion time of this firing can be expressed as $c_{\mathsf{rd}} = s_{\mathsf{rd}} + 2 = [\ 2 \quad -\infty \quad -\infty\ ]\boldsymbol{\gamma}^{conv}$. Such a representation can be obtained for all actor firings in a scenario. When the start or completion time of an actor firing is represented in the form of $t = \boldsymbol{g}^T\boldsymbol{t}$, we refer to it as the *symbolic start time* or *symbolic completion time*, respectively.

Let vectors $\boldsymbol{s}$ and $\boldsymbol{c}$ contain the start and completion times of all actor firings in a scenario, respectively. After obtaining the symbolic start and completion times of the firings in a scenario s, we can represent the relation between the vector of start/completion times of actor firings and the state vector by the following equation.

$$\begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{c} \end{bmatrix} = \boldsymbol{T}_s \boldsymbol{\gamma} \tag{3.1}$$

We refer to $\boldsymbol{T}_s$ as the *tracing matrix* of scenario s. Each row of the upper part of this matrix relates the start time of an actor firing to the state vector. Any row in the lower part of this matrix relates the completion time of an actor firing to the state vector. For scenario cm, this matrix is constructed by simulating the scenario symbolically, and collecting the symbolic start and completion times of all firings in a matrix, as follows.

$$\begin{bmatrix} s_{\mathsf{rd}} \\ s_{\mathsf{ii}} \\ s_{\mathsf{l}} \\ s_{\mathsf{wr}} \\ c_{\mathsf{rd}} \\ c_{\mathsf{ii}} \\ c_{\mathsf{l}} \\ c_{\mathsf{wr}} \end{bmatrix} = \boldsymbol{T}_{cm} \boldsymbol{\gamma}^{conv} = \begin{bmatrix} 0 & -\infty & -\infty \\ 2 & 0 & -\infty \\ 6 & 4 & -\infty \\ 11 & 9 & 0 \\ 2 & -\infty & -\infty \\ 6 & 4 & -\infty \\ 11 & 9 & -\infty \\ 13 & 11 & 2 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

We can use the tracing matrices in combination with the state matrices to obtain the start and completion times of all actor firings in any selected scenario s_i in a sequence $\bar{s} = s_1 s_2 \cdots$, without having to simulate all actor firings involved in the scenarios before s_i. This is done using the state matrices to obtain the initial state vector of the execution of scenario s_i. Then, using matrix $\boldsymbol{T}_{s_i}$ we obtain the start and completion times of all actor firings in s_i. The state vector after execution of a repeated sequence of scenarios can be computed in logarithmic time in the number of repetitions.

Consider the convolution filter example. We would like to compute the start and completion times of actor firings in the first cm scenario that appears in the sequence, i.e., the scenario in which the first output pixel is produced. Considering frames of size 9×11 pixels, this scenario executes after scenario ri is executed for 20 times (2 lines and 2 pixels). This means we can obtain the state vector just before the execution of scenario cm as follows (see Section 3.1).

$$\boldsymbol{\gamma}_{20}^{conv} = \boldsymbol{G}_{ri}^{20} \boldsymbol{\gamma}_0^{conv} = \begin{bmatrix} 40 & -\infty & -\infty \\ 42 & 40 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 40 \\ 42 \\ 0 \end{bmatrix}$$

Now we can use $\boldsymbol{T}_{cm}$ and $\boldsymbol{\gamma}_{20}$ to obtain the start and completion times of

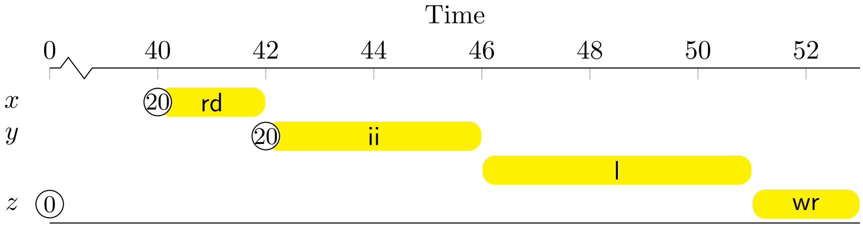


Figure 3.5: The execution trace of the first *cm* scenario.

actor firings during the execution of this scenario as follows.

$$\begin{bmatrix} s_{\mathsf{rd}} \\ s_{\mathsf{ii}} \\ s_{\mathsf{l}} \\ s_{\mathsf{wr}} \\ c_{\mathsf{rd}} \\ c_{\mathsf{ii}} \\ c_{\mathsf{l}} \\ c_{\mathsf{wr}} \end{bmatrix} = \boldsymbol{T}_{cm}\gamma_{20}^{conv} = \boldsymbol{T}_{cm}\boldsymbol{G}_{ri}^{20}\gamma_{0}^{conv}$$

$$= \begin{bmatrix} 0 & -\infty & -\infty \\ 2 & 0 & -\infty \\ 6 & 4 & -\infty \\ 11 & 9 & 0 \\ 2 & -\infty & -\infty \\ 6 & 4 & -\infty \\ 11 & 9 & -\infty \\ 13 & 11 & 2 \end{bmatrix} \begin{bmatrix} 40 \\ 42 \\ 0 \end{bmatrix} = \begin{bmatrix} 40 \\ 42 \\ 46 \\ 51 \\ 42 \\ 46 \\ 51 \\ 53 \end{bmatrix}$$

We used the computed start and completion times to generate the execution trace of the first *cm* scenario of the frame pattern, shown in Figure 3.5. Observe that this execution trace is consistent with Figure 3.3.

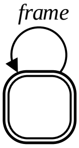
Chapter 4

Scalable Timing Analysis

In the previous chapter, we introduced a method to model multi-scale applications. A multi-scale application is modelled by an SADF, the language of which is represented by a regular expression. Using this model, we showed how to efficiently compute the throughput when the language contains only one scenario sequence. In this chapter, we provide a scalable throughput analysis for SADF models with arbitrary regular expressions. Moreover, we provide the first latency analysis of SADF models. This chapter is based on publication [39].

4.1 Throughput Analysis

We explain our throughput analysis by first recalling how we computed the throughput of the convolution filter in Chapter 3. Recall the language defined for the SADF model of the convolution filter in Section 3.1. It defines only one sequence, which is indefinite repetition of the sequence $ri^{2W}((ri)^2(cm)^{W-2}(ro)^2)^{H-2}(ro)^{2W}$. This finite sequence, which we refer to as the *frame sequence*, corresponds to the production of a complete frame of size $W \times H$. We computed the throughput in three steps. First, we represented the worst-case behaviour of the frame sequence by a state matrix, which we called the frame-level state matrix $\boldsymbol{G}_{frame}$. Second, we calculated the total number of output pixels produced during the execution of the frame sequence, which is WH. Finally, we obtained the throughput by calculating the ratio of the total number of output pixels in the frame sequence over the cycle time of the frame sequence which is obtained from an MCM analysis on $\boldsymbol{G}_{frame}$.



(a) The compact FSM.

Scenario	*frame*
Reward	$9 \times 11 = 99$ output pixels
State matrix	$\begin{bmatrix} 198 & -\infty & -\infty \\ 434 & 432 & -\infty \\ 440 & 438 & 198 \end{bmatrix}$

(b) Scenario specifications.

Figure 4.1: The compact SADF model of the convolution filter.

By taking the first two steps, in fact we *abstracted* the frame sequence in a single scenario with state matrix representation $\boldsymbol{G}_{frame}$, which produces WH outputs. Let's call this abstract scenario, the *frame* scenario. Note that the *frame* scenario is only characterized by its state matrix $\boldsymbol{G}_{frame}$, and it does not have an SDFG representation. Nonetheless, the state matrix representation is enough to compute the throughput. The final step can be seen as computing the MCR of an MPA, created from an SADF that repeatedly executes the abstract *frame* scenario. We refer to this SADF as the compact SADF model of the convolution filter, as it executes only one scenario. This SADF is shown in Figure 4.1, considering 9 × 11 frames. Observe that the repeated execution of the *frame* scenario is represented by an FSM with one state, and that the *frame* scenario is characterized by its state matrix (computed in Section 3.1) and the number of output pixels it produces. The MPA of the compact SADF model of the convolution filter has only three states as $\boldsymbol{G}_{frame}$ is of size 3×3. Hence, the MCR analysis on this MPA quickly terminates.

Abstracting the worst-case behaviour of the frame sequence by the *frame* scenario is the key to a scalable throughput analysis, compared to when the throughput analysis given by Geilen et.al [55] is directly used with the language of the convolution filter. For the latter case, we first need to convert the regular expression $(ri^{2W}((ri)^2(cm)^{W-2}(ro)^2)^{H-2}(ro)^{2W})^\omega$ into an equivalent FSM representation. The number of states in FSM representations of this regular expression is at least the number of scenarios in the frame sequence, which is $2W + (W+2)(H-2) + 2W$. Consequently, the number of states in the MPA obtained from the FSM is of the same order as the number of FSM states. Therefore, performing the final MCR analysis on this MPA is slower, compared to when performed on the MPA created from the compact SADF model of the convolution filter, which has only three states.

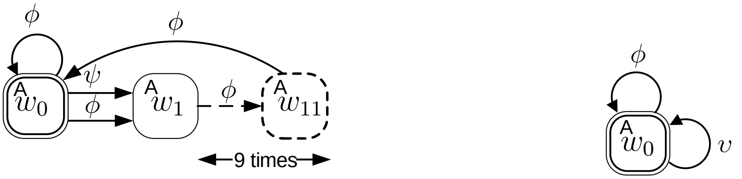
In general, to characterize a scenario that abstracts a finite sequence $\bar{s}$ of scenarios, we need to compute the number of outputs produced during $\bar{s}$, and the state matrix representation of $\bar{s}$. The total number of outputs produced during a finite scenario sequence is computed by adding up the outputs produced in every scenario in the sequence. For a sequence $\bar{s} = s_1 \cdots s_n$, the total number of outputs produced during the execution of this sequence is

$$o(\bar{s}) = o(s_1) + \cdots + o(s_n). \tag{4.1}$$

As discussed in Section 3.1, we compute the state matrix of a scenario sequence by multiplying the state matrices of scenarios in the order that is the reverse of the appearance order of their corresponding scenarios in the sequence. Let $\boldsymbol{G}(\bar{s})$ denote the state matrix of a sequence $\bar{s} = s_1 \cdots s_n$. Then we obtain

$$\boldsymbol{G}(\bar{s}) = \boldsymbol{G}_{s_n} \cdots \boldsymbol{G}_{s_1}. \tag{4.2}$$

We generalize the notion of abstracting a sequence of scenarios by a single scenario to abstracting a *set* of scenario sequences by a single scenario. We illustrate this by an example. To create an interesting example, let's consider another language for the SADF shown in Figure 2.7, instead of the FSM shown in that figure. Assume the new language is given by the regular expression $((\phi^{11} + \psi\phi^{10})^*\phi^*)^\omega$. In this regular expression, the sub-expression $\phi^{11} + \psi\phi^{10}$ can be abstracted into a single scenario, i.e., a single scenario can be found, which represents the worst-case behaviour among all scenario sequences defined by the sub-expression. The sub-expression defines the set that contains the two sequences: ϕ^{11} and $\psi\phi^{10}$. First, both sequences in the set produce the same number of outputs, since $o(\phi^{11}) = 11 \times 2 = 22$ and $o(\psi\phi^{10}) = 2 + 10 \times 2 = 22$. Second, for each of the sequences, the worst-case behaviour can be defined in the form of state matrices, as $\boldsymbol{G}(\phi^{11}) = \boldsymbol{G}_\phi^{11}$ and $\boldsymbol{G}(\psi\phi^{10}) = \boldsymbol{G}_\phi^{10}\boldsymbol{G}_\psi$. Since every sequence in the set produces the same number of outputs, and each sequence has a state matrix to represent its worst-case behaviour, a single scenario that produces the same number of outputs and has the worst-case behaviour among all the sequences can be defined to represent the worst-case behaviour of this set. Let's call it scenario v. We define that scenario v produces 22 outputs, i.e., $o(v) = 22$. Later we show that the state matrix of scenario v is defined as the maximum of the state matrices of the two sequences, i.e., $\boldsymbol{G}_v = \boldsymbol{G}(\psi\phi^{10}) \oplus \boldsymbol{G}(\phi^{11}) = \boldsymbol{G}_\phi^{10}\boldsymbol{G}_\psi \oplus \boldsymbol{G}_\phi^{11}$.



(a) An FSM representation of regular expression $((\phi^{11} + \psi\phi^{10})^*\phi^*)^\omega$.

(b) An FSM representation of regular expression $(\upsilon^*\phi^*)^\omega$.

Figure 4.2: The given and compact FSMs.

The set of scenario sequences defined by the sub-expression ϕ^* cannot be abstracted by a single scenario. This is because this sub-expression expresses a non-deterministic choice between an infinite number of sequences. Hence, neither the number of outputs nor a worst-case state matrix can be defined for this sub-expression. We address such cases using their FSM representations, as we did in Section 2.8. We first replace the sub-expression $\phi^{11} + \psi\phi^{10}$ with scenario υ in the given regular expression $((\phi^{11} + \psi\phi^{10})^*\phi^*)^\omega$, and we obtain regular expression $(\upsilon^*\phi^*)^\omega$. To compute the throughput we convert this regular expression to an ϵ-free FSM (e.g. by the method provided by Brüggemann-Klein [61]), and use the throughput analysis of Geilen et al. [55] (see Section 2.8). Similar to the convolution filter example, after abstracting the set of sequences by a single scenario, we obtain an SADF that has a compact FSM compared to the FSM representation of the given regular expression. For the sake of comparison we show these two FSMs in Figure 4.2. As shown in the figure, the number of states has reduced from 11 to only 1. This in turn reduces the run-time of the final MCR analysis to compute the throughput. Note that the FSM obtained by the Brüggemann-Klein algorithm always has some acceptance conditions. Therefore, the throughput analysis may be conservative in a few cases (see Section 2.8). In this example there is only one state, which is accepting. Since this is equivalent to not having an acceptance condition, the analysis is exact.

To compute the throughput of an SADF, we propose to transform the given SADF to an FSM-SADF with a compact FSM, and then use the throughput analysis given by Geilen et al. [55]. We define scenarios for the transformed FSM-SADF from ordinary regular expressions that represent *finite sets* of scenario sequences in the given SADF as long as the execution of every sequence in the set produces the same number of outputs. We refer to these expressions as *abstractable* expressions. For instance, $\phi^{11} + \psi\phi^{10}$ is an abstractable ex-

$$\left((ri)^{2W}\left((ri)^2(cm)^{W-2}(ro)^2\right)^{H-2}(ro)^{2W}\right)^{\omega}$$

(a) Regular expression σ_{conv}.

Sce.	*ri*	*cm*	*ro*
Rew.	0 output pixels	1 output pixels	1 output pixels
S. M.	$\begin{bmatrix} 2 & -\infty & -\infty \\ 4 & 2 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix}$	$\begin{bmatrix} 2 & -\infty & -\infty \\ 6 & 4 & -\infty \\ 13 & 11 & 2 \end{bmatrix}$	$\begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 3 & -\infty \\ -\infty & 9 & 2 \end{bmatrix}$

(b) Scenario specifications.

Figure 4.3: The matrix-level representation of the convolution filter in Section 3.1.

pression. An abstractable expression does not contain Kleene iterations since they represent infinite sets, e.g., ϕ^* is not abstractable. Moreover, such a regular expression does not contain choices between two scenario sequences that produce different numbers of outputs. For instance, $\phi^2 + \psi$ is not abstractable because $o(\phi^2) = 2 \times 2 = 4 \neq o(\psi) = 2$.

We provide a formal definition to transform a given SADF to a compact SADF by replacing all abstractable sub-expressions with new scenarios. As we showed with examples, before performing the transformation, we need to compute the state matrices of the scenarios of the given SADF, and represent its language with a regular expression. These two steps result in a new representation for the given SADF which we refer to as the matrix-level representation. This representation of SADF defines scenario names, state matrices of scenarios (instead of the ordinary SDFG representations), number of outputs produced in scenarios and a regular expression. In the following, we first define the new SADF representation, then we define the transformation to a compact SADF.

Definition 4.1. *Consider an SADF $F\,(S, g, r, i, f, o, L)$. A tuple $M\,(S, \boldsymbol{G}, o, \sigma)$ is defined as the matrix-level representation of F. The set S is the set of scenario names for every scenario in F. Function $\boldsymbol{G}(s)$ maps every scenario of F to its state matrix. The output function $o(s)$ maps every scenario of F to the number of outputs produced in that scenario. Regular expression σ represents the language L of F.*

Recall the convolution filter model in Section 3.1. The scenarios of this model is shown in Figure 3.2 and its language is given by the regular expression

σ_{conv}. Using the definition above, this model is transformed into the matrix-level representation shown in Figure 4.3. The scenarios, rewards and state matrices are given in the table of Figure 4.3b and the regular expression is given in Figure 4.3a.

Definition 4.2. *Given an SADF $M(S, \boldsymbol{G}, o, \sigma)$, the compact transformation of M is an SADF $M^c(S^c, \boldsymbol{G}^c, o^c, \sigma^c)$ defined as follows.*

The inductively defined function c is used to compute the compacted regular expression $\sigma^c = c(\sigma)$, by replacing all abstractable sub-expressions with a single new scenario.

$$\begin{aligned} c(\sigma):\quad & c(\rho^\omega) = (c(\rho))^\omega \\ & c(\rho\sigma) = c(\rho)c(\sigma) \\ & c(\sigma_1 + \sigma_2) = c(\sigma_1) + c(\sigma_2) \end{aligned}$$

$$\begin{aligned} c(\rho):\quad & c(s) = s \\ & c(\rho_1\rho_2) = \begin{cases} s_{\rho_1\rho_2} & \text{if } \rho_1 \text{ and } \rho_2 \text{ are abstractable} \\ c(\rho_1)c(\rho_2) & \text{otherwise} \end{cases} \\ & c(\rho_1 + \rho_2) = \begin{cases} s_{\rho_1+\rho_2} & \text{if } \rho_1 + \rho_2 \text{ is abstractable} \\ c(\rho_1) + c(\rho_2) & \text{otherwise} \end{cases} \\ & c(\rho^n) = \begin{cases} s_{\rho^n} & \text{if } \rho \text{ is abstractable} \\ (c(\rho))^n = c(\rho)c(\rho)\cdots c(\rho) & \text{otherwise} \end{cases} \\ & c(\rho^*) = (c(\rho))^* \end{aligned}$$

Recall that σ represents an ω-regular expression, and ρ is an ordinary regular expression. The functions $c(\sigma)$ and $c(\rho)$ are obtained for each syntactic composition defined for the construction of regular expressions σ and ρ, respectively. As an example, according to equation $c(\rho^\omega) = (c(\rho))^\omega$, in an ω-regular expression that is composed as $\sigma = (\rho)^\omega$, function c is applied to the regular expression inside ω, in this case, ρ. That is, $\sigma^c = c(\sigma) = (c(\rho))^\omega$. For an ordinary regular expression, $c(s) = s$ means that a single scenario is already a compact scenario and cannot be further compacted. Therefore, a single scenario remains the same after the compaction. As an example of the application of function c on the sequential composition of two ordinary regular expressions, i.e., $c(\rho_1\rho_2)$, consider the example regular expression $((\phi^{11} + \psi\phi^{10})^*\phi^*)^\omega$ made earlier in this chapter. Let $\rho_1 = \psi$ and $\rho_2 = \phi^{10}$. Since $\psi\phi^{10}$ is an abstractable regular expression, $c(\rho_1\rho_2) = c(\psi\phi^{10})$ returns a compact scenario which is called $s_{\psi\phi^{10}}$. Note that using the definition for $c(\rho^n)$, ϕ^{10} itself can be compacted in a scenario called $s_{\phi^{10}}$.

Generally, we use s_ρ to name a scenario that abstracts a set of scenario sequences recognized by a sub-expression ρ. Observe that $c(\sigma)$ does not do any compression except on its ordinary sub-expressions ρ, while $c(\rho)$ takes an ordinary regular expression in a recursive fashion and defines new scenarios from abstractable ones. S^c is defined by collecting all scenarios from σ^c. Note that S^c might contain the same scenarios as in S due to $c(s) = s$. For every scenario $s_\rho \in S^c$, the state matrix $\boldsymbol{G}_{s_\rho} = \boldsymbol{G}^c(s_\rho)$ represents the worst-case timing relations (between the initial and final tokens) among all sequences recognized by ρ. We compute the state matrix function $\boldsymbol{G}^c$ and the output function o^c as follows.

$$\begin{array}{ll}
\boldsymbol{G}^c(s) = \boldsymbol{G}(s) = \boldsymbol{G}_s \quad \text{if } s \in S & o^c(s) = o(s) \quad \text{if } s \in S \\
\boldsymbol{G}^c(s_{\rho_1\rho_2}) = \boldsymbol{G}^c(s_{\rho_2})\boldsymbol{G}^c(s_{\rho_1}) & o^c(s_{\rho_1\rho_2}) = o^c(s_{\rho_1}) + o^c(s_{\rho_2}) \\
\boldsymbol{G}^c(s_{\rho_1+\rho_2}) = \boldsymbol{G}^c(s_{\rho_1}) \oplus \boldsymbol{G}^c(s_{\rho_2}) & o^c(s_{\rho_1+\rho_2}) = o^c(s_{\rho_1}) = o^c(s_{\rho_2}) \\
\boldsymbol{G}^c(s_{\rho^n}) = (\boldsymbol{G}^c(s_\rho))^n & o^c(s_{\rho^n}) = n o^c(s_\rho)
\end{array}$$

Since an abstractable expression represents a language, the sequences of which produce the same number of outputs, a choice between two abstractable regular expressions ρ_1 and ρ_2 can form an abstractable expression only if they both represents languages, the sequences of which produce the same number of outputs. Therefore, $o^c(s_{\rho_1+\rho_2}) = o^c(s_{\rho_1}) = o^c(s_{\rho_2})$.

We need the following two propositions later to prove that our throughput analysis is correct. The first proposition states that the state matrix $\boldsymbol{G}^c(s_\rho)$ of any abstracted scenario s_ρ represents the worst-case state matrix among all scenarios defined by ρ, and that $o^c(s_\rho)$ represents the correct number of outputs.

Proposition 4.1. *Consider an SADF $M(S, \boldsymbol{G}, o, \sigma)$ and its compact transformation $M^c(S^c, \boldsymbol{G}^c, o^c, \sigma^c)$. The following statements hold.*

1. $\forall s_\rho \in S^c, \boldsymbol{G}^c(s_\rho) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho)} \boldsymbol{G}(\bar{s})$.
2. $\forall s_\rho \in S^c$ *and* $\forall \bar{s} \in \mathscr{L}(\rho), o(\bar{s}) = o^c(s_\rho)$.

Proof. We prove *1* by structural induction on ρ. The base case, $\rho = s$, is trivial. The induction steps are in the following.

$$\boldsymbol{G}^c(s_{\rho_1+\rho_2}) \quad = \boldsymbol{G}^c(s_{\rho_1}) \oplus \boldsymbol{G}^c(s_{\rho_2}) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho_1)} \boldsymbol{G}(\bar{s}) \oplus \bigoplus_{\bar{s}\in\mathscr{L}(\rho_2)} \boldsymbol{G}(\bar{s})$$

$$\begin{aligned}
&= \bigoplus_{\bar{s}\in\mathscr{L}(\rho_1)\cup\mathscr{L}(\rho_2)} \boldsymbol{G}(\bar{s}) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho_1+\rho_2)} \boldsymbol{G}(\bar{s}).\\
\boldsymbol{G}^c(s_{\rho_1\rho_2}) &= \boldsymbol{G}^c(s_{\rho_2})\boldsymbol{G}^c(s_{\rho_1}) = \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1),\bar{s}_2\in\mathscr{L}(\rho_2)} \boldsymbol{G}(\bar{s}_2)\boldsymbol{G}(\bar{s}_1)\\
&= \bigoplus_{\bar{s}_1\bar{s}_2\in\mathscr{L}(\rho_1\rho_2)} \boldsymbol{G}(\bar{s}_1\bar{s}_2).\\
\boldsymbol{G}^c(s_{\rho^n}) &= \boldsymbol{G}^c(s_\rho)\boldsymbol{G}^c(s_{\rho^{n-1}}) = \boldsymbol{G}^c(s_\rho)\boldsymbol{G}^c(s_\rho)\boldsymbol{G}^c(s_{\rho^{n-2}}) = \cdots = (\boldsymbol{G}^c(s_\rho))^n.
\end{aligned}$$

The proof of *2* is straightforward. □

The second proposition states that the SADF transformation in Definition 4.2 is conservative with respect to timing behaviour.

Proposition 4.2. *Consider an SADF $M(S,\boldsymbol{G},o,\sigma)$ and its compact transformation $M^c(S^c,\boldsymbol{G}^c,o^c,\sigma^c)$. Let ρ be a sub-expression in σ and ρ^c be the compact transformation of it, i.e. $\rho^c = c(\rho)$. The following statement holds.*

$$\forall \bar{s}\in\mathscr{L}(\rho),\ \exists\ \bar{s}^c\in\mathscr{L}(\rho^c)\ \textit{s.t.}\ \boldsymbol{G}(\bar{s})\le\boldsymbol{G}(\bar{s}^c)\ \textit{and}\ o(\bar{s})=o(\bar{s}^c).$$

Proof. We prove this by structural induction on ρ. The base case is when ρ is abstractable. Then for any given $\bar{s}$ we choose $\bar{s}^c = s_\rho$. According to Proposition 4.1,

$$\boldsymbol{G}(\bar{s}^c) = \boldsymbol{G}^c(s_\rho) = \bigoplus_{\bar{s}'\in\mathscr{L}(\rho)} \boldsymbol{G}(\bar{s}') \ge \boldsymbol{G}(\bar{s}) \text{ and } o(\bar{s}^c) = o^c(s_\rho) = o(\bar{s}).$$

The induction steps are explained in the following.

- In case $\rho = \rho_1\rho_2$, let $\bar{s} = \bar{s}_1\bar{s}_2$ be such that $\bar{s}_1 \in \mathscr{L}(\rho_1)$ and $\bar{s}_2 \in \mathscr{L}(\rho_2)$. According to the induction hypothesis, $\bar{s}_1^c$ and $\bar{s}_2^c$ exist such that $\boldsymbol{G}(\bar{s}_1)\le\boldsymbol{G}(\bar{s}_1^c)$, $\boldsymbol{G}(\bar{s}_2)\le\boldsymbol{G}(\bar{s}_2^c)$, $o(\bar{s}_1)=o(\bar{s}_1^c)$, and $o(\bar{s}_2)=o(\bar{s}_2^c)$. We let $\bar{s}^c = \bar{s}_1^c\bar{s}_2^c$. Then we have $\boldsymbol{G}(\bar{s}) = \boldsymbol{G}(\bar{s}_2)\boldsymbol{G}(\bar{s}_1) \le \boldsymbol{G}(\bar{s}_2^c)\boldsymbol{G}(\bar{s}_1^c) = \boldsymbol{G}(\bar{s}^c)$. Similarly we have $o(\bar{s}) = o(\bar{s}_2)+o(\bar{s}_1) = o(\bar{s}_2^c)+o(\bar{s}_1^c) = o(\bar{s}^c)$.
- In case $\rho = \rho_1+\rho_2$, then $\bar{s}\in\mathscr{L}(\rho_1)\cup\mathscr{L}(\rho_2)$. Without loss of generality let $\bar{s}\in\mathscr{L}(\rho_1)$. According to the induction hypothesis, $\bar{s}^c$ exist such that $\boldsymbol{G}(\bar{s})\le\boldsymbol{G}(\bar{s}^c)$ and $o(\bar{s})=o(\bar{s}^c)$.
- In case $\rho = (\rho_b)^*$, let $\bar{s} = \bar{s}_1\bar{s}_2\cdots\bar{s}_n$ such that $\bar{s}_i\in\mathscr{L}(\rho_b)$. According to the induction hypothesis, there exist $\bar{s}_i^c$ such that $\boldsymbol{G}(\bar{s}_i)\le\boldsymbol{G}(\bar{s}_i^c)$ and $o(\bar{s}_i)=o(\bar{s}_i^c)$. We let $\bar{s}^c = \bar{s}_1^c\bar{s}_2^c\cdots\bar{s}_n^c$. Then we have $\boldsymbol{G}(\bar{s}) = \boldsymbol{G}(\bar{s}_n)\cdots\boldsymbol{G}(\bar{s}_2)\boldsymbol{G}(\bar{s}_1) \le \boldsymbol{G}(\bar{s}_n^c)\cdots\boldsymbol{G}(\bar{s}_2^c)\boldsymbol{G}(\bar{s}_1^c) = \boldsymbol{G}(\bar{s}^c)$ and $o(\bar{s}) = o(\bar{s}_n)\cdots o(\bar{s}_2)o(\bar{s}_1) = o(\bar{s}_n^c)\cdots o(\bar{s}_2^c)o(\bar{s}_1^c) = o(\bar{s}^c)$

- In case $\rho = (\rho_b)^n$, the proof is similar to the previous case.

$\square$

We can now prove the following result.

Theorem 4.1. *An SADF $M(S, \boldsymbol{G}, o, \sigma)$ and its compact transformation $M^c(S^c, \boldsymbol{G}^c, o^c, \sigma^c)$ have the same throughput.*

Proof. We divide the proof into two parts, first proving that for every scenario sequence $\bar{s} \in \mathscr{L}(\sigma)$ there exist a scenario sequence $\bar{s}^c \in \mathscr{L}(\sigma^c)$ such that $\tau(\bar{s}^c) \leq \tau(\bar{s})$ and then proving that for every $\bar{s}^c$ there exists an $\bar{s}$ such that $\tau(\bar{s}) \leq \tau(\bar{s}^c)$. We prove the first part by structural induction on σ.

- For the base case i.e. in case $\sigma = \rho^\omega$, let $\bar{s} = \bar{t}_1\bar{t}_2\bar{t}_3\cdots$ such that $\bar{t}_i \in \mathscr{L}(\rho)$. According to Proposition 4.2, for any $\bar{t}_i$, there exists a sequence $\bar{t}_i^c$ such that $\boldsymbol{G}(\bar{t}_i) \leq \boldsymbol{G}(\bar{t}_i^c)$ and $o(\bar{t}_i) = o(\bar{t}_i^c)$. We let $\bar{s}^c = \bar{t}_1^c\bar{t}_2^c\bar{t}_3^c\cdots$. To show that $Thr(\bar{s}^c) \leq Thr(\bar{s})$ let $\bar{s}_i = \bar{t}_1\bar{t}_2\cdots\bar{t}_i$ and $\bar{s}_i^c = \bar{t}_1^c\bar{t}_2^c\cdots\bar{t}_i^c$. We know that for any i, $\boldsymbol{G}(\bar{s}_i) \leq \boldsymbol{G}(\bar{s}_i^c)$ and $o(\bar{s}_i) = o(\bar{s}_i^c)$. Hence we have

$$Thr(\bar{s}) = \liminf_{i\to\infty} \frac{\Sigma_{n=1}^{i} o(\bar{s}_i)}{\|\boldsymbol{G}(\bar{s}_i)\boldsymbol{\gamma}_0\|} \geq \liminf_{i\to\infty} \frac{\Sigma_{n=1}^{i} o(\bar{s}_i^c)}{\|\boldsymbol{G}(\bar{s}_i^c)\boldsymbol{\gamma}_0\|} = Thr(\bar{s}^c).$$

- In case $\sigma = \rho\sigma_2$, let $\bar{s} = \bar{s}_1\bar{s}_2$ such that $\bar{s}_1 = \mathscr{L}(\rho)$ and $\bar{s}_2 \in \mathscr{L}(\sigma_2)$. Note that the throughput of $\bar{s}$ is equal to the throughput of $\bar{s}_2$, since $\bar{s}_1$ is only a finite prefix of $\bar{s}$ and it does not affect the throughput. According to the induction hypothesis there exists a sequence $\bar{s}_2^c$ such that $Thr(\bar{s}_2) \geq Thr(\bar{s}_2^c)$. Hence we have $Thr(\bar{s}) = Thr(\bar{s}_2) \geq Thr(\bar{s}_2^c)$.

- In case $\sigma = \sigma_1 + \sigma_2$. Consider σ_1^c and σ_2^c such that $c(\sigma_1 + \sigma_2) = c(\sigma_1) + c(\sigma_2) = \sigma_1^c + \sigma_2^c$. Without loss of generality, let $\bar{s}$ belong to $\mathscr{L}(\sigma_1)$. According to the induction hypothesis there exists $\bar{s}_1^c \in \mathscr{L}(\sigma_1^c)$ such that $Thr(\bar{s}) \geq Thr(\bar{s}_1^c)$.

For the second part of the proof let $\bar{s}^c = s_1^c s_2^c s_3^c \cdots$ where $s_k^c \in S^c$. According to [62], for any scenario sequence $\bar{s}^c$ there exists a periodic scenario sequence $\bar{s}_o^c = (s_1^c s_2^c \cdots s_n^c)^\omega$ for some n, such that $Thr(\bar{s}_o^c) \leq Thr(\bar{s}^c)$. Therefore, to prove the second part, it suffices to show that there exist a sequence $\bar{s}$ such that $Thr(\bar{s}) \leq Thr(\bar{s}_o^c)$. The throughput of a periodic sequence is limited by the critical cycle in a graph ((max, +) automaton) that encodes the dependencies between all initial and final tokens in the sequence (See Section 2.8). The nodes in this graph represent the initial/final tokens in each of the scenarios. The state matrices are used to connect the nodes to each other. For every

ALGORITHM 1: Compute the throughput of an SADF model

```
  Input: An SADF $F\ (S, g, r, i, f, o, L)$
  Output: throughput $Thr(F)$
1 $M(S, \boldsymbol{G}, o, \sigma) \leftarrow$ convertToMatrixRepresentation($F$);
2 $M^c(S^c, \boldsymbol{G}^c, o^c, \sigma^c) \leftarrow$ transformToCompactSADF($M$);
3 $A^c \leftarrow$ convertToFSM($\sigma^c$);
4 $Thr(F) \leftarrow$ computeThrFromMatrixRepresentation($S^c, \boldsymbol{G}^c, o^c, A^c$);
```

non $-\infty$ element in the matrix there exists an edge between the corresponding initial/final tokens, labelled with the value of that element and with the number of outputs produced in that scenario. The critical cycle is composed of critical edges and has the lowest output/time ratio.

The critical cycle of the graph for $\bar{s}^c_o = (s^c_1 s^c_2 \cdots s^c_n)^\omega$ specifies a pair $((i,j)_k, o_k)$ for each scenario s^c_k, such that $[\boldsymbol{G}(s^c_k)]_{ij}$ is the critical element (the element corresponding to the critical edge) in the state matrix of s^c_k, and $o_k = o(s^c_k)$. Let ρ_k be the sub-expression abstracted by s^c_k. It follows from Proposition 4.1 that $\bar{s}_k = \mathrm{argmax}_{\bar{t}_k \in \mathscr{L}(\rho_k)} [\boldsymbol{G}(\bar{t}_k)]_{ij}$ exists such that $[\boldsymbol{G}(\bar{s}_k)]_{ij} = [\boldsymbol{G}(s^c_k)]_{ij}$ and $o(\bar{s}_k) = o_k$. If we let $\bar{s} = (\bar{s}_1 \bar{s}_2 \cdots \bar{s}_n)^\omega$, there exists a cycle in the graph of $\bar{s}$ with the same cycle ratio as $\bar{s}^c_o$. Hence, the graph of $\bar{s}$ has a critical cycle which has at most the same ratio as $\bar{s}^c$, i.e. $Thr(\bar{s}) \leq Thr(\bar{s}^c_o)$. Note that if the periodic sub-sequence $\bar{s}^c_o$ is in the language of a sub-expression in σ^c, then $\bar{s}$ belongs to the language of a sub-expression in σ. $\square$

We provide an algorithm for the throughput analysis of SADF models. Algorithm 1, first converts the given SADF to its matrix-level representation using Definition 4.1 (Line 1). In Line 2, it performs the SADF transformation defined by Definition 4.2. Then it converts the regular expression of the transformed SADF to a compact ϵ-free FSM (Line 3). The final analysis is preformed on the transformed FSM-SADF in Line 4. Note that since in our examples we associate rewards with integer numbers of outputs produced in scenarios, the rewards are always integer-valued. Nevertheless, our throughput analysis is also applicable to real-valued rewards.

Next we discuss the complexity of this algorithm. We assume a regular expression σ is given by some syntax tree t_σ. The leaves of the syntax tree are labelled by scenarios $s \in S$, and the inner nodes are either a binary operator labelled $+$ (representing choices) or $\cdot$ (representing sequential compositions), or they are a unary operator and labelled by $*, \omega$ or a natural number $\boldsymbol{n}$ (for the constant repetition). Computing $c(\sigma)$, $\boldsymbol{G}^c$ and o^c can be done by a reversed preorder traversal of t_σ. The time complexity of computing $c(\sigma)$ and o^c is linear in the size (the number of nodes) of t_σ. For a $+$ or $\cdot$ node, computing $\boldsymbol{G}^c$ has a constant complexity and for an n node it has a logarithmic complexity in the value of n. It is common to use Polish Notation (PN) to represent σ. If we represent the value n with $\log(n)$ digits (binary or decimal) and use a character to represent each scenario, then the complexity of the transformation by Definition 4.2 is linear in the length of the representation of σ.

For deterministic cases of SADF i.e. SDF and CSDF, the transformed SADF will always have a regular expression that represents indefinite repetition of a single scenario. Such a regular expression can be transformed to an ϵ-free FSM with only one state, in constant time. Therefore the throughput computation for deterministic cases of SADF is always linear in the length of the regular expression. For non-deterministic cases of SADF, if for all sub-expressions ρ^n in σ, ρ are abstractable, then the transformed SADF will have a conventional regular expression (without a repetition construct). A sequential algorithm is provided by Hagenah et al. for conversion from a conventional regular expression to an ϵ-free FSM that takes $\mathcal{O}(k \log^2(k))$ time, where k is the length of the regular expression [63]. If in the worst-case, the length of the regular expression for the transformed SADF is equal to the length of the regular expression in the given SADF, then the throughput computation will have a polynomial complexity. For sub-expressions ρ^n that are not abstractable, the conversion to FSM is pseudo-polynomial in n, which makes the throughput computations pseudo-polynomial.

4.2 Latency Analysis

This section provides a compositional latency analysis for an SADF, of which the language is given by a regular expression. We first provide a latency analysis for a set of finite SADF scenario sequences given by an ordinary regular expression, as a means to compute the latency of the set of infinite scenario sequences of an SADF. A naive method to compute the latency of a finite scenario sequence is to compute the production times of all outputs and use Eq. 2.3 where k has an upper bound. Consider again the scenario sequence for producing a 9×11 frame in the convolution filter example in Section 3.1, i.e., $(ri)^{18}((ri)^2(cm)^7(ro)^2)^9(ro)^{18}$. Scenario cm is the first scenario that appears in the sequence that produces output. This scenario follows the scenario sequence that corresponds to the frame rush-in and the line rush-in phases. To proceed, we provide the output matrices: $\boldsymbol{H}_{cm} = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix}$ and $\boldsymbol{H}_{ro} = \begin{bmatrix} -\infty & 9 & 2 \end{bmatrix}$. Assuming the initial state vector $\begin{bmatrix} 0 & 0 \end{bmatrix}^T$, which is denoted by $\mathbf{0}$, the outputs are produced at

$$\begin{aligned}
p_0 &= \boldsymbol{H}_{cm}\boldsymbol{G}_{ri}^{20}\mathbf{0} = 53, \\
p_1 &= \boldsymbol{H}_{cm}\boldsymbol{G}_{cm}\boldsymbol{G}_{ri}^{20}\mathbf{0} = 57, \\
&\vdots \\
p_{98} &= \boldsymbol{H}_{ro}\boldsymbol{G}_{ro}^{17}\left(\boldsymbol{G}_{ro}^{2}\boldsymbol{G}_{cm}^{7}\boldsymbol{G}_{ri}^{2}\right)^{9}\boldsymbol{G}_{ri}^{18}\mathbf{0} = 440.
\end{aligned}$$

Using Eq. 2.3, the latency of producing pixels in one frame relative to period 5 is 53. We derived the period 5 from the throughput of $99/432 > 0.2 = 1/5$ computed in Section 3.1. According to Figure 3.3 the first output determines the latency. In the beginning of the execution the convolution filter goes through the frame and line rush-in phases where there are no outputs produced. However, after getting past theses phases, outputs are produced more frequently. Assuming that the first start time of the external actor (see the definition of latency in Section 2.6) is 53, the next outputs will be available before the next starts of the external actor with $\mu = 5$.

Although computing the production times of all outputs one by one is a solution to determine the latency of a sequence, it would scale only linearly in the total number of scenarios in the sequence. Therefore, we provide a compositional method to efficiently compute the latency of a finite SADF scenario sequence. In this section $\bar{s}$ denotes a finite sequence unless stated otherwise. For readability, we leave the period μ implicit and write $\lambda(\bar{s}, \boldsymbol{\gamma}_0)$ instead of $\lambda(\bar{s}, \boldsymbol{\gamma}_0, \mu)$. For all examples in this section we assume $\mu = 5$ and $\boldsymbol{\gamma}_0 = \mathbf{0}$.

A scenario sequence is either a single scenario or the sequential composition of two sub-sequences. In case $\bar{s}$ is a single scenario $\bar{s} = s$ that produces $o(s) > 0$ outputs during its execution, we can use the output matrix to capture the output production times and use them to compute the latency. For instance, for scenario cm we can obtain the latency as follows.

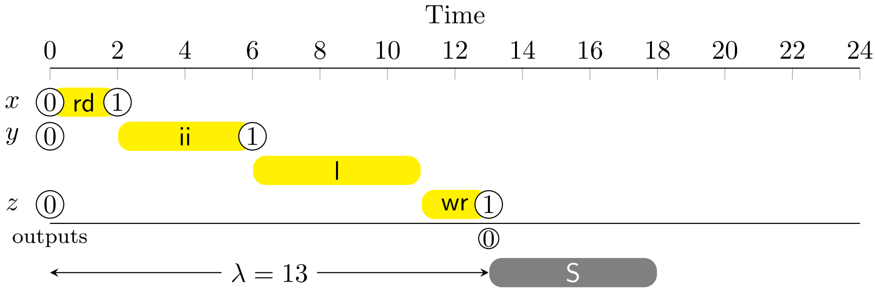
$$\lambda(cm, \boldsymbol{\gamma}_0) = p_0 - 5 \times 0 = \boldsymbol{H}_{cm}\mathbf{0} - 0 = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix}\mathbf{0} = 13$$

In general, for a scenario s, the k^{th} element of $\boldsymbol{H}_s\boldsymbol{\gamma}_0$ corresponds to the production time of the k^{th} output in the scenario. Therefore, we can obtain the latency as follows.

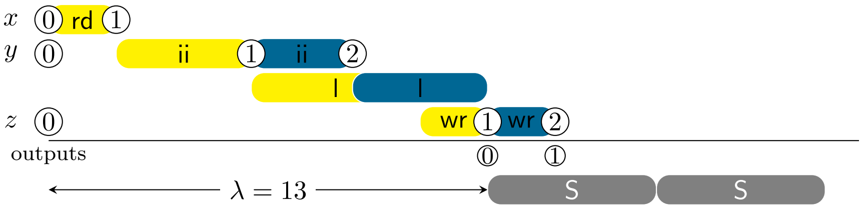
$$\lambda(s, \boldsymbol{\gamma}_0) = \max_{0 \le k < o(s)} p_k - \mu k = \max_{0 \le k < o(s)} (\boldsymbol{H}_s\boldsymbol{\gamma}_0)_k - \mu k$$

Alternatively, we can define the *compensation vector*,

$$\boldsymbol{\mu}(s) = \begin{bmatrix} 0 & -\mu & -2\mu & \cdots & -(o(s)-1)\mu \end{bmatrix},$$



(a) cm



(b) $(cm)(ro)$

Figure 4.4: Execution traces for two example scenario sequences.

that captures the number of periods occurring till the corresponding output. Then we compute the latency of s in matrix form as

$$\lambda(s, \boldsymbol{\gamma}_0) = \boldsymbol{\mu}(s)\boldsymbol{H}_s\boldsymbol{\gamma}_0 = \left[\boldsymbol{H}_s^T\boldsymbol{\mu}^T(s)\right]^T\boldsymbol{\gamma}. \tag{4.3}$$

Observe that Eq. 4.3 uses the following vector.

$$\boldsymbol{\lambda}(s) = \boldsymbol{H}_s^T\boldsymbol{\mu}^T(s) \tag{4.4}$$

We refer to $\boldsymbol{\lambda}(s)$ as the *latency vector* of scenario s. The latency vector $\boldsymbol{\lambda}(s)$ is a column vector that expresses the timing relations between the availability times of initial tokens and the production times of outputs relative to the required period μ, during the execution of the scenario. According to Eq. 4.3 the latency equals the inner product of the latency vector and the initial state vector, i.e.,

$$\lambda(s, \boldsymbol{\gamma}_0) = \boldsymbol{\lambda}^T(s)\boldsymbol{\gamma}_0. \tag{4.5}$$

Let's consider an example of this case in which $s = cm$. Using the above equation, the latency is computed as follows.

$$\boldsymbol{\lambda}(cm) = \boldsymbol{H}_{cm}^T\boldsymbol{\mu}^T(cm) = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix}^T \begin{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix}^T$$

$$\lambda(cm, \mathbf{0}) = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T = 13$$

Figure 4.4a shows the execution trace of *cm* followed by the execution of the external actor S that needs this output. The output is produced at 13. Therefore the latency is 13 since the start of actor S cannot happen earlier.

Now let $\bar{s}$ be the sequential composition of two scenario sequences, i.e., $\bar{s} = \bar{s}_1\bar{s}_2$. We work out an example where $\bar{s}_1 = cm$ and $\bar{s}_2 = ro$ i.e. $\bar{s} = (cm)(ro)$. We need to compute the maximum of the latencies of both scenarios to obtain the latency of $\bar{s}$. The latency of *cm* is 13 (using Eq. 4.3). Since *ro* follows *cm*, the initial state vector for *ro* is $\boldsymbol{\gamma}_1 = \boldsymbol{G}_{cm}\boldsymbol{\gamma}_0 = \begin{bmatrix} 2 & 6 & 13 \end{bmatrix}^T$. This can be observed in Figure 4.4b by circles marked with number one in the x, y, z rows. To compute the latency of *ro* as the second scenario in the sequence $(cm)(ro)$, we also need to account for the output produced in *cm*, because the first firing of the external actor S consumes this output. Therefore, we need to compute the latency of *ro* with the state vector $\boldsymbol{\gamma}_1$ and subtract 5 from the result to compensate for the output produced by *cm*. Using Eq. 4.3, the latency of *ro* with $\boldsymbol{\gamma}_1$ is 15. By subtracting 5 from 15 we obtain the latency of 10 for *ro*. The latency of the entire sequence is the maximum of 13 and 10, which is 13.

In general, in a sequence s_1s_2, to compute the latency of the second scenario s_2, we need to compute the latency of s_2 with the state vector $\boldsymbol{\gamma}_1$ and subtract $o(s_1)\mu$ from the result to compensate for all outputs already produced in s_1, i.e., we compute $\lambda(s_2, \boldsymbol{\gamma}_1) - o(s_1)\mu$. Alternatively, and equivalently, we can consider the compensation term $-o(s_1)\mu$ in computing the latency of the second scenario by normalizing the state vector $\boldsymbol{\gamma}_1$ as follows.

$$\hat{\boldsymbol{\gamma}}_1 = \boldsymbol{\gamma}_1 - o(s_1)\mu = \boldsymbol{G}_{s_1}\boldsymbol{\gamma}_0 - o(s_1)\mu = (\boldsymbol{G}_{s_1} - o(s_1)\mu)\boldsymbol{\gamma}_0$$

We normalized the state vector by multiplying the initial state vector by the following matrix.

$$\hat{\boldsymbol{G}}(s) = \boldsymbol{G}_s - o(s)\mu \tag{4.6}$$

We define the *normalized state matrix* $\hat{\boldsymbol{G}}(s)$ as the state matrix of scenario s that is normalized for its outputs. Note that the normalization depends on μ. Now the latency of the sequence s_1s_2 can be obtained as follows, assuming that $\boldsymbol{\lambda}(s_1)$ and $\boldsymbol{\lambda}(s_2)$ are the latency vectors of s_1 and s_2, respectively.

$$\begin{aligned}\lambda(s_1s_2, \boldsymbol{\gamma}_0) =& \lambda(s_1, \boldsymbol{\gamma}_0) \oplus \lambda(s_2, \hat{\boldsymbol{G}}(s_1)\boldsymbol{\gamma}_0) = \boldsymbol{\lambda}^T(s_1)\boldsymbol{\gamma}_0 \oplus \boldsymbol{\lambda}^T(s_2)\hat{\boldsymbol{G}}(s_1)\boldsymbol{\gamma}_0 \\ =& \left[\boldsymbol{\lambda}^T(s_1) \oplus \boldsymbol{\lambda}^T(s_2)\hat{\boldsymbol{G}}(s_1)\right]\boldsymbol{\gamma}_0 \end{aligned} \tag{4.7}$$

Hence the latency vector for $\bar{s} = s_1s_2$ is

$$\boldsymbol{\lambda}(s_1s_2) = \left[\boldsymbol{\lambda}^T(s_1) \oplus \boldsymbol{\lambda}^T(s_2)\hat{\boldsymbol{G}}(s_1)\right]^T = \boldsymbol{\lambda}(s_1) \oplus \hat{\boldsymbol{G}}^T(s_1)\boldsymbol{\lambda}(s_2). \tag{4.8}$$

Again considering the sequence $(cm)(ro)$, we compute the normalized state matrix of cm (Eq. 4.6) and the latency vector of ro (Eq. 4.4) as follows.

$$
\begin{aligned}
\hat{\boldsymbol{G}}(cm) \quad & = \boldsymbol{G}_{cm} - o(cm) \times \mu = \begin{bmatrix} 2 & -\infty & -\infty \\ 6 & 4 & -\infty \\ 13 & 11 & 2 \end{bmatrix} - 1 \times 5 \\
& = \begin{bmatrix} -3 & -\infty & -\infty \\ 1 & -1 & -\infty \\ 8 & 6 & -3 \end{bmatrix} \\
\boldsymbol{\lambda}(ro) = \boldsymbol{H}_{ro}^{T}\boldsymbol{\mu}^{T}(ro) \quad & = \begin{bmatrix} -\infty & 9 & 2 \end{bmatrix}^{T} \begin{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} -\infty & 9 & 2 \end{bmatrix}^{T}
\end{aligned}
$$

The latency according to Eq. 4.7 is computed as follows.

$$
\lambda((cm)(ro), \boldsymbol{\gamma}_0) =
$$

$$
\left[\begin{bmatrix} 13 & 11 & 2 \end{bmatrix} \oplus \begin{bmatrix} -\infty & 9 & 2 \end{bmatrix} \begin{bmatrix} -3 & -\infty & -\infty \\ 1 & -1 & -\infty \\ 8 & 6 & -3 \end{bmatrix} \right] \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 13
$$

From Figure 4.4b, we can obtain the output production times and compute the latency according to Eq. 2.3 as $\lambda((cm)(ro), \mathbf{0}) = (13 - 0) \oplus (15 - 5) = 13$. This confirms the result computed earlier from Eq. 4.7.

Note that using Eq. 4.7, the latency of a sequence composed of two scenarios is obtained by first computing a vector that is then multiplied by the initial state vector. This vector generalizes the concept of the latency vector of a scenario to a sequence of scenarios. In general, for any finite sequence $\bar{s} = s_1 \cdots s_n$ the latency vector $\boldsymbol{\lambda}(\bar{s})$ is defined as follows.

$$
\boldsymbol{\lambda}(\bar{s}) = \bigoplus_{0<k\leq n} \left(\hat{\boldsymbol{G}}^{T}(s_1 \cdots s_{k-1}) \boldsymbol{\lambda}(s_k) \right) \tag{4.9}
$$

In Eq. 4.9, the normalized state matrix $\hat{\boldsymbol{G}}(\bar{s})$ of a sequence $\bar{s}$ is the state matrix of the sequence that accounts for all outputs produced in the sequence, i.e.,

$$
\hat{\boldsymbol{G}}(\bar{s}) = \boldsymbol{G}(\bar{s}) - o(\bar{s}). \tag{4.10}
$$

In Eq. 4.9, for $k = 1$, $s_1 \cdots s_{k-1}$ represents the empty scenario sequence ϵ and $\hat{\boldsymbol{G}}(\epsilon)$ is defined to be the identity (max, +) matrix $\mathcal{I}$. We show that the latency of a sequence equals the inner product of the latency vector of the sequence as defined in Eq. 4.9, and the initial state vector.

Proposition 4.3. *The latency of any finite scenario sequence $\bar{s} = s_1 \cdots s_n$ is equal to the inner product of its latency vector and the initial state vector, i.e.,*

$$\lambda(\bar{s}, \boldsymbol{\gamma}_0) = \boldsymbol{\lambda}^T(\bar{s})\boldsymbol{\gamma}_0. \tag{4.11}$$

Proof. First we distribute the max term in the definition of the latency (Eq. 2.3) into n max terms each of which corresponds to the latency of outputs produced during a single scenario in the sequence.

$$\begin{aligned}
\lambda(\bar{s}, \boldsymbol{\gamma}_0) &= \max_{0 \le k < o(\bar{s})} (p_k - k\mu) \\
&= \max_{0 \le k < o(s_1)} (p_k - k\mu) \oplus \max_{o(s_1) \le k < o(s_1 s_2)} (p_k - k\mu) \oplus \cdots \\
&\quad \oplus \max_{o(s_1 \cdots s_{n-1}) \le k < o(\bar{s})} (p_k - k\mu) \\
&= \max_{0 \le k < o(s_1)} (p_k - k\mu) \oplus \max_{0 \le k < o(s_2)} (p_{k+o(s_1)} - k\mu) - o(s_1)\mu \oplus \cdots \\
&\quad \oplus \max_{0 \le k < o(s_n)} (p_{k+o(s_1 \cdots s_{n-1})} - k\mu) - o(s_1 \cdots s_{n-1})\mu
\end{aligned} \tag{4.12}$$

Recall that the production times of outputs in scenario s_i within a scenario sequence, can be computed in vector $\boldsymbol{H}_{s_i}\boldsymbol{G}(s_1 \cdots s_{i-1})\boldsymbol{\gamma}_0$. Using the compensation matrix $\boldsymbol{\mu}(s)$ we can rewrite each of the max terms in the following form.

$$\begin{aligned}
&\max_{0 \le k < o(s_i)} (p_{k+o(s_1 \cdots s_{i-1})} - k\mu) - o(s_1 \cdots s_{i-1})\mu \\
&= \big[\ p_{o(s_1 \cdots s_{i-1})} - 0 \quad p_{o(s_1 \cdots s_{i-1})+1} - \mu \quad p_{o(s_1 \cdots s_{i-1})+2} - 2\mu \quad \cdots \\
&\quad p_{o(s_1 \cdots s_{i-1})+o(s_i)-1} - (o(s_i) - 1)\mu\ \big] - o(s_1 \cdots s_{i-1})\mu \\
&= \boldsymbol{\mu}(s_i)\boldsymbol{H}_{s_i}\boldsymbol{G}(s_1 \cdots s_{i-1})\boldsymbol{\gamma}_0 - o(s_1 \cdots s_{i-1})\mu = \boldsymbol{\mu}(s_i)\boldsymbol{H}_{s_i}\hat{\boldsymbol{G}}(s_1 \cdots s_{i-1})\boldsymbol{\gamma}_0
\end{aligned}$$

By substituting the max terms in Eq. 4.12 with their matrix forms above, we obtain

$$\begin{aligned}
\lambda(\bar{s}, \boldsymbol{\gamma}_0) &= \left[\bigoplus_{0 < k \le n} \Big(\boldsymbol{\mu}(s_k)\boldsymbol{H}_{s_k}\hat{\boldsymbol{G}}(s_1 \cdots s_{k-1}) \Big) \right] \boldsymbol{\gamma}_0 \\
&= \left[\bigoplus_{0 < k \le n} \Big(\boldsymbol{\lambda}^T(s_k)\hat{\boldsymbol{G}}(s_1 \cdots s_{k-1}) \Big) \right] \boldsymbol{\gamma}_0 = \boldsymbol{\lambda}^T(\bar{s})\boldsymbol{\gamma}_0.
\end{aligned}$$

□

Now we show how to compositionally compute the latency vector of a sequence composed of two subsequences. We show that the latency vector of $\bar{s} = \bar{s}_1\bar{s}_2$ can be computed from the latency vectors of $\bar{s}_1$ and $\bar{s}_2$ similar to the sequential composition of two scenarios (Eq. 4.8) as follows.

$$\boldsymbol{\lambda}(\bar{s}_1\bar{s}_2) = \boldsymbol{\lambda}(\bar{s}_1) \oplus \hat{\boldsymbol{G}}^T(\bar{s}_1)\boldsymbol{\lambda}(\bar{s}_2) \tag{4.13}$$

Proposition 4.4. *Consider a finite sequence* $\bar{s} = s_1 \cdots s_m s_{m+1} \cdots s_n$. *Let* $\bar{s}_1 = s_1 \cdots s_m$ *and* $\bar{s}_2 = s_{m+1} \cdots s_n$. *The latency vector* $\boldsymbol{\lambda}(\bar{s})$ *can be computed by Eq. 4.13.*

Proof.

$$\begin{aligned}
&\boldsymbol{\lambda}^T(\bar{s}) = \boldsymbol{\lambda}^T(s_1 \cdots s_m s_{m+1} \cdots s_n)\\
&= \left[\bigoplus_{0<k\le m} \boldsymbol{\lambda}^T(s_k)\hat{\boldsymbol{G}}(s_1 \cdots s_{k-1})\right] \oplus \left[\bigoplus_{m<k\le n} \boldsymbol{\lambda}^T(s_k)\hat{\boldsymbol{G}}(s_1 \cdots s_{k-1})\right]\\
&= \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\le n} \boldsymbol{\lambda}^T(s_k)\boldsymbol{G}(s_1 \cdots s_{k-1}) - o(s_1 \cdots s_{k-1})\right]\\
&= \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\le n} \boldsymbol{\lambda}^T(s_k)\boldsymbol{G}(s_{m+1} \cdots s_{k-1})\boldsymbol{G}(s_1 \cdots s_m)\right.\\
&\qquad\qquad \left. - o(s_{m+1} \cdots s_{k-1}) - o(s_1 \cdots s_m)\right]\\
&= \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\le n} \boldsymbol{\lambda}^T(s_k)\left[\boldsymbol{G}(s_{m+1} \cdots s_{k-1}) - o(s_{m+1} \cdots s_{k-1})\right]\right.\\
&\qquad\qquad \left. \left[\boldsymbol{G}(s_1 \cdots s_m) - o(s_1 \cdots s_m)\right]\right]\\
&= \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\le n} \boldsymbol{\lambda}^T(s_k)\boldsymbol{G}(s_{m+1} \cdots s_{k-1}) - o(s_{m+1} \cdots s_{k-1})\right]\\
&\qquad\qquad \left[\boldsymbol{G}(s_1 \cdots s_m) - o(s_1 \cdots s_m)\right]\\
&= \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\le n} \boldsymbol{\lambda}^T(s_k)\hat{\boldsymbol{G}}(s_m \cdots s_{k-1})\right]\hat{\boldsymbol{G}}(\bar{s}_1) = \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \boldsymbol{\lambda}^T(\bar{s}_2)\hat{\boldsymbol{G}}(\bar{s}_1)
\end{aligned}$$

By taking the transposition of both sides of the above equation we obtain Eq. 4.13. □

Next we show that we can use Eq. 4.13 to efficiently compute the latency of repetitive sequences. To compute the latency vector of a finite sequence that repeats n times i.e. $\bar{s}^n$, we can recursively use Eq. 4.13 as follows.

$$\begin{aligned}\boldsymbol{\lambda}(\bar{s}^n) &= \boldsymbol{\lambda}(\bar{s}\bar{s}^{n-1}) = \boldsymbol{\lambda}(\bar{s}) \oplus \hat{\boldsymbol{G}}^{T}(\bar{s})\boldsymbol{\lambda}(\bar{s}^{n-1}) \\ &= \boldsymbol{\lambda}(\bar{s}) \oplus \hat{\boldsymbol{G}}^{T}(\bar{s}) \left(\boldsymbol{\lambda}(\bar{s}) \oplus \hat{\boldsymbol{G}}^{T}(\bar{s})\boldsymbol{\lambda}(\bar{s}^{n-2})\right) \\ &= \boldsymbol{\lambda}(\bar{s}) \oplus \hat{\boldsymbol{G}}^{T}(\bar{s})\boldsymbol{\lambda}(\bar{s}) \oplus \cdots \oplus (\hat{\boldsymbol{G}}^{T}(\bar{s}))^{n-1}\boldsymbol{\lambda}(\bar{s}) \\ &= \left(\boldsymbol{\mathcal{J}} \oplus \hat{\boldsymbol{G}}^{T}(\bar{s}) \oplus \cdots \oplus (\hat{\boldsymbol{G}}^{T}(\bar{s}))^{n-1}\right) \boldsymbol{\lambda}(\bar{s}) \\ &= \left(\bigoplus_{0\leq k<n} (\hat{\boldsymbol{G}}^{T}(\bar{s}))^{k}\right) \boldsymbol{\lambda}(\bar{s}) \end{aligned} \tag{4.14}$$

Computation of Eq. 4.14 looks linear in n; however, it can be computed with the worst-case complexity of $\mathcal{O}(\log n)$. Let's assume $\mathcal{A}(V)$ is the adjacency graph of $\boldsymbol{G}$ where V is the number of vertices of $\mathcal{A}$. When $n > V$, and there are no positive cycles in $\mathcal{A}$, computing $\bigoplus_{0\leq k<n} \boldsymbol{G}^k$ is to compute the longest paths from all nodes to all nodes in $\mathcal{A}$, which has worst-case complexity of $\mathcal{O}(V^3)$ using the Floyd-Warshall algorithm [64]. When $n > V$, and there is a positive cycle in $\mathcal{A}$, $\bigoplus_{0\leq k<n} \boldsymbol{G}^k$ can be computed with the complexity of $\mathcal{O}(\log n)$, because $\bigoplus_{0\leq k<n} \boldsymbol{G}^k = (\boldsymbol{\mathcal{J}} \oplus \boldsymbol{G})^{n-1}$ and rasing a matrix to a power can be computed with log complexity. When $n \leq V$, the complexity of computing $\bigoplus_{0\leq k<n} \boldsymbol{G}^k$ is linear in n. For multi-scale dataflow models, typically $n \gg V$ since, n indicates the number of repetitions of an execution and V indicates the number of initial tokens in the dataflow model. In the convolution example, the scenario repetitions can be as large as the number of lines or columns of the input frame, e.g., 1000 but the number of initial tokens is just 3.

We introduce an approach to compute the latency of a set of sequences based on the compositional latency analysis provided for a single sequence. Since we are dealing with a set of sequences, we first generalize the latency vector concept defined for a single sequence to a set of sequences. To obtain the worst-case latency of the set of sequences, we define the latency vector $\boldsymbol{\lambda}(\rho)$ of an ordinary regular expression ρ as the maximum of the latency vectors over all sequences in the language of ρ as follows.

$$\boldsymbol{\lambda}(\rho) = \bigoplus_{\bar{s}\in\mathcal{L}(\rho)} \boldsymbol{\lambda}(\bar{s}) \tag{4.15}$$

The goal is to perform compositional latency analysis for a language of scenario sequences similar to the way we did for a single sequence case. Recall that to compute the latency of a sequence $\bar{s}_2$ that follows sequence $\bar{s}_1$, we had to compute a normalized state matrix that accounts for the outputs produced during $\bar{s}_1$. In the case of expressions, to compute the latency of ρ_2 that follows ρ_1, we need to obtain the worst-case normalized state matrix over all sequences recognized by ρ_1. Therefore we define the normalized matrix $\hat{\boldsymbol{G}}(\rho)$ of a regular expression ρ as the maximum of normalized matrices over all sequences in the language of ρ, i.e.,

$$\hat{\boldsymbol{G}}(\rho) = \bigoplus_{\bar{s} \in \mathscr{L}(\rho)} \hat{\boldsymbol{G}}(\bar{s}). \tag{4.16}$$

Note that we can not use Eq. 4.16 to compute $\hat{\boldsymbol{G}}(\rho)$ since $\mathscr{L}(\rho)$ may be an infinite set. Therefore we provide a recursive method to compute it and later we prove that the computation corresponds to the definition. The computation is defined as follows.

$$\hat{\boldsymbol{G}}(\rho): \quad \hat{\boldsymbol{G}}(\rho_1\rho_2) = \hat{\boldsymbol{G}}(\rho_2)\hat{\boldsymbol{G}}(\rho_1) \tag{4.17}$$

$$\hat{\boldsymbol{G}}(\rho_1 + \rho_2) = \hat{\boldsymbol{G}}(\rho_1) \oplus \hat{\boldsymbol{G}}(\rho_2) \tag{4.18}$$

$$\hat{\boldsymbol{G}}(\rho^*) = \hat{\boldsymbol{G}}^*(\rho) \tag{4.19}$$

$$\hat{\boldsymbol{G}}(\rho^n) = \hat{\boldsymbol{G}}^n(\rho) \tag{4.20}$$

Now we provide a method to efficiently compute the latency vector of an ordinary regular expression ρ for every syntactic composition in an ordinary regular expression as follows.

$$\boldsymbol{\lambda}(\rho): \quad \boldsymbol{\lambda}(\rho_1\rho_2) = \boldsymbol{\lambda}(\rho_1) \oplus \hat{\boldsymbol{G}}^T(\rho_1)\boldsymbol{\lambda}(\rho_2) \tag{4.21}$$

$$\boldsymbol{\lambda}(\rho_1 + \rho_2) = \boldsymbol{\lambda}(\rho_1) \oplus \boldsymbol{\lambda}(\rho_2) \tag{4.22}$$

$$\boldsymbol{\lambda}(\rho^*) = (\hat{\boldsymbol{G}}^*(\rho))^T\boldsymbol{\lambda}(\rho) \tag{4.23}$$

$$\boldsymbol{\lambda}(\rho^n) = \bigoplus_{0 \le k < n} (\hat{\boldsymbol{G}}^T(\rho))^k\boldsymbol{\lambda}(\rho) \tag{4.24}$$

Note that if for an ordinary regular expression ρ the star closure $\hat{\boldsymbol{G}}^*(\rho)$ in Eq. 4.23 or Eq. 4.19 does not exist, it means that the production of outputs during at least one sequence in the language of that expression cannot keep up with the consumption of the outputs with period μ i.e. the throughput is too low and therefore the latency does not exist.

Now we provide proof of correctness for latency vector computations for ordinary regular expressions. We first show that using Eqs. 4.17-4.20 we can compute the normalized matrix of any recursive combination of ordinary regular expressions.

Proposition 4.5. *For any ordinary regular expression* ρ*,* $\hat{\boldsymbol{G}}(\rho)$ *computed by Eqs. 4.17-4.20 satisfies* $\hat{\boldsymbol{G}}(\rho) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho)} \hat{\boldsymbol{G}}(\bar{s})$.

Proof. We prove this by structural induction on ρ. The base case $\rho = s$ is trivial. For the induction steps we obtain

$$\begin{aligned}
\hat{\boldsymbol{G}}(\rho_1\rho_2) \quad &= \bigoplus_{\bar{s}_1\bar{s}_2\in\mathscr{L}(\rho_1\rho_2)} \boldsymbol{G}(\bar{s}_1\bar{s}_2) - o(\bar{s}_1\bar{s}_2)\\
&= \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1),\bar{s}_2\in\mathscr{L}(\rho_2)} \boldsymbol{G}(\bar{s}_2)\boldsymbol{G}(\bar{s}_1) - o(\bar{s}_1) - o(\bar{s}_2)\\
&= \bigoplus_{\bar{s}_2\in\mathscr{L}(\rho_2)} \boldsymbol{G}(\bar{s}_2) - o(\bar{s}_2) \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1)} \boldsymbol{G}(\bar{s}_1) - o(\bar{s}_1) = \hat{\boldsymbol{G}}(\rho_2)\hat{\boldsymbol{G}}(\rho_1),\\
\hat{\boldsymbol{G}}(\rho_1 + \rho_2) \quad &= \bigoplus_{\bar{s}\in\mathscr{L}(\rho_1)\cup\mathscr{L}(\rho_2)} \hat{\boldsymbol{G}}(\bar{s}) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho_1)} \hat{\boldsymbol{G}}(\bar{s}) \oplus \bigoplus_{\bar{s}\in\mathscr{L}(\rho_2)} \hat{\boldsymbol{G}}(\bar{s})\\
&= \hat{\boldsymbol{G}}(\rho_1) \oplus \hat{\boldsymbol{G}}(\rho_2),\\
\hat{\boldsymbol{G}}(\rho^n) \quad &= \hat{\boldsymbol{G}}(\rho\rho^{n-1}) = \hat{\boldsymbol{G}}(\rho)\hat{\boldsymbol{G}}(\rho^{n-1}) = \hat{\boldsymbol{G}}(\rho)\hat{\boldsymbol{G}}(\rho)\hat{\boldsymbol{G}}(\rho^{n-2}) = \cdots\\
&= \hat{\boldsymbol{G}}^n(\rho),\\
\hat{\boldsymbol{G}}(\rho^*) \quad &= \bigoplus_{\bar{s}\in\bigcup\limits_{n\geq 0}\mathscr{L}(\rho^n)} \hat{\boldsymbol{G}}(\bar{s}) = \bigoplus_{n\geq 0}\bigoplus_{\bar{s}\in\mathscr{L}(\rho^n)} \hat{\boldsymbol{G}}(\bar{s}) = \mathscr{I} \oplus \hat{\boldsymbol{G}}(\rho) \oplus \hat{\boldsymbol{G}}^2(\rho) \oplus \cdots\\
&= \hat{\boldsymbol{G}}^*(\rho).
\end{aligned}$$

□

Proposition 4.6. *For any ordinary regular expression* ρ*,* $\boldsymbol{\lambda}(\rho)$ *computed by Eqs. 4.21-4.24 satisfies* $\boldsymbol{\lambda}(\rho) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho)} \boldsymbol{\lambda}(\bar{s})$.

Proof. We prove this by structural induction on ρ. The base case $\rho = s$ is trivial. For the induction steps we obtain

$$\begin{aligned}
\boldsymbol{\lambda}(\rho_1\rho_2) \quad &= \bigoplus_{\bar{s}_1\bar{s}_2\in\mathscr{L}(\rho_1\rho_2)} \boldsymbol{\lambda}(\bar{s}_1\bar{s}_2) = \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1),\bar{s}_2\in\mathscr{L}(\rho_2)} \Big[\boldsymbol{\lambda}(\bar{s}_1) \oplus \hat{\boldsymbol{G}}^T(\bar{s}_1)\boldsymbol{\lambda}(\bar{s}_2)\Big]\\
&= \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1)} \boldsymbol{\lambda}(\bar{s}_1) \oplus \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1),\bar{s}_2\in\mathscr{L}(\rho_2)} \Big[\hat{\boldsymbol{G}}^T(\bar{s}_1)\boldsymbol{\lambda}(\bar{s}_2)\Big]\\
&= \boldsymbol{\lambda}(\rho_1) \oplus \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1)} \hat{\boldsymbol{G}}^T(\bar{s}_1) \bigoplus_{\bar{s}_2\in\mathscr{L}(\rho_2)} \boldsymbol{\lambda}(\bar{s}_2) = \boldsymbol{\lambda}(\rho_1) \oplus \hat{\boldsymbol{G}}^T(\rho_1)\boldsymbol{\lambda}(\rho_2),
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\lambda}(\rho_1+\rho_2) \quad &= \bigoplus_{\bar{s}\in\mathscr{L}(\rho_1)\cup\mathscr{L}(\rho_2)} \boldsymbol{\lambda}(\bar{s}) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho_1)} \boldsymbol{\lambda}(\bar{s}) \oplus \bigoplus_{\bar{s}\in\mathscr{L}(\rho_2)} \boldsymbol{\lambda}(\bar{s}) \\
&= \boldsymbol{\lambda}(\rho_1) \oplus \boldsymbol{\lambda}(\rho_2), \\
\boldsymbol{\lambda}(\rho^n) \quad &= \boldsymbol{\lambda}(\rho\rho^{n-1}) = \boldsymbol{\lambda}(\rho) \oplus \hat{\boldsymbol{G}}^T(\rho)\boldsymbol{\lambda}(\rho^{n-1}) \\
&= \boldsymbol{\lambda}(\rho) \oplus \hat{\boldsymbol{G}}^T(\rho)\left(\boldsymbol{\lambda}(\bar{s}) \oplus \hat{\boldsymbol{G}}^T(\rho)\boldsymbol{\lambda}(\rho^{n-2})\right) \\
&= \boldsymbol{\lambda}(\rho) \oplus \hat{\boldsymbol{G}}^T(\rho)\boldsymbol{\lambda}(\rho) \oplus \cdots \oplus (\hat{\boldsymbol{G}}^T(\rho))^{n-1}\boldsymbol{\lambda}(\rho) \\
&= \left(\boldsymbol{\mathscr{I}} \oplus \hat{\boldsymbol{G}}^T(\rho) \oplus \cdots \oplus (\hat{\boldsymbol{G}}^T(\rho))^{n-1}\right)\boldsymbol{\lambda}(\rho) \\
&= \left(\bigoplus_{0\le k<n} (\hat{\boldsymbol{G}}^T(\rho))^k\right)\boldsymbol{\lambda}(\rho), \\
\boldsymbol{\lambda}(\rho^*) \quad &= \bigoplus_{\bar{s}\in\bigcup\limits_{n\ge 0}\mathscr{L}(\rho^n)} \boldsymbol{\lambda}(\bar{s}) = \bigoplus_{n\ge 0}\bigoplus_{0\le k<n} (\hat{\boldsymbol{G}}^T(\rho))^k\boldsymbol{\lambda}(\rho) = (\hat{\boldsymbol{G}}^T(\rho))^*\boldsymbol{\lambda}(\rho) \\
&= (\hat{\boldsymbol{G}}^*(\rho))^T\boldsymbol{\lambda}(\rho).
\end{aligned}$$

□

Finally we provide the latency analysis for an FSM-SADF with a given ω-regular expression σ. We use the same latency vector definition (Eq. 4.15) for regular expressions where $\bar{s}$ denotes in this case, an infinite scenario sequence. We use the fact that the latency vector of an indefinite repetition of an ordinary regular expression i.e. ρ^ω directly follows from Eq. 4.24 when k does not have an upper-bound. This enables us to provide the following computations for the latency of an FSM-SADF.

$$\boldsymbol{\lambda}(\sigma): \quad \boldsymbol{\lambda}(\rho^\omega) = (\hat{\boldsymbol{G}}^T(\rho))^*\boldsymbol{\lambda}(\rho) \tag{4.25}$$

$$\boldsymbol{\lambda}(\rho\sigma) = \boldsymbol{\lambda}(\rho) \oplus \hat{\boldsymbol{G}}^T(\rho)\boldsymbol{\lambda}(\sigma) \tag{4.26}$$

$$\boldsymbol{\lambda}(\sigma_1+\sigma_2) = \boldsymbol{\lambda}(\sigma_1) \oplus \boldsymbol{\lambda}(\sigma_2) \tag{4.27}$$

Proposition 4.7. *For any expression* σ, $\boldsymbol{\lambda}(\sigma)$ *computed by Eqs. 4.25-4.27 satisfies* $\boldsymbol{\lambda}(\sigma) = \bigoplus_{\bar{s}\in\mathscr{L}(\sigma)} \boldsymbol{\lambda}(\bar{s})$.

Proof. We prove this by structural induction on σ. For the base case $\sigma = \rho^\omega$ we obtain

$$\boldsymbol{\lambda}(\rho^\omega) = \bigoplus_{k\ge 0}(\hat{\boldsymbol{G}}^T(\rho))^k\boldsymbol{\lambda}(\rho) = (\hat{\boldsymbol{G}}^T(\rho))^*\boldsymbol{\lambda}(\rho).$$

The proofs of the latency vector computations for the sequential composition $\rho\sigma$ and the choice $\sigma_1 + \sigma_2$ are similar to the proofs of Eq. 4.21 and Eq. 4.22 respectively. $\square$

For the filter example on 9 × 11 frames, we can use the recursive computation to obtain $\boldsymbol{\lambda}^T\left(((ri)^{18}((ri)^2(cm)^7(ro)^2)^9(ro)^{18})^\omega\right) = \left[\begin{array}{ccc} 53 & 51 & 2 \end{array}\right]$, and the latency is 53 when $\boldsymbol{\gamma}_0 = \mathbf{0}$. The latency vector signifies that there are at least 53, 51 and 2 time units time differences between the output(s) that contribute to the worst-case latency (in this case the first output) and the availability time of initial tokens x, y and z respectively. This can be observed also from Figure 3.3.

4.3 Evaluation

We implemented the scalable throughput and latency analysis as two algorithms in the SDF3 tool [65]. We applied our analysis to dataflow models of several realistic applications listed in Table 4.1: H.263 decoder, MP3 decoder, Down-Sampler (DS), Up-Sampler (US), sampler (DS→US), Convolution Filter (CF), sub-sampled convolution filter (DS→CF→US) and Multi-Resolution Convolution Filter (MRCF). The number x following the applications denotes that the input image size is $x \times x$. Parameter L is the length of the representation of the regular expression. For SDF and CSDF graphs, parameter F is the total number of actor firings within one iteration of the application, and for FSM-SADF it is the sum of all actor firings in one iteration of all scenarios.

The SDFG of the H.263 decoder is given in Section 1.2.2. This graph models the behaviour of the decoder on CIF size frames. We adapted the graph to model its behaviour for 16CIF (1408 × 1152) images. We applied our scalable analysis on this application by expressing its behaviour as an SADF sequence of two different scenarios as explained in Section 3.2. The MP3 decoder is modelled as an FSM-SADF [31]. To apply our analysis, we formalized its FSM by a regular expression. The convolution filter is modelled as shown in Figure 3.2.

The up-sampler uses a 3×3 kernel to generate four pixels out of every input pixel. The generated pixels form two pixels in one line and two pixels in the next line. To preserve the line-by-line output of data, the two bottom pixels are first stored as the kernel moves along the first line. After the first line is output, the stored pixels from the second line are output to form another complete line. The structure of operations is as follows.

Table 4.1: The list of applications used for analysis

Nr.	Application	Expression Length	Nr. of Firings
1	CF (8)	37	312
2	CF (32)	45	4344
3	CF (128)	53	66552
4	CF (512)	61	$1.05 \cdot 10^{+6}$
5	CF (2048)	69	$1.67 \cdot 10^{+7}$
6	DS (512)	75	$8.52 \cdot 10^{+5}$
7	DS (1024)	81	$3.40 \cdot 10^{+6}$
8	US (128)	59	$2.13 \cdot 10^{+5}$
9	US (512)	69	$3.40 \cdot 10^{+6}$
10	DS→US (2048)	98	$2.66 \cdot 10^{+7}$
11	DS→CF→US (2048)	113	$3.07 \cdot 10^{+7}$
12	3-level MRCF (2048)	168	$7.12 \cdot 10^{+7}$
13	H.263 (16CIF)	26	$3.04 \cdot 10^{+5}$
14	MP3 decoder	12	3169
15	Example (Figure 2.7)	6	8

1. **frame rush-in phase**: read pixels one by one without computing any output yet for one line of the image, or W pixels;

2. **line rush-in phase**: read one pixel without producing output;

3. **line computation phase**: one pixel is consumed and two pixels are produced (and 2 others are stored) for a whole line minus one pixel i.e. $W-1$ pixels. During this phase $W-1$ pixels are consumed and $2W-2$ pixels are produced;

4. **line rush-out**: two pixels are produced without reading new input;

5. **memory flush**: one line of output is flushed from memory: $2W$ pixels;

6. repeat phases 2–5 for $H-1$ lines;

7. **frame rush-out phase**: produce 2 more lines of output without new input: $4W$ pixels.

We can use the same scenario graphs used in the model of the convolution filter (Figure 3.2) with a suitable expression to express the behaviour of the up-sampler. For example the following expression models the pattern we just explained for the up-sampler.

$$\sigma_{us} = (ri)^W \left((ri)\left((cm)(ro)\right)^{W-1}(ro)^{2W+2}\right)^{H-1}(ro)^{4W}$$

For the computation phase, we used an alternation of *ro* and *cm* to express the behaviour in which one pixel is consumed and two pixels are produced. We also used *ro* for the memory flush and rush-out phases. For the down-sampler we use a 4 × 4 kernel to generate one pixel out of every 4 input pixels. To reduce the number of output pixels by four, the down-sampler produces an output only for every other input line and only for every other input pixel. Again if we use the same scenario graphs we used in the filter example, the pattern in which our down-sampler behaves can be described by the following expression, assuming W and H are even numbers.

$$\sigma_{ds} = (ri)^W \left((ri)^W\left((ri)(cm)\right)^{W/2}\right)^{H/2-1}(ri)^W(ro)^{W/2}$$

The sampler is an up-sampler down-sampler pipeline, where the up-sampler consumes the outputs produced by the down-sampler. In the sub-sampled convolution filter, first the image is down-sampled, then the convolution is applied to the down-sampled image and finally the filtered image is up-sampled. A block diagram of a multi-resolution filter is provided by Keinert et al. [66].

Table 4.2: Throughput and latency analysis results for few realistic applications

Nr.	REM (Exact)				SOTA (Exact)		SOTA (Approx.)	
	Thr.	RT	Lat.	RT	Thr.	RT	Thr.	RT
1	0.2253	< 1	49	< 1	0.2253	< 1	0.2025	$6.00 \cdot 10^{+3}$
2	0.2426	< 1	145	< 1	0.2426	1	0.2229	$8.40 \cdot 10^{+4}$
3	0.2480	< 1	529	< 1	0.2480	7	–	–
4	0.2495	< 1	2065	< 1	0.2495	122	–	–
5	0.2497	< 1	8209	< 1	0.2497	2010	–	–
6	$9.99 \cdot 10^{-2}$	< 1	2063	< 1	$9.99 \cdot 10^{-2}$	93	–	–
7	$9.99 \cdot 10^{-2}$	< 1	4111	< 1	$9.99 \cdot 10^{-2}$	360	–	–
8	0.3073	< 1	271	< 1	0.3073	22	–	–
9	0.3076	< 1	1039	< 1	0.3076	352	–	–
10	0.3076	1	18453	1	0.3076	3202	–	–
11	0.2496	1	38947	1	0.2496	3695	–	–
12	0.2497	1	99225	1	0.2497	8571	–	–
13	$1.17 \cdot 10^{-8}$	< 1	$1.40 \cdot 10^{+4}$	1	$1.17 \cdot 10^{-8}$	21	$4.84 \cdot 10^{-10}$	< 1
14	$1.71 \cdot 10^{-7}$	9	$1.23 \cdot 10^{+7}$	9	$1.71 \cdot 10^{-7}$	14	N/A	N/A
15	0.33	1	8	< 1	0.33	< 1	N/A	N/A

Table 4.2 shows the analysis results for the mentioned applications. In this table the Regular Expression Method (REM) is the method provided in this chapter. It uses the regular expression representation of SADF to recognize repeated sequences and transforms the SADF to an FSM-SADF with a fewer number of FSM states, compared to when the given expression is represented by an FSM. Then it uses the method of Geilen et al. [55] as the final throughput analysis. We use the same SADF representations to compute the latency. SOTA represents the state of the art throughput analysis methods (exact CSDF analysis [29], exact FSM-SADF analysis [55] and approximate CSDF analysis [67]). There is no existing latency analysis. The run-times (RT) reported in this table are obtained on an Ubuntu server with a 3.8Ghz processor and they are measured in milliseconds; the experiments that took more than 10 minutes were terminated, indicated with hyphens for the result and run-time. The REM throughput analysis results in this table are exact, as the FSM representations of the applications in this table do not require acceptance conditions. The results confirm that the REM method is scalable to applications operating on large image frames. Observe that the run-time of the state of the art methods scale linearly in F. This causes scalability problems for, for example, buffer sizing algorithms on complex applications such as multi-resolution filters. The approximate analysis often takes more than the exact analysis to terminate or it is very pessimistic. The analysis run-time for the approximate method is mainly due to transformation to a conservative SDF. According to De Groote et al., the transformation has a quadratic time complexity in the maximum number of CSDF phases, which for our examples is in the order of F [67]. We used the REM method on the MP3 decoder and the example in Figure 2.7 to show the applicability, in general, of the method to SADF models. However, we do not gain much with our throughput analysis compared to the existing methods, because these models do not contain repetitive structures that exhibit multi-scale behaviour.

4.4 Related Work

The prime performance property of interest for dataflow models of computation is throughput. The earliest works on throughput analysis of SDF [20, 46] use a conversion of the graph to its equivalent Homogeneous Synchronous Dataflow (HSDF) graph. HSDF is the single-rate version of SDF. This enables the use of MCM or MCR [68] analysis techniques to obtain the throughput of the graph. This method can also be applied to CSDF by first using a conversion to HSDF [67]. Apart from the fact that the conversion step itself is time consuming, such conversions often result in a very large HSDF (especially for CSDF graphs) which poses scalability issues even for efficient MCM analysis techniques such as Karp's algorithm [69]. Approaches to provide an accurate but more scalable throughput analysis for SDF and CSDF are provided by Ghamarian et al. and Stuijk et al., respectively [49, 29]. In both cases, the analysis can be directly applied to the original graph, which removes the costly conversion step of the earlier works. They both include a simulation phase based on the operational semantics of dataflow. It still simulates all actor firings in each iteration, but stores only one state per iteration. The algorithm builds a global state-space representation of the self-timed execution of the dataflow graph by the simulation. It is known that the self-timed execution of a consistent and strongly-connected (C)SDF consists of a transient phase followed by a periodic phase in which actors fire in a periodic fashion. Throughput is extracted from the periodic phase of the self-timed execution–the transient phase is often very short in practice, but can be arbitrarily large in the worst case.

Theelen et al. provide a throughput analysis for SADF [32]. The analysis is built upon a state-space representation of the graph. The state-space representation captures the behaviour of the graph across sequences of scenarios. Since transitions are at the level of individual firings of actors, the resulting state space becomes very large with larger models and leads to tractability issues. The SADF analysis [55] uses a symbolic simulation method to get the $(\max, +)$ representation of each scenario. This representation abstracts the actor firing dependencies within the scenarios. Then such a representation together with the FSM is used to generate either a state-space of all reachable states (state vectors) or a $(\max, +)$ automaton. In the analysis techniques for FSM-SADF, a (partial) iteration of the graph in a particular scenario is captured in a single transition of the state-space or automaton, instead of the individual actor firings. This leads to improved scalability. The analysis is ultimately mapped on a MCR analysis on a directed graph of the state-space or directly on the automaton. Irrespective of the size of the resulting directed graph or the automaton and the complexity of the final MCR analysis, the process of generating the state space of the time-stamp vectors or the automaton scales linearly in the number of FSM states. We use the same symbolic simulation method to obtain (max,+) representations for scenarios as Geilen et al. [55]. Then we use a transformation on the given SADF to generate a new FSM-SADF with fewer number of FSM states, leading to further scalability improvements for multi-scale dataflow models compared to the analyses of Geilen et al.

Besides exact throughput analysis, it is possible to approximate throughput. An approximate conversion method [67] can be used to generate pessimistic and optimistic HSDF abstractions of an SDF, but with the same size as the original graph, the same number of actors and edges. Throughput analysis over the approximate HSDF is done for the sake of shorter analysis run-time. We compare our technique to the conservative approximate method of De Groote et al. [67] in Section 4.3. Our technique turns out to be better scalable, despite being exact.

A second performance property of interest is latency. Geilen et al. provide a definition for latency which we also use as a basis in this thesis (where it is in fact generalized) [55]. They sketch a possible way to compute the latency for FSM-SADF without providing an algorithm. The proposed approach requires the generation of the state-space which scales linearly in the number of FSM states. Moreira et al. [70] use the same latency notion as Geilen et al. [55]. They provide an algorithm to compute the latency. Their approach promotes the use of static periodic schedules as a conservative approximation of self-timed schedules. Therefore, the analysis gives an upper bound on the latency. Moreover, it can only be applied to SDF, CSDF, i.e., the deterministic cases of SADF where the language has only one scenario sequence. We provide an algorithm for latency analysis of an arbitrary SADF model. We show that when we represent the set of possible scenario sequences of an SADF with a regular expression, the latency computation time using our algorithm scales linearly in the length of the regular expression.

4.5 Conclusion

We provided a scalable throughput and latency analysis for multi-scale applications that are modelled by scenario-aware dataflow graphs. We showed that such models often have a cyclic behaviour with a large number of actor firings in the cycle. We overcame the scalability issue of existing exact analysis techniques by exploiting the repetitive structures within the large cycle. Our analysis scales logarithmically in the number of repetitions for such repetitive structures whereas the state of the art analysis scales at least linearly. We implemented our analysis and applied it to several realistic applications. The results show that our analysis provides accurate analysis in a shorter time compared to the existing dataflow analysis methods.

Chapter 5

Compositional Modelling

In Chapter 3 we used SADF models to represent real-time streaming applications. In this chapter, we propose a method to generate SADF models of cyclo-static applications, compositionally, from the SADF models of their modules. This chapter is based on publication [41].

5.1 Introduction

Modular design is a design method that regards a system as several smaller, independently created modules that are interacting with each other through common interfaces. Modular design facilitates the creation of systems by composing pre-defined, reusable modules that can be designed separately. For instance, video encoders/decoders are typically constructed from a number of functional modules such as quantizers, transformers, sub-samplers, multiplexers, etc. A complex real-time streaming application with a predictable performance can be created with a *modular model*, constructed by composing the models of its modules. A composed model has the same type of interface as the modules have. This allows the composite models to be composed in an iterative manner to generate a hierarchical model of a complex system. A concise representation and scalable analysis methods for the model are very important, because the models can grow quickly.

In Chapter 3, we provided an approach to obtain SADF models of streaming applications with long periodic patterns. Composing models with periodic patterns results in a model with a periodic pattern with a common hyperperiod. For instance, consider the composition of two convolution filters as

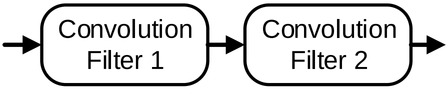


Figure 5.1: Two producer-consumer composed convolution filters.

shown in Figure 5.1. In this figure, Filter 2 consumes the data produced by Filter 1. Such a composition is called the *Producer-Consumer (PC)* composition. Recall that the behaviour of a convolution filter is described by a pattern of multiple phases (see Section 3.1). The composite behaviour composes the different phases of both filters according to the data dependencies between the two filters. This leads to the creation of *composite phases.* A composite phase describes a behaviour in which the dependency relations of two phases, one from each module, are combined.

In Figure 5.1, in the first composite phase, when Filter 1 is in the frame rush-in phase, Filter 2 is not active, since it does not receive any inputs. After consuming two lines of input pixels, Filter 1 goes to the line rush-in phase, while Filter 2 is still inactive due to the same reason. This constitutes the second composite phase. In the third composite phase, while Filter 1 is producing (intermediate) outputs in the computation phase, Filter 2 goes to the frame rush-in phase, consuming the outputs produced by Filter 1. In the next phase, Filter 1 goes to the line rush-out phase while Filter 2 is still in the frame rush-in phase, and so on. This pattern repeats after Filter 2 completes its frame rush-out phase. The pattern of the composite phases is called the *hyper-period pattern.*

The hyper-period patterns of complex applications are often very long and computing them is not trivial. For instance, consider the block diagram of a 3-level multi-resolution filter shown in Figure 5.2 [66]. Input images are fed to the highest level. At each level, the image is decomposed into a difference image and a down-sampled image. The difference image is fed to a filter module and the down-sampled image is used as the input image of the lower level. After filtering, the filtered images from all levels are added up to reconstruct the output image. This application has complicated data dependencies between different modules due to its non-trivial topology. This leads to the creation of many composite phases. Moreover, the filter, up-sampler and down-sampler modules have different periodic patterns. For instance, the down sampler pattern involves skipping the production of outputs for every other input line while the up sampler pattern shows the production of two lines per every input line. Composing modules with different periodic patterns results in a

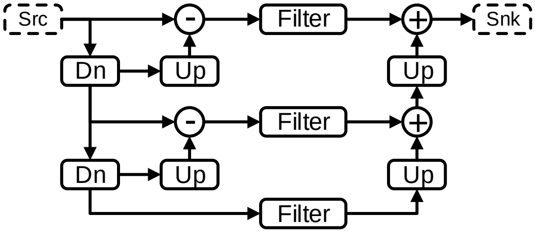


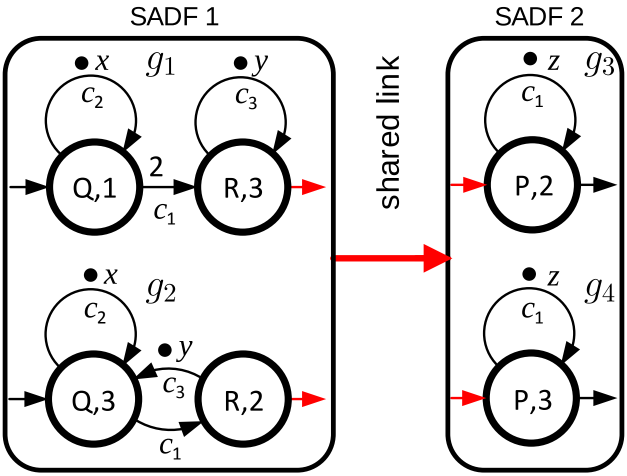
Figure 5.2: A multi-resolution filter [66].

long hyper-period pattern. Hence, the manual composition is tedious and error-prone.

In this chapter, we propose an efficient algorithmic method to compose SADF models of applications with periodic behaviours by generating the hyper-period patterns automatically in a concise regular expression representation. Our approach can be iteratively used to generate SADF models of applications that are composed of more than two modules. This approach involves two steps: computing the hyper-period of the composition and the timing behaviour of its composite phases. The timing behaviour of the composite phases is computed from the $(\max, +)$ characterizations of the composing modules, straightforwardly. We provide an efficient algorithm for computing the hyper-period pattern. The algorithm supports composition with producer-consumer and feedback synchronization on data dependencies between scenarios of different modules. The algorithm exploits the scenario repetitions in the regular expressions to identify repetitive patterns in the composition. This technique is used to efficiently and directly generate a concise regular expression for the hyper-period pattern. We show that our approach is fast enough to automatically generate concise models of several complex realistic image processing applications such as the multi-resolution filter.

5.2 Producer-Consumer Composition

This section defines the producer-consumer composition of two SADF models and provides an efficient algorithm to generate the composition. Figure 5.3 shows an example of this composition. The producer (on the left) and consumer (on the right) SADF models communicate through a *shared link*. That is, all scenarios in the producer SADF produce output tokens on the shared link and all scenarios in the consumer SADF consume input tokens from this



(a) Composed SADF models.

Scenario	a	b	c	d
Graph	g_1	g_2	g_3	g_4
Reward	2 (firings of R)	1 (firings of R)	1 (firings of P)	1 (firings of P)
Reps	$[\mathsf{Q}, \mathsf{R}] \mapsto [1, 2]$	$[\mathsf{Q}, \mathsf{R}] \mapsto [1, 1]$	$[\mathsf{P}] \mapsto [1]$	$[\mathsf{P}] \mapsto [1]$
ITs	$c_2(x), c_3(y)$	$c_2(x), c_3(y)$	$c_1(z)$	$c_1(z)$
FTs	$c_2(x), c_3(y)$	$c_2(x), c_3(y)$	$c_1(z)$	$c_1(z)$

(b) Scenario specifications.

SADF	SADF 1	SADF 2
Scenarios	$\{a, b\}$	$\{c, d\}$
Regular Expression	$(ab^{10})^{\omega}$	$(cd)^{\omega}$

(c) SADF specifications.

Figure 5.3: An example of a producer-consumer composition.

link. Observe that the scenario graphs are open (see Section 2.10). To compose two SADF models we need to compose periodic sequences of the producer and consumer. The composition is a periodic sequence of *composite scenarios.* A composite scenario combines a finite sequence of scenarios from the producer with a finite sequence from the consumer such that its execution does not leave any tokens on the shared link. In other words, a composite scenario can be created from a non-empty pair of finite scenario sequences from the producer and consumer, such that the total number of outputs produced by the producer sequence equals the total number of tokens consumed by the consumer sequence. For instance, scenario a from SADF 1 can be composed with sequence cd from SADF 2, since a produces two outputs and c and d consume one input each.

A pair $(\bar{s},\bar{t})$ of sequences is called a *composable pair* if the number of tokens produced by $\bar{s}$ is the same as the number of tokens consumed by $\bar{t}$. A composable pair can form a composite scenario. We denote the composite scenario by $\bar{s}{\rightarrow}\bar{t}$ and derive its $(\max,+)$ characterization. To do this, we first need to introduce the $(\max,+)$ characterizations of finite sequences of open scenarios, $\bar{s}$ and $\bar{t}$. Recall that in Section 3.1, we obtained a $(\max,+)$ characterization for a finite sequence of non-open scenarios, by repeatedly substituting the initial state vectors with final state vectors in the state equations of the scenarios. For a finite sequence of open scenarios, a $(\max,+)$ characterization can be computed in a similar way, with an additional step of augmenting input and output matrices of all scenarios in the sequence.

Consider the sequence ab in SADF 1 shown in Figure 5.3 as an example. As explained in Section 2.10, we can obtain the following equations for scenarios a and b.

$$\begin{aligned} \boldsymbol{\gamma}_a' &= \boldsymbol{G}_a\boldsymbol{\gamma}_a \oplus \boldsymbol{K}_a\boldsymbol{u}_a & \qquad \boldsymbol{\gamma}_b' &= \boldsymbol{G}_b\boldsymbol{\gamma}_b \oplus \boldsymbol{K}_b\boldsymbol{u}_b \\ \boldsymbol{y}_a &= \boldsymbol{H}_a\boldsymbol{\gamma}_a \oplus \boldsymbol{L}_a\boldsymbol{u}_a & \qquad \boldsymbol{y}_b &= \boldsymbol{H}_b\boldsymbol{\gamma}_b \oplus \boldsymbol{L}_b\boldsymbol{u}_b \end{aligned}$$

To compute the $(\max,+)$ characterization of the sequence ab, we need to substitute $\boldsymbol{\gamma}_b$ in the equations of the right side with $\boldsymbol{\gamma}_a'$ as follows.

$$\begin{aligned} \boldsymbol{\gamma}_b' &= \boldsymbol{G}_b\boldsymbol{\gamma}_b \oplus \boldsymbol{K}_b\boldsymbol{u}_b = \boldsymbol{G}_b\left(\boldsymbol{G}_a\boldsymbol{\gamma}_a \oplus \boldsymbol{K}_a\boldsymbol{u}_a\right) \oplus \boldsymbol{K}_b\boldsymbol{u}_b \\ \boldsymbol{y}_b &= \boldsymbol{H}_b\boldsymbol{\gamma}_b \oplus \boldsymbol{L}_b\boldsymbol{u}_b = \boldsymbol{H}_b\left(\boldsymbol{G}_a\boldsymbol{\gamma}_a \oplus \boldsymbol{K}_a\boldsymbol{u}_a\right) \oplus \boldsymbol{L}_b\boldsymbol{u}_b \end{aligned}$$

By distributing the multiplications over additions in the above equations and augmenting the input and output vectors, we can derive the $(\max,+)$ characterization of the sequence ab in the form of Eq. 2.8 as follows.

$$\begin{bmatrix} \gamma'_b \\ \boldsymbol{y}_a \\ \boldsymbol{y}_b \end{bmatrix} = \left[\begin{array}{c|cc} \boldsymbol{G}_b\boldsymbol{G}_a & \boldsymbol{G}_b\boldsymbol{K}_a & \boldsymbol{K}_b \\ \hline \boldsymbol{H}_a & \boldsymbol{L}_a & -\infty \\ \boldsymbol{H}_b\boldsymbol{G}_a & \boldsymbol{H}_b\boldsymbol{K}_a & \boldsymbol{L}_b \end{array}\right] \begin{bmatrix} \gamma_a \\ \boldsymbol{u}_a \\ \boldsymbol{u}_b \end{bmatrix}$$

In the rest of the thesis, $\boldsymbol{G}_{\bar{s}}, \boldsymbol{K}_{\bar{s}}, \boldsymbol{H}_{\bar{s}}$ and $\boldsymbol{L}_{\bar{s}}$ denote the $(\max, +)$ matrices of a finite scenario sequence $\bar{s}$.

Now we can compute the $(\max, +)$ characterization of a composite scenario $\bar{s}{\to}\bar{t}$. When $\bar{s}$ and $\bar{t}$ are composed, the time-stamps of the output tokens produced by $\bar{s}$ are assigned to input tokens consumed by $\bar{t}$ in the order determined by the sequences. For instance in the composite scenario $a{\to}cd$, the time-stamp of the first output of a, i.e., o_1 (note that it is logically the first, not necessarily temporally first) is assigned to the input of c and the second output, o_2 is assigned to the input of d. Since all inputs consumed by $\bar{t}$ are fed by outputs produced by $\bar{s}$, the inputs of the composite scenario are only the inputs of $\bar{s}$ and its outputs are only the outputs of $\bar{t}$, i.e., $\boldsymbol{u}_{\bar{s}\to\bar{t}} = \boldsymbol{u}_{\bar{s}}$ and $\boldsymbol{y}_{\bar{s}\to\bar{t}} = \boldsymbol{y}_{\bar{t}}$. The $(\max, +)$ characterization of a composite scenario $\bar{s}{\to}\bar{t}$ can be obtained by substituting the input vector $\boldsymbol{u}_{\bar{t}}$ with output vector $\boldsymbol{y}_{\bar{s}}$ in the $(\max, +)$ characterization of $\bar{t}$. Doing so and considering $\gamma_{\bar{s}\to\bar{t}} = [\ \gamma_{\bar{s}} \quad \gamma_{\bar{t}}\ ]^T$ and $\gamma'_{\bar{s}\to\bar{t}} = [\ \gamma'_{\bar{s}} \quad \gamma'_{\bar{t}}\ ]^T$ we can obtain the following $(\max, +)$ matrices for $\bar{s}{\to}\bar{t}$, straightforwardly.

$$\boldsymbol{G}_{\bar{s}\to\bar{t}} = \begin{bmatrix} \boldsymbol{G}_{\bar{s}} & -\infty \\ \boldsymbol{K}_{\bar{t}}\boldsymbol{H}_{\bar{s}} & \boldsymbol{G}_{\bar{t}} \end{bmatrix} \qquad \boldsymbol{K}_{\bar{s}\to\bar{t}} = \begin{bmatrix} \boldsymbol{K}_{\bar{s}} \\ \boldsymbol{K}_{\bar{t}}\boldsymbol{L}_{\bar{s}} \end{bmatrix}$$

$$\boldsymbol{H}_{\bar{s}\to\bar{t}} = \begin{bmatrix} \boldsymbol{L}_{\bar{t}}\boldsymbol{H}_{\bar{s}} & \boldsymbol{H}_{\bar{t}} \end{bmatrix} \qquad \boldsymbol{L}_{\bar{s}\to\bar{t}} = \boldsymbol{L}_{\bar{t}}\boldsymbol{L}_{\bar{s}}$$

The *hyper-period* of periodic producer and consumer sequences is a sequence of composite scenarios that is composed of a whole number of periods of the producer and consumer sequences. The number of producer and consumer periods within the hyper-period depends on the total number of tokens produced by the producer and consumed by the consumer sequences in their periods. For instance, one period of $(ab^{10})^\omega$ produces 12 tokens and, 6 periods of $(cd)^\omega$ consume 12 tokens (2 tokens per period); therefore ab^{10} and $(cd)^6$ compose the hyper-period of the example in Figure 5.3. Let $(\bar{s})^\omega$ and $(\bar{t})^\omega$ denote a periodic producer and consumer, respectively. Assume an execution of $\bar{s}$ produces $o(\bar{s})$ output tokens, and an execution of $\bar{t}$ consumes $i(\bar{t})$ input tokens. The producer and consumer sequences, $(\bar{s})^u$ and $(\bar{t})^v$, compose the hyper-period, where u and v are computed using the least common multiple operation as follows.

$$u = \frac{\mathrm{lcm}(o(\bar{s}), i(\bar{t}))}{o(\bar{s})}, v = \frac{\mathrm{lcm}(o(\bar{s}), i(\bar{t}))}{i(\bar{t})} \tag{5.1}$$

Trivially, the hyper-period can be a single composite scenario that is composed of the producer and consumer sequences in the hyper-period. For instance, we can create the composite scenario $(ab^{10} \rightarrow (cd)^6)$. However, such a composition hides the periodic patterns of the modules inside possibly a gigantic scenario, which will have an adverse effect on the scalability of analysis. Moreover, such a gigantic composite scenario may have a very large $(\max, +)$ characterization in terms of the sizes of matrices. For instance, if all phases of the filters in Figure 5.1 are composed into one scenario, the composite scenario will have millions of inputs and outputs for high resolution frames, which requires matrices $\boldsymbol{K}$, $\boldsymbol{H}$ and $\boldsymbol{L}$ to have millions of rows and/or columns.

To preserve the periodic patterns of the producer and consumer in the hyper-period, we construct the hyper-period from composable pairs of sequences that are *minimal.* A composable pair of sequences is called minimal if no pair of prefixes from those sequences can make a composite scenario. For instance (ab, cdc) is not a minimal composite pair, because the prefixes a (from ab) and cd (from cdc) can form a composite scenario. A minimal pair of composable sequences makes a *minimal composite scenario.*

In a naive way, such a hyper-period can be obtained by successively taking minimal pairs of composable sequences from the producer and consumer sequences and composing them into a sequence of composite scenarios. For instance, let's consider the composition of sequences ab^{10} and $(cd)^6$. We start with composing a and cd and, create the first minimal composite scenario $a \rightarrow cd$. Now b^{10} is left from ab^{10} and $(cd)^5$ is left from $(cd)^6$. We proceed with composing b and c then, b and d and create minimal composite scenarios $b \rightarrow c$ and $b \rightarrow d$, which leaves us with b^8 and $(cd)^4$. By repeating b and c and, b and d composition four times more, we have completed the composition. This method produces a flat representation of the hyper-period, i.e. $(a \rightarrow cd)(b \rightarrow c)(b \rightarrow d)(b \rightarrow c)(b \rightarrow d) \cdots (b \rightarrow c)(b \rightarrow d)$. However, a compact representation $(a \rightarrow cd)((b \rightarrow c)(b \rightarrow d))^5$ is desired as the outcome.

The run-time of this naive approach scales linearly in the length of the hyper-period. For instance in SADF 1, if b repeats 10000 times instead of 10 times, the naive approach takes approximately 1000 fold more time to terminate, since b and c and, b and d composition will repeat for 5000 times in the hyper-period. Moreover, since it produces a flat representation of the hyper-period, it requires an additional step to obtain a concise representation from the flat representation. This additional step can be performed using the method introduced by Nakamura et al. [71]. This method has a time complexity of $\mathcal{O}(n^2 \log n)$, where n is the length of the flat representation. Therefore, it dominates the complexity of the naive approach. We improve the scalability of the naive method by detecting repetitive patterns during the composition and directly generating a repetition representation of the composite sequence.

A repetitive pattern within the hyper-period occurs when repetitive subsequences from both producer and consumer sequences are composed. The repetitive pattern repeats until there are no repetitions left from at least one of the subsequences. Recognizing repetitive patterns during the composition can be done by tracking the remaining repetitions left from all repetitive subsequences of the producer and consumer sequences. To this end, we define a notion of *state* that carries the necessary information about the remaining repetitions from their subsequences. We use a representation for the states that is similar to the representation of the sequences, except that for the states, we use the notation n/m, where n is the initial number of repetitions and m is the number of remaining repetitions.

We illustrate the whole procedure by considering the composition of two convolution filters (Figure 5.1), as their scenario sequences contain many repetitions. In Section 3.1, we described the behaviour of the convolution filters by the repetitive sequence $((ri)^{2W}((ri)^2(cm)^{W-2}(ro)^2)^{H-2}(ro)^{2W})^{\omega}$. However, this sequence is rather complicated for the purpose of illustrating the composition. Instead, we define new scenarios, which are in the coarser granularity of lines, rather than individual pixels. Let the *preparation* scenario, p, correspond to the consumption of one line of input pixels in the frame rush-in phase, let the *main* scenario, m, correspond to the computation of a line of output pixels between the frame rush-in and frame rush-out phases, and let finally the *termination* scenario, t, correspond to the production of one line of output pixels in the frame rush-out phase. Using the new scenarios, we describe the convolution filter pattern by the regular expression $(p^2m^{H-2}t^2)^{\omega}$.

Now we proceed with the PC composition of two convolution filters considering input frames of size 960×960 pixels. Since all output pixels produced in one period of the producer filter are consumed in one period of the consumer filter, the hyper-period contains one complete period from each filter. Therefore, the consumer and producer sequences in the hyper-period are initially both at state $(p^{2/0}m^{958/0}t^{2/0})^{1/1}$. This means that the whole producer/consumer filter sequence is yet to be composed, i.e., no scenarios from the filter sequence have been composed yet.

When a composite scenario sequence is created, the producer and/or the consumer states transition. Again consider the initial state $(p^{2/2}m^{958/958}t^{2/2})^{1/0}$ for both filters. When sequence $(p \rightarrow \epsilon)^2$ is composed (this sequence corresponds to the first composite phase), two p scenarios are taken from the producer sequence and the producer state transitions to $(p^{2/0}m^{958/958}t^{2/2})^{1/0}$. The consumer state does not change, because the composed sequence contains only empty scenarios from the consumer.

From the state transitions, we can identify repetitive patterns in the composite sequence. For instance, when the first $m \rightarrow p$ scenario is created, the producer state transitions from $(p^{2/0}m^{958/958}t^{2/2})^{1/0}$ to $(p^{2/0}m^{958/957}t^{2/2})^{1/0}$ and the consumer state transitions from $(p^{2/2}m^{958/958}t^{2/2})^{1/0}$ to $(p^{2/1}m^{958/958}t^{2/2})^{1/0}$. Comparing the states before and after this composition shows that, for the producer, the remaining repetitions of scenario m has been changed from 958 to 957 and, for the consumer, the remaining repetitions of scenario p has been changed from 2 to 1. Since for both producer and consumer there are differences in the remaining repetitions of exactly one exponent, that means repetitive subsequences have been composed and therefore, a repetitive pattern is detected. In this case the composite scenario $m \rightarrow p$ is repeated two times, since there are only two repetitions of p in the consumer sequence. Note that if we create the composite scenario $p^2m \rightarrow p$, the producer state transitions from $(p^{2/2}m^{958/958}t^{2/2})^{1/0}$ to $(p^{2/0}m^{958/957}t^{2/2})^{1/0}$. No repetitions of this composite scenario can be found in the hyper-period as there are differences in the remaining repetitions of two exponents.

In general, if after creating a composite sequence, the remaining repetitions of at most one exponent in the producer state changes from i to i' and the remaining repetitions of at most one exponent in the consumer sequence changes from j to j', a repetition of that composite sequence in the hyper-period is found and the number of repetitions is computed as follows.

$$r = \min\left\{ \left\lfloor \frac{i}{i-i'} \right\rfloor, \left\lfloor \frac{j}{j-j'} \right\rfloor \right\} \tag{5.2}$$

If the state of the producer/consumer transitions to itself (this happens when an empty scenario is composed into a composite scenario), the corresponding term is removed from Eq. 5.2, as the denominator would be zero.

After creating the repetitive pattern in the hyper-period the remaining repetitions of the producer and consumer subsequences are updated as follows.

$$i'' = i - r(i-i') \ , \ j'' = j - r(j-j') \tag{5.3}$$

We provide an efficient algorithm for computing the hyper-period by detecting repetitive patterns via state comparison. Algorithm 2 sketches our hyper-period computation method. The algorithm accepts the repetition representation of the producer and consumer scenario sequences, $\bar{s}$ and $\bar{t}$ and,

ALGORITHM 2: Hyper-period Computation

```
Input: The pair of sequences $(\bar{s},\bar{t})$
Output: The hyper-period sequence $\bar{r}$
$\bar{l} = \bar{r} = \epsilon$;
$(\tilde{s}, \tilde{t})$ = computeSequencesInHyperPeriod$(\bar{s}, \bar{t})$;
$(S_p, S_c)$ = initialState$(\tilde{s}, \tilde{t})$;
repeat
    append $(S_p, S_c)$ to $\bar{l}$;
    $\sigma$ = createMinimalCompositeScenario$(S_p, S_c)$;
    append $\sigma$ to $\bar{r}$;
    $(S'_p, S'_c)$ = newState$((S_p, S_c), \sigma)$;
    repeat
        $\bar{\sigma} = \epsilon$; $k = 0$;
        repeat
            $(T_p, T_c) = l_k$;
            if $S'_p \preceq T_p$ and $S'_c \preceq T_c$ then
                $\bar{\sigma}$ = getSequence$(\bar{r}, (T_p, T_c), (S'_p, S'_c))$;
                $(i', i)$ = getRemainingReps$(S'_p, T_p)$;
                $(j', j)$ = getRemainingReps$(S'_c, T_c)$;
                break;
            end
            $k = k + 1$;
        until $k \neq$ length$(\bar{l})$;
        if $\bar{\sigma} \neq \epsilon$ then
            $r = \min\{\lfloor \frac{i}{i-i'} \rfloor, \lfloor \frac{j}{j-j'} \rfloor\}$;
            replace $\bar{\sigma}$ with $\bar{\sigma}^r$ in $\bar{r}$;
            $i'' = i - r(i - i')$; $j'' = j - r(j - j')$;
            updateState$(S'_p, i'')$; updateState$(S'_c, j'')$;
        end
        $(S_p, S_c) = (S'_p, S'_c)$;
    until $S_p = S_c = \epsilon$ or $\bar{\sigma} = \epsilon$;
until $S_p = S_c = \epsilon$;
```

Table 5.1: Data of Algorithm 2 running on the example

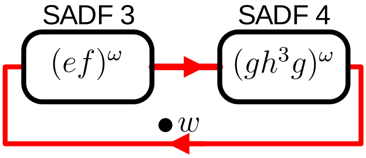
$\bar{l}$	S_p	S_c	$\bar{r}$
l_0	$(a^{1/1}b^{10/10})^{1/0}$	$(c^{1/0}d^{1/0})^{6/6}$	ϵ
l_1	$(a^{1/0}b^{10/10})^{1/0}$	$(c^{1/0}d^{1/0})^{6/5}$	$a{\to}cd$
l_2	$(a^{1/0}b^{10/9})^{1/0}$	$(c^{1/0}d^{1/1})^{6/5}$	$(a{\to}cd)(b{\to}c)$
l_3	$(a^{1/0}b^{10/8})^{1/0}$	$(c^{1/0}d^{1/0})^{6/4}$	$(a{\to}cd)(b{\to}c)(b{\to}d)$
l_4	$(a^{1/1}b^{10/0})^{1/0}$	$(c^{1/0}d^{1/0})^{6/0}$	$(a{\to}cd)((b{\to}c)(b{\to}d))^5$

generates the composite sequence $\bar{r}$. The algorithm stores a sequence $\bar{l}$ that contains pairs of producer and consumer states. Initially $\bar{l}$ and $\bar{r}$ are empty sequences. The algorithm first computes the producer and consumer sequences in the hyper-period, using Eq. 5.1 (Line 2). Then, it obtains the pair of initial producer and consumer states and stores them in (S_p, S_c) (Line 3). Lines 5-27 are repeated until the sequences in the hyper-period are completely composed. Line 5 appends the latest pair of producer and consumer states to $\bar{l}$. A new minimal composite scenario is created and appended to the composite sequence in Lines 6 and 7. After the composite scenario is created, the new pair of states is stored in (S'_p, S'_c) (Line 8). Then, it is compared with the pairs in $\bar{l}$, starting from the first pair, i.e., the oldest pair (Lines 12-19). An element of $\bar{l}$ is stored in (T_p, T_c). If T_p compares to S'_p and T_c compares to S'_c (we denote the comparison with operator $\preceq$ in the algorithm), then the sequence $\bar{\sigma}$ of minimal composite scenarios that are appended to $\bar{r}$ while transitioning from (T_p, T_c) to (S'_p, S'_c) is identified as a repetitive sequence. The remaining repetitions from the repetitive producer subsequence at states S_p and S'_p are stored in i and i', respectively. Similarly, j and j' respectively store these values for the consumer at states S_c and S'_c. If a repetitive pattern is detected, i.e., $\bar{\sigma}$ is non-empty, the repetition count is computed using Eq. 5.2, $\bar{\sigma}^r$ is created and replaced with $\bar{\sigma}$ in $\bar{r}$ and finally, the remaining repetitions i'' and j'' are computed using Eq. 5.3 and used to update S'_p and S'_c (Lines 22-25). The algorithm goes to Line 10 to check for another repetition after the states are updated. Otherwise the algorithm starts the next iteration (i.e., it goes to Line 5) with the latest pair of states stored in (S_p, S_c) (Line 27).

To illustrate, consider the example in Figure 5.3. The sequences ab^{10} and $(cd)^6$ are obtained as the producer and consumer sequences in the hyper-period. The initial pair of producer and consumer states is computed and appended to $\bar{l}$ (l_0 in Table 5.1). Then the composite scenario $a{\to}cd$ is created and appended to $\bar{r}$. Now the latest pair of states, $((a^{1/0}b^{10/10})^{1/0}, (c^{1/0}d^{1/0})^{6/5})$, is compared with the first entry of $\bar{l}$ i.e. l_0. The comparison reveals a repetitive pattern, since in both producer and consumer, the remaining repetitions of only one exponent has been changed (exponent of a from $i = 1$ to $i' = 0$ and the outer exponent of consumer from $j = 6$ to $j' = 5$). Therefore, scenario $a \rightarrow cd$ repeats with the repetition count of one (using Eq. 5.2). Then the states are updated to $((a^{1/0}b^{10/10})^{1/0}, (c^{1/0}d^{1/0})^{6/5})$ using Eq. 5.3. The second iteration starts with appending the latest pair of states to $\bar{l}$ (i.e. entry l_1). Then the minimal composite scenario $b \rightarrow c$ is created. Comparing the pair of states after the creation of this scenario with l_0 does not show a repetitive pattern. However, comparing with l_1 shows a repetitive pattern on $b \rightarrow c$ with repetition of one. The flow of the third iteration is similar to the second iteration where $b \rightarrow d$ is created. In the fourth iteration, the comparison between the latest pair of states i.e. $l_3 = ((a^{1/0}b^{10/8})^{1/0}, (c^{1/0}d^{1/0})^{6/4})$ and the pair l_1 shows a repetitive pattern ($i = 10$, $i' = 8$, $j = 5$, $j' = 4$). Eq. 5.2 calculates the repetition of 5 for the sequence that is appended to $\bar{r}$ between the compared pairs of states, i.e. the sequence $(b \rightarrow c)(b \rightarrow d)$. This iteration is the last iteration, since both states become ϵ after they are updated using Eq 5.3. Therefore, the computed hyper-period is $(a \rightarrow cd)((b \rightarrow c)(b \rightarrow d))^5$.

To explain the complexity of Algorithm 2, we assume the producer and consumer sequences are given by syntax trees and the hyper-period sequence is outputted as a syntax tree. The leaves of the syntax trees are labelled by scenarios and the inner nodes are either a binary operator labelled $\cdot$ (sequential composition), or they are a unary operator and labelled by ω or a natural number n (for the repetition). To compute the input sequences in the hyper-period (Line 2) we need to compute the number of tokens produced and consumed by the producer and consumer in their periods, respectively. This can be computed by a reversed traversal of the input trees and it has the complexity $\mathcal{O}(n_{\bar{s}} + n_{\bar{t}})$ where $n_{\bar{s}}$ and $n_{\bar{t}}$ denote the sizes (the number of nodes) of the input syntax trees. In the iterative part of the algorithm (Lines 5-27), by creating a new minimal composite scenario, a new leaf on the output tree is created. Also a new pair of states is added to list l which is looked up in every iteration. This means the algorithm scales quadratically in the number of leaves of the output tree. The number of leaves in the output tree is at least the maximum of the number of leaves in the input sequences. In the worst-case, when the input sequences do not match, in the sense that their composition does not produce any repetitive patterns (i.e. when the output turns out to be a flat sequence), the number of leaves in the output tree may grow as large as the length of the flattened hyper-period sequence.

It is worth mentioning that in the case of multiple consumers and producers, Algorithm 2 can be repetitively used to generate the composition. More-



(a) Composed SADF models.

Scenario	e	f	g	h
Nr. of Inputs	1	1	0	1
Nr. of Outputs	2	1	1	0

(b) Scenario specifications.

Figure 5.4: An example of a feedback composition.

over, the algorithm can handle the cases with more that one shared channel between the producer and consumer. In this case, the minimal composite scenarios are created such that their execution does not leave tokens on any of the shared channels.

5.3 Feedback Composition

This section generalizes the results of the previous section to the case where there are cyclic dependencies between the modules, as opposed to the PC composition. Figure 5.4a shows an example of such a composition. Observe that the cyclic dependencies are established by two shared links in opposite directions between the two modules and the shared link at the bottom contains an initial token w. We refer to such a composition as a *feedback composition.* SADF 3 has scenario sequence $(ef)^\omega$, where e and f are distinct scenarios. In SADF 4, g and h each denote a scenario and $(gh^3g)^\omega$ represents the sequence of scenarios. To compute the composite scenario sequence, we need to know the number of inputs and outputs consumed and produced by all scenarios. We do not provide the complete specifications of the models of the modules, as we want to discuss only the computation of the hyper-period in the feedback composition. The table of Figure 5.4b specifies the number of inputs and outputs for every scenario.

In the feedback composition, the composite scenarios are defined such that their execution does not change the number of tokens on the shared links. For example in Figure 5.4, scenario e and sequence gh^2 can create a composite scenario. We denote this scenario by $e{\leftrightarrow}fg^2$. The composite scenario $e{\leftrightarrow}fg^2$ can execute both scenario sequences in the composition and therefore, it is *deadlock free*. First, scenario e consumes token w, executes, and produces two tokens on the link from SADF 3 to 2. At the same time, execution of g produces a token on the link from SADF 4 to 1. Then, two executions of h consume the tokens produced by scenario e. After these executions, the shared links contain the same numbers of tokens as they did before the execution of the composite scenario, which complies with the definition of composite scenarios. Observe that this definition is consistent with the definition provided for the PC composition. In the PC composition, the composite scenarios do not leave tokens on the shared links because they are initially empty. Moreover, the PC composition cannot create deadlocks as it does not introduce any cyclic dependencies.

The $(\max, +)$ characterization of composite scenarios in the feedback composition can be computed using the method provided by Skelin et al. [57]. In this method, first the $(\max, +)$ characterizations of the modules are generated by performing the traditional symbolic simulation [31] on the models of the modules. Then, these representations are used in symbolic simulation of the composite graph to generate the $(\max, +)$ characterization of the composite scenario. The deadlock freedom of the composite scenario is automatically checked during the simulation. That is, if there exists an actor firing schedule for all firings in the composite scenario, the simulation terminates and the composition is deadlock free.

Similar to the PC composition, the hyper-period combines the whole numbers of periods from the two modules. Here, the hyper-period is defined only if the cyclic dependencies are correctly modelled, whereas the PC composition is guaranteed to have a hyper-period. For the feedback composition to have a hyper-period, the module periods must be consistent. That is, positive numbers p and q must exists such that the outputs produced by p periods of module 1 are consumed by q periods of module 2 and the outputs produced by q periods of module 2 are consumed by p periods of module 1. For instance, a period of SADF 3 consumes $1 + 1 = 2$ tokens and produces $2 + 1 = 3$ tokens. Similarly, one period of SADF 4 consumes $0 + 3 \times 1 + 0 = 3$ tokens and produces $1 + 3 \times 0 + 1 = 2$ tokens. Therefore the periods are consistent and have a hyper-period that contains one period from each module. As shown in the example above, the consistency of a feedback composition can be simply checked by solving a balance equation for every shared link, similar to the consistency check for SDF graphs.

Given a consistent pair of scenario sequences, Algorithm 2 can be used to compute the feedback composition, where Lines 2 and 6 are performed considering the feedback composition rules. Using this algorithm, the hyper-period of the example shown in Figure 5.4 is computed as $(e{\leftrightarrow}gh^2)(f{\leftrightarrow}hg)$.

5.4 Experimental Results

We implemented Algorithm 2 in SDF3 [65], the tool developed to analyse dataflow models and map them onto multi-processor platforms already introduced in Section 4.3. We used this tool to generate modular SADF models of MRCFs (also introduced in Section 4.3) from the modular models of the up-sampler, down-sampler and filter modules, each of which is obtained by individually analysing the behaviour of their FPGA implementations for a given schedule and window size. We also used the throughput analysis provided in Section 4.1 to compute the throughput of the generated models. The computed throughput values for MRCFs are exact, as SADF models of cyclo-static applications do not require acceptance conditions for their FSM representations (see Section 2.8). The models are generated for square input images with different frame sizes (FS) and different numbers of layers (L) in the structure shown in Figure 5.2. Table 5.2 summarizes the experimental results. For each of these models, we report the throughput (T), in pixels per clock cycle, the hyper-period expression length (EL), in the number of characters used to represent the expression, the time needed to compose the models, i.e., composing time (CT), and the throughput analysis run-time (AT), both in milliseconds. The times are measured on an Ubuntu server with a 3.8Ghz processor.

To compare the results with CSDF models, we generated CSDF graphs of this application by composing the CSDF graphs of the modules. The CSDF analysis runs a dataflow simulation to compute the throughput. For this application, the simulation involves a huge number of actor firings (NF), which has an adverse effect on the analysis time. The results show that the computed throughput values from both SADF and CSDF models are exactly the same, as expected, for every configuration, and that the SADF models are quicker to analyze compared to CSDF models. Moreover, the model construction time is very short, even for such a complex application and, it is only slightly affected by the size of the image or the number of layers. It is worthwhile to mention that a fast analysis is particularly important when called frequently in algorithms such as the throughput-buffering exploration algorithm in the next chapter.

Table 5.2: Analysis on SADF and CSDF models of MRCF

FS	L	T	SADF			CSDF	
			EL	CT	AT	NF	AT
960	3	0.2495	142	1	1	$1.56 \cdot 10^{+7}$	1882
2048	3	0.2496	168	1	1	$7.12 \cdot 10^{+7}$	8571
960	4	0.2495	186	1	1	$1.72 \cdot 10^{+7}$	2070
2048	4	0.2497	210	2	3	$7.86 \cdot 10^{+7}$	9228
960	5	0.2495	246	1	2	$1.77 \cdot 10^{+7}$	2132
2048	5	0.2497	272	2	5	$8.06 \cdot 10^{+7}$	9685

5.5 Related Work

Compositionality of SDF models has been addressed in a number of works [34, 35] by introducing the concept of hierarchical SDF graphs. In a hierarchical SDF graph, a composite SDF actor is perceived as an atomic SDF actor. Such a representation is not compositional, i.e., it might lead to deadlock and/or rate inconsistencies [35]. The authors of [36] introduced a compositional abstraction for SDF actors, called deterministic SDF with shared FIFOs profiles. This allows for modular compilation and code generation for applications modelled as hierarchical SDF graphs.

Skelin et al. [57] propose a $(\max, +)$ algebraic throughput analysis method for hierarchical SDF graphs. The approach exploits the inherent hierarchy in the graph when generating a $(\max, +)$ characterization of the graph. This leads to a considerably faster analysis compared to the analysis performed on the unfolded graph. Another throughput analysis method for hierarchial SDF graphs is provided by Deroui et al. [72]. The authors use the Interface-Based SDF (IBSDF) [73] as the composite actor model. This model extends the semantics of the SDF model to provide a graph composition mechanism based on hierarchical interfaces. An approximate throughput is obtained by constructing a periodic ASAP schedule for the IBSDF graph in a bottom-up approach.

Our work can be considered as a generalization of the works above, since we allow for composite actors with cyclo-static nature. This adds the extra complexity of computing the hyper-period of the composite actor from the periods of its modules. We use SADF to model the composite CSDF actors, where every scenario is represented by a $(\max, +)$ matrix. For conciseness and scalability of the timing analyses, representing periodic behaviours of the composite actors by regular expressions is crucial. Computing the hyper-period of the composite actors which is the core contribution of this chapter, involves composing regular expressions in a specific way. We are not aware of other works that address this problem. We believe it may have interesting applications in other domains as well.

Another difference between this work and some of the works above is that the composite actors in this work communicate through channels that are not necessarily FIFOs. In fact the consumption order of the data on these channels is determined by the sequence of the consumer scenarios and not necessarily by the time the data is produced on the channels. This is necessary to ensure the correct timing of the cyclo-static models.

Other composable abstractions of dataflow models are based on actor interfaces [74]. These interfaces express the relation between sequences of input tokens and sequences of output tokens. The authors provide a composable refinement relation based on the earlier-is-better principle that preserves worst-case bounds on throughput and latency. The compositional temporal analysis model based on these bounds is provided by Hausmans et al. [75]. The authors show that this model can be used for buffer sizing and, it can accommodate latency constraints. In contrast to the works of Geilen et al. [74] and Hausmans et al. [75], we provide a composite model that has a dataflow behaviour as opposed to an abstraction of it.

5.6 Conclusion

We aimed to generate modular models of complex cyclo-static applications via model composition. For the sake of compactness and analysis scalability, we used the SADF MoC as the basic model of the modules. We provided an approach to automatically generate SADF models of a cyclo-static application by composing SADF models of its modules. The results show that such models can be generated in a short time for complex applications. Moreover, the generated models can be quickly analysed compared to the traditional CSDF models. In the next chapter we show that the buffer sizing problem can be addressed using application models that can be quickly analysed for their timing behaviour.

Generalizing the composition method introduced in this chapter to SADF models with non-determinism scenario transitions is a future work direction. One may need to consider the product of two given regular expressions, similar to the product of two FSMs, with an additional constraint of preserving the repetitive patterns of the given regular expressions in the product.

Chapter 6

Throughput-Buffering Trade-Off Analysis

In real-time streaming applications, buffers represent storage spaces that are used to store the data communicated between different tasks in the application. The capacities of the buffers influence the throughput, by altering the waiting times for tasks that need to read or write data from or to the buffers (see Section 1.2.3 for a realistic example). The buffers are often realized using memory blocks or hardware FIFO queues. To minimize the memory usage, we need to find the minimum capacities for buffers to execute an application under a given throughput constraint. A throughput-buffering trade-off analysis computes all optimal points on the space of throughput vs buffer resources, from which the minimum buffer capacities for a given throughput constraint can be obtained.

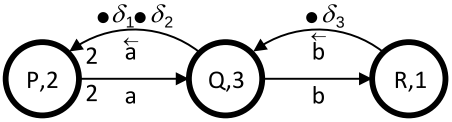
Finding the minimal buffer sizes for SDF and CSDF models to satisfy a minimum throughput constraint is known to be a complex problem (it is NP-complete [76]). Heuristic approaches are proposed to obtain near optimal solutions for this problem [77, 78]. Stuijk et al. [29] provide an exact throughput-buffering trade-off analysis that terminates in a reasonable time for SDF and CSDF models of realisitc applications. The algorithm uses a *Design Space Exploration (DSE)* scheme that prunes the design space (without losing any optimal points) during the exploration. The approach uses a throughput analysis algorithm to compute the throughput of the model, and at the same time find the throughput-limiting buffers. Further explorations are performed only for these buffers. This prunes the exploration space and makes it smaller as the exploration continues.

In Chapter 4, we provided timing analysis techniques for SADF models. In this chapter, we provide the first throughput-buffering trade-off analysis for applications modelled as SADF. Since SADF generalizes SDF, finding optimal throughput-buffering trade-offs for SADF models is at least as complex as that for SDF models. We also use a DSE approach. As an enabling contribution, we propose a novel throughput computation method for SADF models that obtains the throughput and at the same time determines the buffers that limit the throughput of the model. This is an important contribution, because it enables the application of smart exploration schemes in performing efficient buffer sizing for SADF models. For example, the method of Stuijk et al. [29] or Hendriks et al. [78] can be generalized to SADF models using this contribution. We integrate our throughput analysis in a DSE scheme similar to the one provided by Stuijk et al. [29] to find all optimal throughput-buffering trade-offs. We prove that the proposed DSE finds all optimal points in the space of the throughput-buffer size trade-offs. We show the applicability of our approach using several realistic applications modelled as SADF. This chapter is based on publication [42].

6.1 The Main Idea

As mentioned, the DSE approach for throughput-buffering trade-off analysis requires that for a given storage distribution, the buffers that limit the throughput, are identified. In this section we explain the key idea behind a technique we provide to obtain the throughput limiting buffers in SADF models. The throughput analysis given by Geilen et al. [31] (see Section 2.8), and consequently the throughput analysis given in Section 4.1 does not give an indication of the buffer channels that limit the throughput, since state matrices of the scenarios do not include the notion of actors and channels. Therefore, those analyses cannot be directly used in the DSE.

The idea is to arm the state matrices introduced in Section 2.7.1 with information on the buffers. The MPA (see Section 2.8) built from the new matrices carries the buffer information on every state transition. This enables us to translate the critical cycles in the MPA back to a set of buffer channels we refer to as *critical buffer channels*. We illustrate this procedure on an SADF model which repeatedly executes scenario κ, shown in Figure 6.1. We call this model the simple SADF model in the remainder, as it contains only one scenario. In the SDFG of this scenario, channels a and b are buffer channels, and $\overleftarrow{a}$ and $\overleftarrow{b}$ are their capacity channels, respectively (see Section 2.9).



Reps: $[\mathsf{P},\mathsf{Q},\mathsf{R}] \mapsto [1,2,2]$

ITs: $\overleftarrow{a}(\delta_1,\delta_2)$, $\overleftarrow{b}(\delta_3)$

FTs: $\overleftarrow{a}(\delta_1,\delta_2)$, $\overleftarrow{b}(\delta_3)$

Reward: 1

Figure 6.1: Scenario κ.

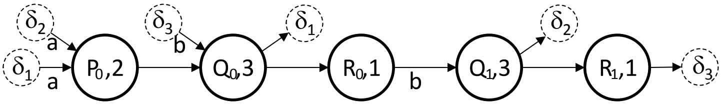


Figure 6.2: The dependency graph of scenario κ.

We extract additional information from the symbolic simulation when transforming scenario κ to its state matrix. We explain this by defining an *dependency graph* of the scenario. The dependency graph shows the dependencies between actor firings defined by the scenario. The dependency graph of scenario κ is shown in Figure 6.2. In this graph large nodes represent actor firings. A directed edge from firing A_i to firing B_j indicates that the j^{th} firing of actor B can start when the i^{th} firing of actor A is completed. If an edge represents a firing dependency that is caused by a capacity channel, the corresponding buffer is mentioned under the edge. The production and consumption of the initial tokens by the firings are shown by smaller nodes.

By backtracking the (longest) path from the production of every token to the consumption of every token, we can obtain the state matrix $\boldsymbol{G}_\kappa$, as well as the set of buffers that affect every element of this matrix. The result is the *Extended Matrix Representation (EMR)* of the scenario where every element of the matrix is a pair that contains the time-stamp component and the buffer-set component. The EMR $\boldsymbol{G}^e_\kappa$ for scenario κ is as follows.

$$\boldsymbol{G}^e_\kappa = \begin{bmatrix} (5,\{a\}) & (5,\{a\}) & (3,\{b\}) \\ (9,\{a,b\}) & 9,\{a,b\} & (7,\{b\}) \\ (10,\{a,b\}) & (10,\{a,b\}) & (8,\{b\}) \end{bmatrix}$$

Observe, for instance, that buffer b affects every path to δ_2 and δ_3 in Figure 6.2. Therefore, the buffer-set components of the elements in the second and third row of the matrix $\boldsymbol{G}^e_\kappa$, all include b. The MPA of the simple SADF example can be built from the EMR of scenario κ similar to the way explained in

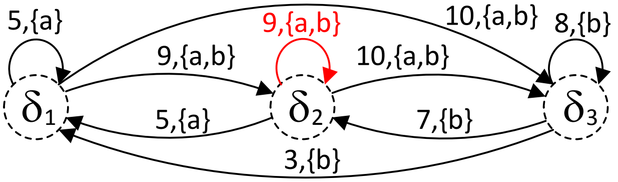


Figure 6.3: The EMPA of the simple SADF model.

Section 2.8, except that now the MPA edge labels contain scenario rewards, time-stamp components, i.e. the delays, and buffer-set components. We refer to such an MPA as the *Extended* $(\max, +)$ *Automaton (EMPA)*. The EMPA of the simple SADF is shown in Figure 6.3. The reward component of every edge label is 1. We do not show the reward components on the edge labels to avoid cluttering. The critical cycle of the EMPA is coloured in red. Since buffers a and b are on the critical cycle, they are identified as critical. The throughput can only increase by increasing at least on of these buffers.

In Section 6.2, we extend the symbolic simulation method provided by Geilen et al. [31] to directly generate the EMR of scenarios (without the need to first create the dependency graph). Section 6.3 defines the critical buffers in SADF models. In Section 6.4, we provide our throughput-buffering trade-off analysis for SADF. The experimental results are provided in Section 6.5. Section 6.6 discusses the related work. Finally, we conclude this chapter in Section 6.7.

6.2 The Extended Matrix Representation

This section introduces the extended matrix representation of a scenario execution that does not deadlock. Then it provides an algorithm to compute it for a given SDF scenario. Consider a scenario with a set B of buffer channels in its scenario graph. The elements of the EMR are pairs $(t, \tilde{B})$. Component $t \in \mathbb{R}^{+}_{-\infty}$ indicates the relation between the initial and final state vectors (as in the traditional $(\max, +)$ characterization). Component $\tilde{B} \subseteq B$ is the set of buffer channels that affect the timing relation t.

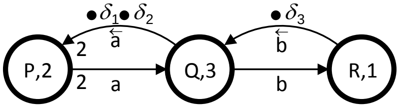
To generate the EMR, we propose an extended symbolic simulation during which, tokens are assigned with vectors $\boldsymbol{v} = [\ (t_1, \tilde{B}_1) \quad \cdots \quad (t_m, \tilde{B}_m)\ ]$, where t_n are the time-stamps of the token and $\tilde{B}_n$ are the sets of buffer channels that affect their respective time-stamps t_n. We refer to $\boldsymbol{v}$ as the *Extended Symbolic Time-stamp Vector (ESTV)*. To improve readability, we refer to t_n and $\tilde{B}_n$ of the ESTV of a token τ, as t_n and $\tilde{B}_n$ of τ. We introduce functions $t_n(\tau)$ and $\tilde{B}_n(\tau)$ that map token τ to t_n and $\tilde{B}_n$ of its ESTV.

We illustrate the proposed simulation on the example SDF scenario. Figure 6.4 shows the simulation steps. Figure 6.4a shows the initial distribution of the tokens and their ESTVs. Figures 6.4b-6.4f respectively show the distribution of tokens after a firing according to the sequence $\bar{a} = \mathsf{PQRQR}$ of actor firings. Since the tokens are initial, their time-stamps are not affected by any buffer and therefore, the buffer-sets of all ESTVs are empty. The time-stamp components of the ESTVs are not discussed since they are exactly the same as in the traditional symbolic simulation (see Section 2.7.1).

In Figure 6.4b, P has consumed δ_1 and δ_2, fired, and produced τ_1 and τ_2. To compute $\tilde{B}_n(\tau_1)$, we first need to find out, which of the consumed tokens determine the time-stamp $t_n(\tau_1)$. According to actor firing semantics, consumed tokens with the latest time-stamp determine the time-stamp of the produced token. For example, $t_1(\tau_1)$ is determined by $t_1(\delta_1)$, since $t_1(\delta_1) = 0$ is later than $t_1(\delta_2) = -\infty$. We say that δ_1 is the dominating token for $t_1(\tau_1)$. Since $t_1(\tau_1)$ is dominated by $t_1(\delta_1)$ and, δ_1 is on the capacity channel of a, we can conclude that $t_1(\tau_1)$ is influenced by a. Therefore, $\tilde{B}_1(\tau_1) = \{a\}$. Similarly one can conclude that, $t_2(\tau_1)$ is dominated by δ_2 and therefore, $\tilde{B}_2(\tau_1) = \{a\}$. Since $t_3(\tau_1) = -\infty$, this time-stamp is not affected by anything, and $\tilde{B}_3(\tau_1)$ is empty. Since τ_2 is produced by the same firing that produced τ_1, they have the same ESTV $\boldsymbol{v}_{\tau_2} = \boldsymbol{v}_{\tau_1}$.

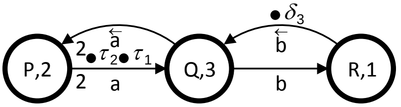
In Figure 6.4c, Q has consumed τ_1 and δ_3, fired, and produced τ_3 and the final token δ_1 (not to be confused with the initial token δ_1). For $t_1(\delta_1)$ and $t_2(\delta_1)$, τ_1 is the dominating token because, $t_1(\tau_1) = 2 > t_1(\delta_3) = -\infty$ and $t_2(\tau_1) = 2 > t_2(\delta_3) = -\infty$. Since $t_1(\delta_1)$ is dominated by τ_1, it is influenced by all buffers that influence $t_1(\tau_1)$. This means, the buffers that belong to $\tilde{B}_1(\tau_1)$, also belong to $\tilde{B}_1(\delta_1)$. Similarly, the buffers that belong to $\tilde{B}_2(\tau_1)$ also belong to $\tilde{B}_2(\delta_2)$. Token δ_3 is the dominating token for $t_3(\delta_1)$. Since δ_3 is on the capacity channel of buffer b, $\tilde{B}_3(\delta_1) = \{b\}$. Figures 6.4d-6.4f show the rest of the simulation steps. Figure 6.4f shows the ESTVs of the tokens remaining on the channels after the scenario is completely executed. By collecting these vectors in a matrix, we obtain the extended matrix $\boldsymbol{G}^e_\kappa$.

We formalize our symbolic simulation by defining the ESTV of a token ρ that is produced by a symbolic firing of an actor a with execution time $e(a)$, consuming a set T of tokens. Identical to the traditional symbolic firing, $t_n(\rho)$,



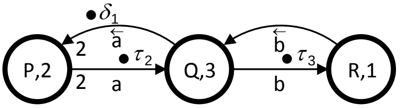
$$\begin{aligned}
\boldsymbol{v}_{\delta_1} &= \begin{bmatrix} (0,\{\}) & (-\infty,\{\}) & (-\infty,\{\}) \end{bmatrix} \\
\boldsymbol{v}_{\delta_2} &= \begin{bmatrix} (-\infty,\{\}) & (0,\{\}) & (-\infty,\{\}) \end{bmatrix} \\
\boldsymbol{v}_{\delta_3} &= \begin{bmatrix} (-\infty,\{\}) & (-\infty,\{\}) & (0,\{\}) \end{bmatrix}
\end{aligned}$$

(a) Initial distribution



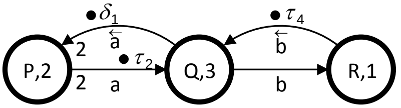
$$\begin{aligned}
\boldsymbol{v}_{\tau_1} &= \begin{bmatrix} (2,\{a\}) & (2,\{a\}) & (-\infty,\{\}) \end{bmatrix} \\
\boldsymbol{v}_{\tau_2} &= \begin{bmatrix} (2,\{a\}) & (2,\{a\}) & (-\infty,\{\}) \end{bmatrix} \\
\boldsymbol{v}_{\delta_3} &= \begin{bmatrix} (-\infty,\{\}) & (-\infty,\{\}) & (0,\{\}) \end{bmatrix}
\end{aligned}$$

(b) After the firing of P



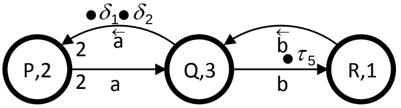
$$\begin{aligned}
\boldsymbol{v}_{\delta_1} &= \begin{bmatrix} (5,\{a\}) & (5,\{a\}) & (3,\{b\}) \end{bmatrix} \\
\boldsymbol{v}_{\tau_2} &= \begin{bmatrix} (2,\{a\}) & (2,\{a\}) & (-\infty,\{\}) \end{bmatrix} \\
\boldsymbol{v}_{\tau_3} &= \begin{bmatrix} (5,\{a\}) & (5,\{a\}) & (3,\{b\}) \end{bmatrix}
\end{aligned}$$

(c) After the first firing of Q



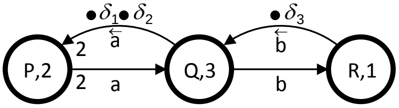
$$\begin{aligned}
\boldsymbol{v}_{\delta_1} &= \begin{bmatrix} (5,\{a\}) & (5,\{a\}) & (3,\{b\}) \end{bmatrix} \\
\boldsymbol{v}_{\tau_2} &= \begin{bmatrix} (2,\{a\}) & (2,\{a\}) & (-\infty,\{\}) \end{bmatrix} \\
\boldsymbol{v}_{\tau_4} &= \begin{bmatrix} (6,\{a\}) & (6,\{a\}) & (4,\{b\}) \end{bmatrix}
\end{aligned}$$

(d) After the first firing of R



$$\begin{aligned}
\boldsymbol{v}_{\delta_1} &= \begin{bmatrix} (5,\{a\}) & (5,\{a\}) & (3,\{b\}) \end{bmatrix} \\
\boldsymbol{v}_{\delta_2} &= \begin{bmatrix} (9,\{a,b\}) & (9,\{a,b\}) & (7,\{b\}) \end{bmatrix} \\
\boldsymbol{v}_{\tau_5} &= \begin{bmatrix} (9,\{a,b\}) & (9,\{a,b\}) & (7,\{b\}) \end{bmatrix}
\end{aligned}$$

(e) After the second firing of Q



$$\begin{aligned} \boldsymbol{v}_{\delta_1} &= \begin{bmatrix} (5,\{a\}) & (5,\{a\}) & (3,\{b\}) \end{bmatrix} \\ \boldsymbol{v}_{\delta_2} &= \begin{bmatrix} (9,\{a,b\}) & (9,\{a,b\}) & (7,\{b\}) \end{bmatrix} \\ \boldsymbol{v}_{\delta_3} &= \begin{bmatrix} (10,\{a,b\}) & (10,\{a,b\}) & (10,\{b\}) \end{bmatrix} \end{aligned}$$

(f) After the second firing of R

Figure 6.4: Token distributions in the symbolic simulation of scenario κ.

is defined as follows.

$$t_n(\rho) = \max_{\tau \in T} t_n(\tau) + e(a) \tag{6.1}$$

To define $\tilde{B}_n(\rho)$, we first need to define the dominating tokens for $t_n(\rho)$. The dominating tokens for $t_n(\rho)$ are all consumed tokens with the latest t_n. Therefore, t_n of the dominating tokens determines $t_n(\tau)$. In other words, if $t_n(\tau) + e(a) = t_n(\rho)$, then τ is a dominating token for $t_n(\rho)$. The set $T_n(\rho)$ of the dominating tokens for $t_n(\rho)$ is defined as follows.

$$T_n(\rho) = \{\tau \in T \mid t_n(\tau) + e(a) = t_n(\rho)\} \tag{6.2}$$

A buffer b belongs to $\tilde{B}_n(\rho)$ if at least one of the following two conditions holds. First, if a token that belongs to T_n is currently located on the capacity channel of b. Second, if b belongs to $\tilde{B}_n(\tau)$ for any $\tau \in T_n$. Let $bf(\tau, b)$ be a function that returns *true* if token τ is on the capacity channel of buffer b and *false* otherwise. The function $\tilde{B}_n(\rho)$ is defined as follows.

$$\tilde{B}_n(\rho) = \left\{ b \in B \mid \exists_{\tau \in T_n(\rho)} [b \in \tilde{B}_n(\tau) \vee bf(\tau, b)] \right\} \tag{6.3}$$

Algorithm 3 sketches the proposed symbolic simulation method to obtain the EMR of a scenario execution that does not deadlock. Line 1 assigns ESTVs to all initial tokens and stores them in a set Y. In Line 2, a valid actor firing sequence that complies with repetition vector of the scenario is generated and stored in $\bar{a}$ (such a schedule exists as the execution of the given scenario does not deadlock). Starting from the first firing in $\bar{a}$, for every firing in the sequence the following actions are performed. The actor responsible for the firing is recognized. The actor consumes a set T of tokens from Y and fires. The consumed tokens are removed from Y (Lines 4-6). The ESTVs of the

ALGORITHM 3: Compute the EMR of an SDF scenario

```
   Input: An SDF scenario $F$ with $I$ initial tokens
   Output: EMR $\boldsymbol{G}^e$
1  $Y = \{(\delta, \mathrm{ESTV}(\delta)) \mid \delta \in \mathit{InitialTokens}(F)\}$;
2  $\bar{a} = \mathrm{computeSeqSchedule}(F)$;
3  for $j = 1$ to $\mathrm{length}(\bar{a})$ do
4      $a = \mathrm{getActor}(\bar{a}, j)$;
5      Fire $a$, let $T \subseteq Y$ be the tokens consumed by firing $a$ and
         $\mathit{ProducedTokens}$ be the tokens produced by firing $a$;
6      $Y = Y \setminus T$;
7      for $n = 1$ to $I$ do
8          $t_n = \max_{\tau \in T} t_n(\tau) + e(a)$;
9          $T_n = \{\tau \in T \mid t_n(\tau) + e(a) = t_n\}$;
10         $\tilde{B}_n = \left\{ b \in B \mid \exists_{\tau \in T_n} [b \in \tilde{B}_n(\tau) \vee bf(\tau, b)] \right\}$;
11     end
12     $\boldsymbol{v} = [\ (t_1, \tilde{B}_1) \quad \cdots \quad (t_I, \tilde{B}_I)\ ]$;
13     $V = \{(\rho, \boldsymbol{v}) \mid \rho \in \mathit{ProducedTokens}\}$;
14     $Y = Y \cup V$;
15 end
16 $\boldsymbol{G}^e = [\boldsymbol{v}(\delta)]$ for all $\delta \in Y$;
```

consumed tokens and the execution time of the actor are used in Lines 8-10 to compute the elements of the ESTV $\boldsymbol{v}$ according to equations Eqs. 6.1 and 6.3. $\boldsymbol{v}$ is constructed from its elements in Line 12. In Line 13, the tokens are produced and assigned with $\boldsymbol{v}$. These tokens are added to Y in Line 14. Finally, the ESTVs of the tokens in Y are collected to generate the EMR of the scenario (Line 16).

6.3 The Critical Buffers of an FSM-SADF

This section defines the critical buffers of an SADF F that incorporates a storage distribution d, which we denote as F_d. In Section 2.9, we introduced common buffers of SADF models. A common buffer corresponds to a buffer channel in the scenario graph of every scenario in the given SADF model. We denoted the corresponding capacity channel of a common buffer u in a scenario s, with $b_s(u)$. Moreover, with a distribution d we associated a capacity with every common buffer u in the given SADF model.

A buffer channel can be critical in two distinct cases. The first case is when a buffer causes a deadlock in the execution of a scenario. In a deadlock state, a critical buffer channel creates a cycle of actor firings that are dependent on each other, but non of them are enabled. In the second case, there are no scenario deadlocks and the SADF has a positive throughput, however, this throughput is limited by an actor, the firings of which are delayed because of waiting for empty space in a buffer. A buffer that limits the throughput of an SADF model in this way is critical. We first provide a definition for the case that at least one of the scenarios in F_d deadlocks due to insufficient storage space. Then we provide a definition for an F_d with a deadlock free execution.

For the deadlock case we use a definition from Stuijk et al. [29]. For every scenario that deadlocks under the given storage distribution, a Deadlock Dependency Graph (DDG) is determined. The DDG (D, E) is generated from the scenario graph when it is in the deadlock state. The set D of nodes contains a node for every actor in the scenario graph. The set E of edges contains an edge from an actor P to actor Q, labelled with c, if and only if firing of P is prohibited by a lack of tokens on a channel c from Q to P. Buffers are critical when their capacity channels appear in cycle of such dependencies in the DDG.

Definition 6.1. *A buffer channel b in the scenario graph of a deadlock scenario is critical if the is an edge labelled with $\overleftarrow{b}$ in a cycle of the DDG of the scenario. A common buffer u of an SADF is critical if $b = b_s(u)$ is critical for some deadlock scenario s.*

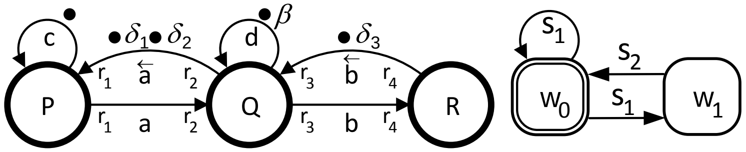
For the deadlock free F_d, we generate the extended $(\max, +)$ automaton (an example is shown in Figure 6.3). The EMPA is obtained by replacing the matrices $\boldsymbol{G}(s)$ with $\boldsymbol{G}^e(s)$ in the MPA generation method introduced in Section 2.8. A critical cycle of the EMPA is a cycle with the maximum cycle ratio. The definition of critical buffers for the deadlock free case is as follows.

Definition 6.2. *A buffer channel b in the scenario graph of a scenario in a deadlock free F_d is critical if it is in a critical cycle of the EMPA of F_d. A common buffer u of an SADF is critical if $b = b_s(u)$ is critical for a scenario s.*

In the next section, we use these two definitions in a DSE to obtain the throughput-buffering trade-offs for a given SADF model.

6.4 Design Space Exploration

Using the definitions in Section 6.3, we can find the trade-offs between the distribution size and the throughput, i.e., the Pareto space. Consider an FSM-



(a) Scenario graphs. (b) FSM.

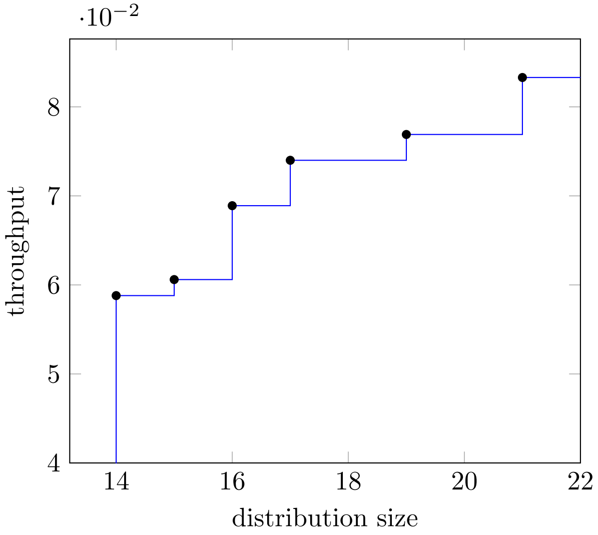
Scenarios	s_1	s_2
ETs	$e(P) = 2, e(Q) = 3, e(R) = 1$	$e(P) = 1, e(Q) = 5, e(R) = 2$
Rates	$r_1 = 3, r_2 = 4, r_3 = 2, r_4 = 6$	$r_1 = 1, r_2 = 2, r_3 = 4, r_4 = 6$
Reward	1	1
Reps	$[P, Q, R] \mapsto [4, 3, 1]$	$[P, Q, R] \mapsto [6, 3, 2]$
ITs	$\overleftarrow{a}(\delta_1, \delta_2), \overleftarrow{b}(\delta_3), c(\alpha), d(\beta)$	$\overleftarrow{a}(\delta_1, \delta_2), \overleftarrow{b}(\delta_3), c(\alpha), d(\beta)$
FTs	$\overleftarrow{a}(\delta_1, \delta_2), \overleftarrow{b}(\delta_3), c(\alpha), d(\beta)$	$\overleftarrow{a}(\delta_1, \delta_2), \overleftarrow{b}(\delta_3), c(\alpha), d(\beta)$

(c) Scenario specifications.

Figure 6.5: An FSM-SADF example for buffer sizing.

SADF example shown in Figure 6.5. Figure 6.6 shows the Pareto space obtained for this example. The interesting points on the throughput-distribution size trade-off space are the points that have the minimum distribution size for a given throughput, i.e., the Pareto points. Formally, for an SADF, a storage distribution d with throughput Thr is minimal if and only if there is no storage distribution d' with throughput Thr' such that $|d'| \leq |d|$ and $Thr' > Thr$ or $|d'| < |d|$ and $Thr' \geq Thr$.

The table below the figure lists all Pareto points from the lowest positive throughput to the maximum achievable throughput. The table shows for every Pareto point also the capacity of the buffers u_a and u_b and the scenario sequence that limits the throughput. For all distributions for which the SADF deadlocks the throughput is considered to be 0. Therefore the zero distribution $\langle 0, 0 \rangle$ is the minimal distribution for zero throughput. Obviously, zero throughput is not an interesting design point, thus we do not show it in Figure 6.6. Observe that the smallest distribution with a throughput larger than zero is $6 + 8 = 14$ tokens, as scenario s_1 deadlocks with any capacity lower than 6 tokens for the buffer a and, scenario s_2 deadlocks with any capacity lower than 8 tokens for the buffer b. The sequence $(s_1 s_2)^\omega$ limits the maximum achievable throughput to $1/12$. The maximal throughput can be achieved with



Thr	distribution	limiting sequence
0.058	$\langle 6, 8\rangle$	s_1^ω
0.060	$\langle 7, 8\rangle$	s_1^ω
0.068	$\langle 8, 8\rangle$	$(s_1 s_2)^\omega$
0.074	$\langle 9, 8\rangle$	$(s_1 s_2)^\omega$
0.076	$\langle 9, 10\rangle$	$(s_1 s_2)^\omega$
0.083	$\langle 9, 12\rangle$	$(s_1 s_2)^\omega$

Figure 6.6: Throughput-buffering trade-offs for Figure 6.5.

the minimum distribution size of $9 + 12 = 21$ as shown in the figure.

Algorithm 4 sketches the adapted DSE. The exploration is based on the monotonicity property of SADF [55]. Monotonicity ensures that an increase in the capacity of a buffer cannot lower the throughput. The input is an FSM-SADF F and the output is the set P of all pairs (d, Thr) such that d is a minimal distribution and Thr is its throughput. In the first line, the maximal throughput Thr_{max} is computed by removing all capacity channels from scenario graphs and using the throughput analysis given in Section 2.7.1 (we assume that the modelled application has a bounded throughput, otherwise, its trade-off space is not finite). The set I of the storage distributions that will be explored by the algorithm is initialized with zero distribution $\langle 0, \ldots, 0 \rangle$ and, P is initially empty (Line 2). Lines 4-19 are repeated until all minimal distributions with Thr_{max} are in P. A distribution d with the smallest size is picked and then removed from I (Line 4). F_d is created from d and F in Line 5. If F_d is deadlock free, the EMPA of the F_d is created in Line 7. Using an MCR analysis on the EMPA, the throughput and a critical cycle are obtained and stored in Thr and C, respectively (Line 8). If F_d deadlocks, then the algorithm generates the DDG of F_d, finds a cycle C in the DDG and sets $\mathit{Thr} = 0$ (Lines 10-11). The pair (Thr, d) is added to P in Line 13. A set U of critical buffers are obtained from C in Line 14 using the definitions provided in Section 6.3. For every $u \in U$, a new distribution d' is generated from d by increasing the capacity of u by one step, then it is added to the distributions pool I (Lines 16-18). The step size for a buffer is the minimum number of tokens that if added to the capacity of the buffer, it may break actor firing dependencies on that buffer. We define the step size of a common buffer as the minimum of the input and output rates of the corresponding buffer channel over all scenarios. In Figure 6.5, the step size for u_a and u_b are $\min\{3, 4, 1, 2\} = 1$ and $\min\{2, 6, 4, 6\} = 2$, respectively. In the final step (Line 21), all the non-minimal distributions are removed from P, after which it is returned.

To prove the correctness of the algorithm, we proceed with three lemmas as follows (by an adaptation from Stuijk et al. [29]). The first lemma states that at least one of the critical buffers in a deadlocking graph needs to be increased to resolve the deadlock.

Lemma 6.1. *Given a storage distribution d_i with throughput $\mathit{Thr}_i > 0$, for any storage distribution $d_j \preceq d_i$ for which F_{d_j} deadlocks in some scenario s, in any cycle in the DDG of s, there exists a capacity channel $\overleftarrow{b} = c_s(u)$ for which, $d_i(u) > d_j(u)$.*

Proof. Since $\mathit{Thr}_i > 0$, F_{d_i} is deadlock free. Since F_{d_i} is deadlock free, but

ALGORITHM 4: Find all minimal storage distributions

```
Input: An FSM-SADF $F$
Output: The set $P$ of all pairs $(d, Thr)$ s.t. $d$ are minimal
$Thr_{max}$ = computeThrMax($F$);
$I = \{\langle 0, \cdots, 0\rangle\}$; $P = \{\}$;
repeat
    $d$ = getSmallestDist($I$); $I = I \setminus \{d\}$;
    $F_d$ = createFSMSADF($F, d$);
    if $F_d$ is deadlock free then
        $M_d$ = computeEMPA($F_d$);
        $(Thr, C)$ = computeThrAndACriticalCycle($M_d$);
    else
        $G_d$ = computeDDG($F_d$);
        $Thr = 0$; $C$ = findACycle($G_d$);
    end
    $P = P \cup \{(d, Thr)\}$;
    $U$ = getCriticalBuffers($C$);
    foreach $u \in U$ do
        $d'$ = copyDist($d$);
        $d'(u) = d(u)$ + step($u$);
        $I = I \cup \{d'\}$;
    end
until $\exists_{(d, Thr) \in P}[Thr = Thr_{max} \wedge \nexists_{d' \in I}[|d'| = |d|]]$;
$P$ = removeNonOptimalDists($P$);
```

F_{d_j} deadlocks in scenario s, d_i has resolved any cycle in DDG of scenario s in F_{d_j}. To resolve a cycle in the DDG of F_{d_j}, the capacity of at least one buffer channel that its corresponding capacity channel is on the cycle, must be increased. For every capacity channel $\overleftarrow{b} = c_s(u)$ that the capacity of its corresponding buffer must be increased, we have $d_i(u) > d_j(u)$. $\square$

The second lemma states that at least one of the critical buffers needs to be increased to increase the throughput in a non-deadlocking SADF model with a throughput which is less than the maximum achievable throughput.

Lemma 6.2. *Given a storage distribution d_i with throughput Thr_i, for any storage distribution $d_j \preceq d_i$ with throughput $0 < Thr_j < Thr_i$, in any critical cycle in the EMPA of F_{d_j} there exists a buffer channel $b = b_s(u)$ for which, $d_i(u) > d_j(u)$.*

Proof. Since d_j has a positive throughput, it is deadlock free and F_{d_j} has an

EMPA. Since F_{d_i} has a higher throughput than F_{d_j}, we can conclude that any critical cycle in the EMPA of F_{d_j} has been resolved in the EMPA of F_{d_i}. To resolve a critical cycle in the EMPA of F_{d_j} the capacity of at least one buffer channel on the critical cycle must be increased. Note that any critical cycle in the EMPA of F_{d_j} contains at least one edge with a non-empty set of buffers channels (otherwise Thr_j would be maximal, contradicting $Thr_j < Thr_i$). For every buffer channel $b = b_s(u)$ the capacity of which must be increased, we have $d_i(u) > d_j(u)$. ☐

The third lemma states that from a given distribution, Algorithm 4 explores at least one distribution which might have a higher throughput compared to the throughput of the given distribution. Later we use this lemma to prove that from a given distribution, all distributions which may increase the throughput are explored by Algorithm 4.

Lemma 6.3. *Consider a storage distribution* d_i *with throughput* Thr_i *and a storage distribution* d_j *such that* $d_j \preceq d_i$ *with throughput* $Thr_j < Thr_i$. *From* d_j, *Algorithm 4 explores a storage distribution* d_k *for which* $d_j \preceq d_k \preceq d_i, |d_j| < |d_k|$ *and* $Thr_j \leq Thr_k \leq Thr_i$.

Proof. If F_{d_j} deadlocks, according to Lemma 6.1 and, if it is deadlock free, according to Lemma 6.2, there exists a buffer u in the set U of critical buffers such that $d_i(u) > d_j(u)$. Therefore, the capacity of u in d_j can be increased before the capacity of u becomes equal to the capacity of u in d_i. Since u is critical, Algorithm 4 increases the capacity of u which results in a new distribution d_k. As the capacity of u is increased but not beyond its capacity in d_i, it holds that $d_j \preceq d_k \preceq d_i$ and $|d_j| < |d_k|$. From the monotonicity property it holds that $Thr_j \leq Thr_k \leq Thr_i$. ☐

Theorem 6.1. *Algorithm 4 obtains all minimal storage distributions in* P.

Proof. Let d_i be a minimal storage distribution with throughput Thr_i. We show that the algorithm explores d_i starting from any distribution $d_j \preceq d_i$ already explored by the algorithm. We show by strong induction that d_i is explored from any d_j, such that $|d_i| - |d_j| = n$ for any non-negative integer n. The base case, $n = 0$, is trivial, because $|d_i| = |d_j|$ and $d_j \preceq d_i$ means that $d_i = d_j$, i.e., d_i is explored. Now let's assume d_i is explored from any explored d_j such that $|d_i| - |d_j| = k$ and $0 \leq k \leq n$. We need to show that the algorithm also explores d_i from any explored distribution d_j such that $|d_i| - |d_j| = n+1$. Lemma 6.3 states that from a distribution d_j, a distribution $d_k \succeq d_j$ is explored such that $|d_k| > |d_j|$. According to the algorithm, d_k is generated from d_j by increasing the capacity of at least one buffer by a step of

Table 6.1: Experimental results on SADF models

	WLAN	MP3 Decoder	SUSAN	Figure 6.5
nrBuffers	8	46	4	2
nrParetoPoints	2	3	7	6
nrMinDist	2	3	7	6
nrVisitedDist	3	32	12	10
maxThr($\times 10^{-4}$)	2.5	0.0017	0.066	830
distSizeMaxThr	10	179	20	21
minPosThr($\times 10^{-4}$)	2.4	0.0016	0.012	5.80
distSizeMinThr	8	172	8	14
RunTime Alg. 4	3ms	4559ms	11ms	19ms

some size s. Therefore $|d_k| - |d_j| = s$. By comparing $|d_i|$ and $|d_k|$ we conclude that $|d_i| - |d_k| = n + 1 - s$. Since $s \geq 1$, $n + 1 - s \leq n$ and, according to induction hypothesis, d_k is explored by the algorithm. Since d_i is explored from d_k, and d_k is explored from d_j, we conclude that d_i is explored from d_j. The algorithm starts from the zero distribution. Since $\langle 0, \cdots, 0\rangle \preceq d_i$ for any d_i, the algorithm explores any minimal distribution d_i. $\square$

6.5 Evaluation

We implemented Algorithm 3 and Algorithm 4 in the SDF3 tool [65]. We applied Algorithm 4 to SADF models of several realistic applications. The set of SADF application models contain models an MP3 decoder [55], an edge detection algorithm known as SUSAN [79] and a WLAN [55]. We also transformed a set of SDF application models to SADF (the SADF contains a single scenario that is defined as the set of actor firings in one iteration of the SDFG) to apply our analysis and compare the results with the existing buffer sizing techniques. The set of SDF applications include an H.263 decoder [29], a Modem [22], a sample rate converter [22] and a satellite receiver [80].

The buffer sizing results for SADF and SDF models are shown in Tables 6.1 and 6.2, respectively. For every model, we report the number of sized buffers, Pareto points, minimum distributions, visited storage distributions, the maximum and minimum positive throughput and the size of minimal distributions for which the throughput numbers are obtained. We have excluded the trivial zero storage distribution from the number of Pareto points, minimum distri-

Table 6.2: Experimental results on SDF models

	H.263 Decoder	Modem	Sample Rate	Satellite
nrBuffers	3	35	10	48
nrParetoPoints	36	3	3	2
nrMinDist	146	3	3	2
nrVisitedDist	193	7	5	6
maxThr($\times 10^{-4}$)	0.03	620	10	7.5
distSizeMaxThr	1224	72	44	1586
minPosThr($\times 10^{-4}$)	0.015	320	9.2	9.4
distSizeMinThr	1189	70	42	1588
RunTime Alg. 4	1296s	74ms	545ms	419s
RunTime [29]	349ms	1ms	8ms	514ms

butions and visited storage distributions, as it is not an interesting design point. Note that the number of minimal distributions might be more than the number of Pareto points, as there might be distributions with the same size but different configurations in a Pareto point. The run-time of Algorithm 4 is reported for all models. The run-times include all function calls in the algorithm including the run-time of Algorithm 3, as it is called by Algorithm 4. For SDF graphs, we compared the results of our buffer sizing algorithm with the SDF buffer sizing method provided by Stuijk et al. [29]. We observed that the results are identical. We also report the run-time of both algorithms and observe that for the SDF graphs their algorithm is generally faster. All run-times are measured on an Ubuntu server with a 3.8Ghz processor.

Observe that the run-time of Algorithm 4 is affected by a number of parameters. First, the input model should have a bounded throughput, otherwise, its trade-off space is not finite and the algorithm does not terminate. If the model deadlocks even with unbounded buffers, then the model is incorrect. This case is detected at the first step of the algorithm, since it computes max throughput, which will be 0. For all other cases the algorithm returns all optimal distributions. The complexity analysis for those cases is as follows.

To compute the throughput of a distribution we first need to compute the EMR of all scenarios. The time complexity of this step is $\mathcal{O}(|S| \cdot |I|^2 \cdot |F|)$ where $|S|$, $|I|$ and $|F|$ are the number of scenarios, initial tokens and actor firings respectively. Since the distributions are modelled by adding initial tokens on the capacity channels, computing the EMRs scales quadratically in the distribution size (the total number of added tokens equals the size of the distribution).

The time complexity of an MCR analysis and obtaining a critical cycle (Karp [69]) on a MPA is $O(|R| \cdot |E|)$, where $|R|$ and $|E|$ are the number of nodes and edges of the MPA. The MPA of an F_d by definition has a number of nodes $|R| = |W| \times |I|$ and edges $|E| = |S| \times |I|^2$, where $|W|$ is the number of FSM states. Consequently, MCR analysis has a cubic time complexity in the size of the distribution. Therefore computing the throughput of distribution has a worst-case cubic complexity in the size of the distribution. The results confirm that the analysis time is longer for the models with larger distributions compared to the models with smaller distributions.

6.6 Related Work

The problem of finding the minimum buffer requirements to execute an SDF with the maximum, minimum or a given throughput has been addressed in a number of works [81][82][29]. Hwang et al. [81] and Govindarajan et al. [82] formalize the maximum throughput version of the question as a linear programming problem. The formalization provided by Hwang et al. [81] consider time-constrained and resource-constrained schedules and it is applicable to acyclic SDF only. Govindarajan et al. [82] provide a heuristic approach to solve the problem for SDF, as the problem is NP-complete [76]. The approach taken by Stuijk et al. [29] is based on a DSE. Their approach finds all trade-offs (Pareto points) in the space of throughput-buffer size using a guided exploration rather than an exhaustive exploration. The exploration is narrowed down to the buffers that create the so-called storage dependencies while the SDF is executing with a certain throughput.

Buffer minimization techniques for CSDF graphs are presented in several works [77][83][84][29]. Denolf et al. [83] present an approach to find the minimum buffers required to execute a CSDF graph with a periodic schedule. However, there is no guarantee that another schedule that realizes the same throughput with smaller buffer requirements does not exist. A heuristic algorithm given by Wiggers et al. [77] minimizes buffer usage under a given throughput. Bodin et al. [84] provide a linear programming formalization of the minimum buffer requirements under a throughput constraint. The authors give an algorithm to solve the problem approximately.

Our work extends the exact throughput-buffering trade-off analysis for SDF and CSDF provided by Stuijk et al. [29] to SADF models. We use a guided exploration of the design space by identifying critical buffers, similar to the so-called storage dependencies identified from the state-space of the SDF/CSDF execution [29]. In this chapter, we provided a method of identifying critical buffers for SADF models. This method enables the application of the existing SDF and CSDF heuristic buffer sizing approaches such as the method given by Hendriks et al. [78], to SADF models.

It is worthwhile to mention that the trade-off analysis of Stuijk et al. [29] is not directly applicable to SADF models. Even though every scenario can be separately sized for its buffers using the SDF techniques, the results will be useful only in a corner case where the scenarios do not share any actors or buffers. For instance, if the I and P-frame decoding tasks in the MPEG-4 decoder are implemented as two separate applications. This is often not the case, because it increases the resource usage and reduces the maintainability of the application. Dynamic reading and writing patterns on buffers lead to different requirements compared to the strictly regular patterns occurring in SDF.

6.7 Conclusion

For real-time streaming applications on multi-processor systems, there is a trade-off between the amount of allocated storage space to the application and the throughput of the application. A complicated design problem is to find the minimal capacity requirements for buffers to execute an application under a given throughput constraint. To formalize the problem, a suitable model of computation is needed to mathematically describe the application. Nowadays, applications are dynamic, meaning that their behaviour changes at run-time. SADF is a suitable model for dynamic applications. In this chapter we provided a technique to find all optimal throughput-storage space trade-offs for applications using SADF models. The analysis performs a guided design space exploration that cuts-off the exploration space during the exploration without losing any optimal points and consequently, terminates in a reasonably short time. We showed that our method is practical by applying it to a number of realistic application models.

Chapter 7

Timing Bounds on the Worst-Case Execution

Minimizing resource usage is an important challenge when we want to run multiple applications on a single multi-processor system. In the previous chapter we provided a method to obtain the minimum storage space to run an application represented by an SADF model under a given throughput constraint. In this chapter we find the minimum time budgets on shared resources to run an application under the given throughput constraint. This requires an analysis method to estimate the throughput of the application, given the allocated processor budgets.

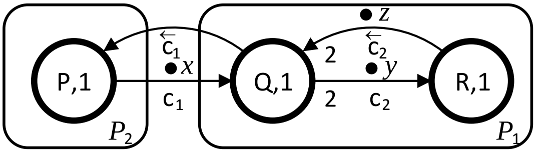
In this chapter we present an analysis method that provides tight and conservative timing bounds for SDF scenarios that are running on shared resources. We consider the resource sharing effects on the timing behaviour of the application by using an augmented symbolic simulation, where we embed worst-case resource availability curves in the symbolic simulation of the application model. Such simulation results in a state matrix that bounds the worst-case execution of the application on the shared resources. We obtain tighter timing bounds on the completion times of tasks than the state of the art analysis. This is achieved by improving the upper bounds on the response times of the tasks by identifying the busy periods of the resources. This enables us to use accumulated response times which are less pessimistic. Applying the proposed approach to models of realistic applications improves the timing bounds compared to the state of the art.

This chapter is organized as follows. Section 7.1 provides a brief introduction to the subject of this chapter. Section 7.2 gives the required background on the worst-case timing bounds obtained for application scenarios running on shared processors. Tighter timing bounds for such scenarios are obtained in Section 7.3. The results obtained from realistic case studies are reported and discussed in Section 7.4. The related work is discussed in Section 7.5. Section 7.6 concludes the chapter. This chapter is based on publication [43].

7.1 Introduction

Nowadays multiple real-time streaming applications are being realized on a single multi-processor system and share resources. One of the most important steps in designing embedded applications on shared systems is allocating enough resources to the applications to guarantee their real-time constraints. Often resource allocation strategies have an iterative process in which they initially allocate resources, they subsequently analyse the timing behaviour of the application and then adjust resource allocation parameters based on the analysis results [85]. The timing analysis is one of the core parts of such strategies and since it is a part of an iterative process, it should be fast enough to make the whole allocation process practical. Sharing resources introduces uncertainties (non-determinism) to the timing behaviour of the applications depending on the scheduling policy. For example when sharing a resource by Time Division Multiple Access (TDMA) arbitration, clock drifts cause uncertainties in the relative position of the allocated time slots which in turn causes uncertainties in the completion times of the tasks. To guarantee that the allocated resources make an application meet its constraints, we need to obtain conservative, but tight, timing bounds on the worst-case behaviour of the system (taking into account the uncertainties) in a reasonable time. We need the bounds to be tight to avoid over-allocation of resources.

In our setting, the application is realized on a multi-processor system of which the processors are being shared among several applications. Consider the SDF scenario shown in Figure 7.1. This scenario is running on a multiprocessor system with two processors, namely P_1 and P_2. As shown in the figure, in this system, actors Q and R are executed by P_1 and actor P is executed by P_2. We use a *static order schedule* to determine the order of actor executions on a processor. A static order schedule is a finite sequence of actors that shows the execution order of actor firings that are running on the same processor. The static order schedules for the example application are given in the table of Figure 7.1. The schedule for P_1 contains only one execution of actor P, and the schedule for P_2 executes the actor sequence RQR. We assume



Reps: $[\mathsf{P},\mathsf{Q},\mathsf{R}] \mapsto [1,1,2]$

ITs: $c_1(x), c_2(y), \overleftarrow{c_2}(z)$

FTs: $c_1(x), c_2(y), \overleftarrow{c_2}(z)$

Reward: 1

Sch.: $[\mathsf{RQR},\mathsf{P}] \mapsto [P_1, P_2]$

Figure 7.1: An example SDF scenario running on a two-processor system.

the communication delay between actors on the same processor is negligible. The communication delay between the actors executing on different processors can be modelled. For example, a point-to-point connection with a fixed communication delay can be modelled by adding an actor with suitable execution time between the communicating actors. More complex communication models such as a network-on-chip connection model given by Moonen et al. [86] can also be considered. In the example, for simplicity, we do not consider communication delays between actors executing on different processors, as they complicate the scenario graph.

We assume the processors are shared by budget schedulers to have an independent bound for each application [87]. A *budget scheduler* guarantees the application a minimum amount of budget (processing time) over a periodic time frame called the *replenishment interval.* Note that such a scheduler requires a system that supports the preemption of task executions. The challenge is that in this setting the exact completion times of actor firings cannot be determined because the precise state of the scheduler is not known when an actor is able to start its execution. For example, an actor might be enabled at a time instant where the whole budget allocated to the application has already been used for the current replenishment interval, and the task has to wait for the next replenishment interval (worst-case), or the actor might immediately start executing when it is enabled at the start of the allocated budget (best-case). Although it is possible to obtain conservative timing bounds using the worst-case completion times for all task executions separately, the obtained bounds are too pessimistic as shown by Siyoum et al. [88].

In this chapter, we adapt the symbolic simulation method introduced in Section 2.7.1 to extract the worst-case timing bounds, in the form of state matrices, for scenarios running on shared multi-processor systems. To obtain tighter bounds, we exploit two techniques. First, we use the aggregation of actor firings. When the firing of an actor a_1 enables a firing of actor a_2 on the same processor, the worst-case completion time assumption can be avoided for the firing of a_2. In the worst-case, the completion time of a_2 is equal to the completion time of an aggregate actor that is enabled at the same time as a_1 and has an execution time equal to the sum of the execution times of a_1 and a_2. This techniques is already applied by Siyoum et al. [88] for dataflow and generally for real-time response time analysis.

The second technique is a contribution of this chapter. We compute the completion time of an actor firing, separately per dependency of the firing. In this way, per dependency we can decide whether we can use the first technique or we are forced to use the worst-case assumption. For instance, in Figure 7.1 we compute the completion time of actor Q considering the following dependencies, separately: the availability of tokens x and z, and the completion of the firing of R. We show that with the availability of x and y we are forced to use the worst-case assumption to be conservative, however, with the completion of the firing of actor R, we can use the first technique. In the earlier work [88], such a decision is made once, considering worst-case among all dependencies. For instance, for the firing of Q, it considers the worst-case assumption for all dependencies. We show that our technique improves the upper bounds on the completion times of actor firings, and consequently, results in a tighter timing bound.

7.2 Worst-Case Timing Bounds

7.2.1 Worst-Case Response Times

As mentioned in the previous section, the order in which the actors of an application get access to a processor is determined by a static order schedule. Therefore, the order of actor firings in applications can be tracked on processors. Assume a processor executes an actor a_2 after executing an actor a_1. Actor a_2 can start firing if the processor has completed the firing of actor a_1 and the tokens required by the firing of actor a_2 are available. Hence, the *Earliest Start Time (EST)* of the firing of actor a_2 is equal to the latest of the following events: the processor completes the firing of actor a_1 and all required tokens for the firing of a_2 are available. Note that the firing of a_2 may not be able to start at its EST, because the scheduler may be processing other applications at that particular time instant. For instance, when the firing of a_1 has used all the budget allocated to the application in the current replenishment interval, a_2 has to wait for the allocated budget in the next replenishment interval, before it can actually start.

Let $\bar{a}_P(k)$ denote the k^{th} actor that has access to P according to a given static order schedule $\bar{a}_P$. Let tuple (P, k) denote firing k on processor P which corresponds to actor $a = \bar{a}_P(k)$. From the time (P, k) starts, it requires $e(a)$ units of processing time to be completed where $e(a)$ is the execution time of actor a. Since preemption may occur in the system, the actor completes its firing when the sum of the processing time that it gets after its EST, exceeds $e(a)$. The total time it takes for an actor to complete its firing after its EST, is called the *response time* of the firing. Let $s(P, k)$ denote the EST of firing (P, k), and $\omega(a)$ denote the *Worst-Case Response Time (WCRT)* of actor a. We associate a WCRT with an actor (instead of a firing), because later we see that it only depends on the execution time of the actor. An upper bound $c(P, k)$ on the completion time of firing (P, k) is calculated as follows.

$$c(P, k) = s(P, k) + \omega(\bar{a}_P(k)) \tag{7.1}$$

To calculate $\omega(\bar{a}_P(k))$, we have to find the minimum amount of time that guarantees the application, $e(\bar{a}_P(k))$ units of processing time on processor P. Consider processor P_1 in Figure 7.1, shared by the TDMA shown in Figure 7.2. In this TDMA, the green slots are allocated to the application shown in Figure 7.1. Now consider the first firing on processor P_1, i.e., $(P_1, 1)$ which is of actor R, with $e(\mathsf{R}) = 1$. The worst-case alignment of EST with TDMA schedule, with respect to the positioning of the allocated slots is depicted in the same figure. In this situation, the actor has to wait 4 time units to start processing at the next allocated slot; hence, $\omega(\mathsf{R}) = 5$ and in the worst-case it will completely be processed at $c(P_1, 1) = s(P_1, 1) + 5$. This means that 5 time units is the smallest amount of time that guarantees the application, 1 unit of processing time using the TDMA in Figure 7.2, assuming the TDMA schedule is not synchronized with the execution of the application.

To compute the WCRTs of actors, we use the notion of *WCRC* [88]. A WCRC is a function $\zeta(\delta)$ that specifies the minimum amount of processing time allocated to the application in any time interval of length δ [88]. For a TDMA scheduler for example, the WCRC has a periodic behaviour of length w, where w is the size of the time wheel, i.e., $\zeta(\delta + w) = \zeta(\delta) + \Delta$ for some constant Δ. The WCRC of the TDMA arbitration in Figure 7.2 is shown in the same figure. This figure allows us to determine the WCRC of a firing as the earliest time that the processor dedicates the required processing time to the actor. The WCRT of a firing actor can be obtained from the WCRC of the processor as follows.

$$\omega(a) = \inf\{\delta \in \mathbb{R}^+ \mid \zeta(\delta) \geq e(a)\} \tag{7.2}$$

For instance, actors with execution times of 1, 2 and 3 time units have WCRTs

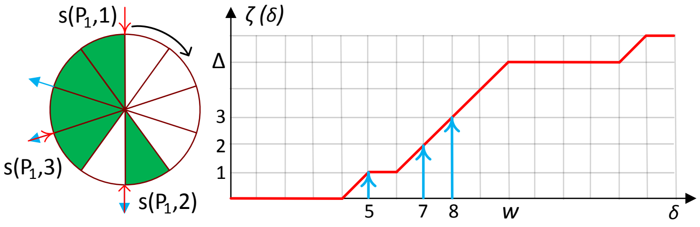


Figure 7.2: A TDMA and its corresponding Worst-Case Resource Curve (WCRC) used for processor P_1. Only the green-coloured slots are allocated to the application.

of 5, 7 and 8 time units, respectively, as illustrated by the blue arrows in the figure.

7.2.2 The State Matrix With Worst-Case Response Times

In Section 2.7.1 we explained the computation of the state matrix for the self-timed execution of a scenario. In a self-timed execution, actors fire as soon as they are enabled. This section illustrates the computation of the state matrix for a scenario that is running on shared processors. We use a symbolic simulation to obtain the state matrix as we did for the self-timed execution. The firing semantics in this case are different from the semantics in the self-timed execution in three ways. First, actor firings that are executed on the same processor are ordered by static order schedules. Second, they start only if the processor that executes their corresponding actors is available. Third, the WCRTs are used instead of the actor execution times.

To illustrate, consider the scenario shown in Figure 7.1. Let's assume P_1 is shared by the TDMA allocation shown in Figure 7.2, and P_2 is shared by another TDMA shown in Figure 7.3. Using the WCRCs of theses TDMAs, we obtain $\omega(\mathsf{R}) = \omega(\mathsf{Q}) = 5$ and $\omega(\mathsf{P}) = 2$. Let t_x, t_y, t_z denote the time-stamps of the initial tokens labelled x, y, z in Figure 7.1, respectively. Further let t_{P_1} and t_{P_2} denote the initial availability times of processors P_1 and P_2, respectively. In the following, we compute upper bounds for the symbolic completion times (see Section 3.3) of the firings involved in the scenario.

To apply the availability of the processors in the symbolic execution, we consider the worst-case availability times of the processors as new elements

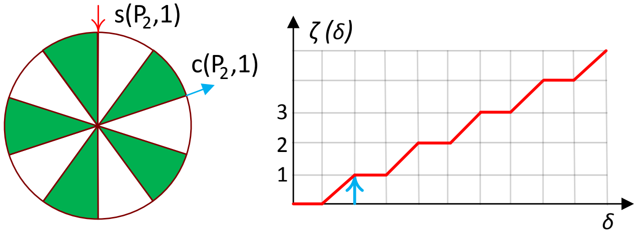


Figure 7.3: The TDMA scheduler used on Processor P_2 and its corresponding WCRC

in the state vector. We consider the state vector $[\ t_x \quad t_y \quad t_z \quad t_{P_1} \quad t_{P_2}\ ]^T$. Actor P is the first actor in the static schedule of P_2. This actor is not enabled initially because it does not have enough tokens on channel $\overleftarrow{c}_1$. The first firing on P_1, i.e., $(P_1, 1)$, is a firing of R. This firing is dependent on the availability of token y and processor P_1. Therefore, the symbolic time-stamp vector of its EST is computed as

$$\boldsymbol{s}(P_1, 1) = \begin{bmatrix} -\infty \\ 0 \\ -\infty \\ -\infty \\ -\infty \end{bmatrix}^T \oplus \begin{bmatrix} -\infty \\ -\infty \\ -\infty \\ 0 \\ -\infty \end{bmatrix}^T = \begin{bmatrix} -\infty \\ 0 \\ -\infty \\ 0 \\ -\infty \end{bmatrix}^T .$$

By adding the WCRT of actor R to $\boldsymbol{s}(P_1, 1)$, we obtain an upper bound on the symbolic completion time of the first firing on processor P_1 as follows.

$$\boldsymbol{c}(P_1, 1) = \boldsymbol{s}(P_1, 1) + \omega(\mathsf{R}) = \begin{bmatrix} -\infty \\ 0 \\ -\infty \\ 0 \\ -\infty \end{bmatrix}^T + 5 = \begin{bmatrix} 5 \\ -\infty \\ -\infty \\ 5 \\ -\infty \end{bmatrix}^T .$$

By the completion of this firing, a token is produced on channel $\overleftarrow{c}_2$ and P_1 becomes available again. The availability of P_1, the token produced on $\overleftarrow{c}_2$ and tokens x and z are required for the second firing on P_1 which is of Q.

Therefore, the symbolic EST of this firings is computed as follows.

$$\boldsymbol{s}(P_1,2) = \begin{bmatrix} -\infty \\ 5 \\ -\infty \\ 5 \\ -\infty \end{bmatrix}^T \oplus \begin{bmatrix} -\infty \\ 5 \\ -\infty \\ 5 \\ -\infty \end{bmatrix}^T \oplus \begin{bmatrix} 0 \\ -\infty \\ -\infty \\ -\infty \\ -\infty \end{bmatrix}^T \oplus \begin{bmatrix} -\infty \\ -\infty \\ 0 \\ -\infty \\ -\infty \end{bmatrix}^T = \begin{bmatrix} 0 \\ 5 \\ 0 \\ 5 \\ -\infty \end{bmatrix}^T$$

Consequently, we can compute the following upper bound for the symbolic completion time of the second firing on P_1 as follows.

$$\boldsymbol{c}(P_1,2) = \boldsymbol{s}(P_1,2) + \omega(\mathsf{Q}) = \begin{bmatrix} 0 \\ 5 \\ 0 \\ 5 \\ -\infty \end{bmatrix}^T + 5 = \begin{bmatrix} 5 \\ 10 \\ 5 \\ 10 \\ -\infty \end{bmatrix}^T$$

The completion of this firing produces a token on channel $\overleftarrow{c_1}$ and makes P_1 available again. Now, P is enabled, and R is enabled for the second time. Firing $(P_2,1)$ is of P and requires the availability of P_2 and the token produced on channel $\overleftarrow{c_1}$ by Q. Thus, it completes at

$$\boldsymbol{c}(P_2,1) = \boldsymbol{s}(P_2,1) + \omega(\mathsf{P}) = \left(\begin{bmatrix} 5 \\ 10 \\ 5 \\ 10 \\ -\infty \end{bmatrix}^T \oplus \begin{bmatrix} -\infty \\ -\infty \\ -\infty \\ -\infty \\ 0 \end{bmatrix}^T \right) + 2 = \begin{bmatrix} 7 \\ 12 \\ 7 \\ 12 \\ 2 \end{bmatrix}^T .$$

Finally, an upper bound on the symbolic completion time of the third firing on P_1 is computed as follows.

$$\boldsymbol{c}(P_1,3) = \boldsymbol{s}(P_1,3) + \omega(\mathsf{R}) = \left(\begin{bmatrix} 5 \\ 10 \\ 5 \\ 10 \\ -\infty \end{bmatrix}^T \oplus \begin{bmatrix} 5 \\ 10 \\ 5 \\ 10 \\ -\infty \end{bmatrix}^T \right) + 5 = \begin{bmatrix} 10 \\ 15 \\ 10 \\ 15 \\ -\infty \end{bmatrix}^T$$

After symbolically simulating all actor firings in the scenario, we can compute the state matrix by collecting the symbolic time-stamp vectors of the final tokens and the latest symbolic availability times of the processors in a matrix as follows.

$$\begin{bmatrix} t'_x \\ t'_y \\ t'_z \\ t'_{P_1} \\ t'_{P_2} \end{bmatrix} = \begin{bmatrix} 7 & 12 & 7 & 12 & 2 \\ 5 & 10 & 5 & 10 & -\infty \\ 10 & 15 & 10 & 15 & -\infty \\ 10 & 15 & 10 & 15 & -\infty \\ 7 & 12 & 7 & 12 & 2 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \\ t_{P_1} \\ t_{P_2} \end{bmatrix}$$

Using the state matrix above, the lower bound on the throughput of an application that repeatedly executes the scenario shown in Figure 7.1, can be computed as 1/15. In the remainder of this section we present an existing technique [88] to obtain an even tighter, i.e., less conservative lower bound compared to the bound computed above.

7.2.3 Accumulated Worst-Case Response Times

As explained, using WCRCs to find the response times of the firings assumes the longest waiting times for actors, i.e., it always starts from the beginning of the WCRC, where the processor is not allocated to the application. It is well known that this assumption can be avoided for time intervals in which we can ensure the processor will be kept busy within those time intervals. Such time intervals are called the *busy periods* of the processors [89]. Within a busy period, the first firing can be assumed to start from the beginning of the curve and the subsequent firings that follow, continue on the same curve because they can start as soon as the processor completes the previous firing. One way to be sure that the processor is kept busy is when a sequence of *consecutive* firings are started. A sequence of firings is said to be consecutive if we know that for each of its firings, the EST is no later than the completion of the previous firing.

Definition 7.1. *A sequence* $\bar{f} = (P, k), \cdots, (P, k+n)$ *of firings is said to be consecutive if the earliest start time of firing* (P, i) *is guaranteed to be at most the completion time of firing* $(P, i-1)$ *for* $k + 1 \leq i \leq k + n$*.*

Consider again the three firings on processor P_1. Observe that the location of the initial tokens are such that as soon as the first firing of R completes, Q is able to start, and as soon as Q completes, R becomes able to start its second firing, i.e., $s(P_1, i) = c(P_1, i-1)$ for $i = 2, 3$. Considering the WCRC shown in Figure 7.2, the first firing starts from the beginning of the curve. The second firing can start at the same time as the first firing completes. Therefore, it starts from the point on the curve where $(P_1, 1)$ completed, and hence it holds that $c(P_1, 2) = s(P_1, 1) + 7$. Similarly, the third firing also continues the same curve and $c(P_1, 3) = s(P_1, 1) + 8$ as shown in Figure 7.2. Note that if we fail to exploit the fact that these firings are consecutive, we obtain the pessimistic upper bound of 15 time units, given that Eq. 7.2 returns the WCRT of 5 time units for all three actor firings and the sum of the WCRTs equal to 15 time units.

From the example, one can realize that the completion time of firings within a sequence of consecutive firings are computed from the EST of the first firing and the WCRT for a task with the same execution time as the sum of the execution times of the individual actor firings. We adapt Eq. 7.1 and Eq. 7.2 to obtain tight upper bounds for the completion times of firings in busy periods. Consider a sequence $\bar{f} = (P, k), \cdots, (P, k + n)$ of consecutive firings. Let $\bar{a}_P(k, k + i)$ denote the sequence of actors corresponding to the sequence $(P, k), \cdots, (P, k + i)$ of firings for $0 \leq i \leq n$. An upper bound on the completion time of firing $(P, k + i)$, can be calculated by adding the *accumulated worst-case response time*, $\omega(\bar{a}_P(k, k + i))$, to the EST, $s(P, k)$ of the first firing in the sequence, as follows.

$$c(P, k + i) = s(P, k) + \omega(\bar{a}_P(k, k + i)) \tag{7.3}$$

The accumulated worst-case response time $\omega(\bar{a}_P(k, k + i))$, is equal to the WCRT of an execution time equal to the sum of execution times of the corresponding actors of the firings in $\bar{a}_P(k, k + i)$. $\omega(\bar{a}_P(k, k + i))$ is defined as follows.

$$\omega(\bar{a}_P(k, k + i)) = \inf\{\delta \in \mathbb{R}^+ \mid \zeta(\delta) \geq \sum_{l=k}^{k+i} e(\bar{a}_P(l))\} \tag{7.4}$$

Using the above equation on the consecutive sequence RQR, we obtain $\omega(\mathsf{R}) = 5$, $\omega(\mathsf{RQ}) = 7$ and $\omega(\mathsf{RQR}) = 8$. In the following we show how accumulated worst-case response times can be used in the symbolic simulation to obtain a less conservative state matrix for the example scenario, compared to the one obtained in Section 7.2.2.

7.2.4 Symbolic Simulation With Accumulated Worst-Case Response Times

Siyoum et al. [88] developed an approach to provide tighter upper bounds on the symbolic completion times of firings. This approach sets up a condition to identify consecutive firings on the processors symbolically to avoid using WCRTs for every individual firing. To calculate the symbolic completion time upper bound of a firing, the authors initially assume that it will start a new sequence of consecutive firings on the processor. If the symbolic EST of the next firing is not greater than the symbolic completion time upper bound of the firing, i.e., $\boldsymbol{s}(P, k+1) \leq \boldsymbol{c}(P, k)$, then the next firing is appended to the sequence of consecutive firings. Then, the accumulated worst-case response time can be used to calculate its completion time upper bound based on Eq. 7.3. Otherwise, it is assumed to start a new sequence of consecutive firings and therefore, its completion time is calculated by Eq. 7.1.

In Figure 7.1, the first firing of actor R starts a sequence of consecutive firings on P_1. According to the symbolic simulation performed in Section 7.2.2, $\boldsymbol{s}(P_1, 1) = [\ -\infty \quad 0 \quad -\infty \quad 0 \quad -\infty\ ]$, and $\boldsymbol{c}(P_1, 1) = [\ -\infty \quad 5 \quad -\infty \quad 5 \quad -\infty\ ]$. Moreover, $\boldsymbol{s}(P_1, 2)$ is computed as $[\ 0 \quad 5 \quad 0 \quad 5 \quad -\infty\ ]$. Note that $\boldsymbol{s}(P_1, 2)$ has additional dependencies on initial tokens. Hence, it cannot be guaranteed to be consecutive. This is recognized by the condition that does not hold in this case, i.e.,

$$\boldsymbol{s}(P_1, 2) = [\ 0 \quad 5 \quad 0 \quad 5 \quad -\infty\ ] \not\leq \boldsymbol{c}(P_1, 1) = [\ -\infty \quad 5 \quad -\infty \quad 5 \quad -\infty\ ].$$

As the condition is not satisfied, the second firing on P_1 starts a new sequence of consecutive firings and completes at $\boldsymbol{c}(P_1, 2) = \boldsymbol{s}(P_1, 2) + \omega(\mathsf{Q}) = [\ 0 \quad 5 \quad 0 \quad 5 \quad -\infty\ ] + 5 = [\ 5 \quad 10 \quad 5 \quad 10 \quad -\infty\ ]$. The third firing on P_1 starts at $\boldsymbol{s}(P_1, 3) = [\ 5 \quad 10 \quad 5 \quad 10 \quad -\infty\ ]$. Since $\boldsymbol{s}(P_1, 3) \leq \boldsymbol{c}(P_1, 2)$, the firings are consecutive and the completion of the third firing is calculated using Eq. 7.3. Therefore, for this firing, $\boldsymbol{c}(P_1, 3) = \boldsymbol{s}(P_1, 2) + \omega(\mathsf{QR}) = [\ 0 \quad 5 \quad 0 \quad 5 \quad -\infty\ ] + 7 = [\ 7 \quad 12 \quad 7 \quad 12 \quad -\infty\ ]$, which is a tighter upper bound compare to the bound computed in Section 7.2.2. With the symbolic completion times as computed above we obtain the following state matrix.

$$\begin{bmatrix} t'_x \\ t'_y \\ t'_z \\ t'_{P_1} \\ t'_{P_2} \end{bmatrix} = \begin{bmatrix} 7 & 12 & 7 & 12 & 2 \\ 5 & 10 & 5 & 10 & -\infty \\ 7 & 12 & 7 & 12 & -\infty \\ 7 & 12 & 7 & 12 & -\infty \\ 7 & 12 & 7 & 12 & 2 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \\ t_{P_1} \\ t_{P_2} \end{bmatrix}$$

Using the above state matrix, we can compute the throughput lower bound of 1/12 for an application that repeatedly executes the scenario shown in Figure 7.1. Observe that the throughput lower bound computed from the worst-case accumulated response times is less pessimistic than the throughput lower bound of 1/15 computed earlier using worst-case response times. In the next section we propose a new technique to obtain an even tighter lower bound compared to the bounds obtained in this section, using the existing techniques.

7.3 Tighter Worst-Case Timing Bounds

In this section we improve the symbolic simulation introduced in the previous section to obtain state matrices that are still conservative but even less pessimistic. Our symbolic simulation is based on a new method for symbolic identification of consecutive firings. We show that the condition $\boldsymbol{s}(P, k+1) \le \boldsymbol{c}(P, k)$ is too restrictive. It requires *all* elements of $\boldsymbol{s}(P, k+1)$ not to be greater than $\boldsymbol{c}(P, k)$. When this condition is not satisfied, the WCRT is used according to Eq. 7.1. We show that this often creates overly pessimistic dependency relations between the availability times of the tokens/processors and the completion times of the firings. For example, recall that from the results of Section 7.2.4,

$$\boldsymbol{s}(P_1, 2) = \begin{bmatrix} 0 \\ 5 \\ 0 \\ 5 \\ -\infty \end{bmatrix}^T \not\le \boldsymbol{c}(P_1, 1) = \begin{bmatrix} -\infty \\ 5 \\ -\infty \\ 5 \\ -\infty \end{bmatrix}^T .$$

Note that the entries with the value of 0 in $\boldsymbol{s}(P_1, 2)$ correspond to the initial tokens x and z, which are consumed by firing $(P_1, 2)$. This means that at least every time an initial token is consumed by actors, this condition is not met. Therefore, for such cases we always have to use Eq. 7.1. In this example, it leads to the upper bound

$$\boldsymbol{c}(P_1, 2) = \begin{bmatrix} 0 \\ 5 \\ 0 \\ 5 \\ -\infty \end{bmatrix} + 5 = \begin{bmatrix} 5 \\ 10 \\ 5 \\ 10 \\ -\infty \end{bmatrix} .$$

To understand why this upper bound is pessimistic, we need to take a look at the meaning behind every individual element of the symbolic time-stamp $t = \boldsymbol{g}^T \boldsymbol{t} = \max_i(t_i + g_i)$. According to the definition, if of all limiting factors of t, t_j is the actual limiting factor, i.e., $j = \text{argmax}(t_j + g_j)$, g_j is the *maximum* time difference between t and t_j. Therefore, the maximum time difference between $c(P_1, 2)$, computed above, and t_y, is equal to the second element of $\boldsymbol{c}(P_1, 2)$, i.e., 10, if we know that the availability of y is limiting the completion of $(P_1, 2)$. However, this is too pessimistic, because in case y is indeed the limiting factor, then $(P_1, 2)$ will immediately start when $(P_1, 1)$ completes; we can conclude that in this situation, $(P_1, 1)$ and $(P_1, 2)$ are consecutive.

Therefore, the maximum time difference between t_y and $(P_1, 2)$ cannot be more than the accumulated response time of $\omega(\mathsf{RQ}) = 7$. However, if t_z or t_x is the limiting factor, then the maximum time difference between t_x and $(P_1, 2)$ or t_z and $(P_1, 2)$ can be at most the WCRT of $\omega(\mathsf{Q}) = 5$. This means that the symbolic completion time of $(P_1, 2)$ is bounded by $[\begin{array}{ccccc} 5 & 7 & 5 & 7 & -\infty \end{array}]$. To avoid pessimistic upper bounds on the symbolic completion times, we propose to compute them with respect to each limiting factor t_i, separately.

To find the maximum time differences between the completion times of firings and all t_i separately, we have to find the firing dependencies for all the firings in the scenario. This enables us to separately capture all possible dependency paths that connect the completion times of firings to all t_i. Then, for each dependency path we can separately determine which firings are consecutive in it. To do so, we first find all firings on which the start of a firing directly depends, namely the firings that produce a token consumed by the firing as well as the firing that precedes it in the static order schedule on the processor.

Definition 7.2. *The dependency set $D_{(P,k)}$ comprises the firings the completions of which are required for (P, k) to start, i.e., $D_{(P,k)} = \{(P', k') \mid (P, k)$ consumes a token produced by $(P', k')\} \cup \{(P, k - 1)\}$.*

Note that the first firing on the processor, i.e., $(P, 1)$ is dependent on the initial availability of P which we denote by $(P, 0)$, as if it would become available after the completion of firing 0 on P. In addition, to be able to represent the dependency on the availability of initial tokens, we represent them as if they were produced by completion of firings with negative indices on an arbitrary processor, i.e., (P, k) with $k < 0$. The firings with non positive indices are referred to as *initial dependencies*.

During symbolic simulation, for each token that is produced by firings, in addition to the symbolic time-stamp that shows its production time, we add extra information regarding the firing that produced them. By keeping track of the tokens produced and consumed by firings, we can extract the dependency sets of actor firings. Figure 7.4 shows the dependency graph associated with the execution of the scenario shown in Figure 7.1. The dependency graph is the graphical representation of the dependency sets of all actor firings in the scenario. In this graph, the nodes represent the firings. A directed edge from (P', k') to (P, k) indicates that (P, k) is dependent on (P', k'). The black edges indicate the firing dependencies on the same processor and the red edges indicate the dependencies on different processors. For the sake of readability, the initial dependencies are given their own names instead of firings with non positive indices.

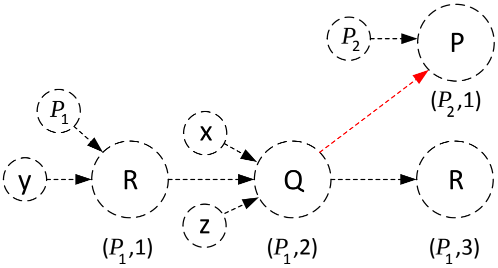


Figure 7.4: The dependency graph of the scenario shown in Figure 7.1.

Using the dependency graph we can track the dependencies of each firing back to the initial dependencies and separately compute the time difference between them. The key point is that if two nodes are connected only by black edges, then the time difference between them is equal to the accumulated worst-case response times of all firings between them including the last node. For example, the time difference between $(P_1, 3)$ and t_y (node y in the figure) is equal to $\omega(\mathsf{RQR}) = 8$ (which is less pessimistic compared to 12 obtained by the method of Siyoum et al. [88]). If the connecting path contains red edges, we have to separate the path on the red edges, as a consecutive firing can be guaranteed only on a single processor (we assume TDMA arbitrations in processors are not guaranteed to be synchronized with each other). For each separate piece of the path, we compute the accumulated response times. Finally, the response time is equal to the sum of all computed accumulated response times. For example, the time difference between $(P_2, 1)$ and t_y is equal to $\omega(\mathsf{RQ}) + \omega(\mathsf{P}) = 7 + 2 = 9$. If there is more than one path between two nodes (which does not occur in our example), then the time difference is equal to the maximum of time differences in all paths. As an example, using our method, $\boldsymbol{c}(P_2, 1)$ is computed as follows.

$$\boldsymbol{c}(P_2,1) = \begin{bmatrix} \omega(\mathsf{Q}) \\ \omega(\mathsf{RQ}) \\ \omega(\mathsf{Q}) \\ \omega(\mathsf{RQ}) \\ -\infty \end{bmatrix}^T + \omega(\mathsf{P}) = \begin{bmatrix} 5 \\ 7 \\ 5 \\ 7 \\ -\infty \end{bmatrix}^T + 2 = \begin{bmatrix} 7 \\ 9 \\ 7 \\ 9 \\ 2 \end{bmatrix}^T$$

Using the proposed method the state matrix of the scenario in Figure 7.1

ALGORITHM 5: Compute the state matrix of an SDF scenario that is running on a multi-processor system

```
   Input: A scenario $s$ running on a set $\Pi$ of processors, and WCRCs $\zeta_P$ and
          static order schedules $\bar{a}_P$ for every processor $P$ in $\Pi$
   Output: The state matrix $\boldsymbol{G}_s$
1  $D \leftarrow \emptyset$;
2  $\Phi \leftarrow$ initialDependencies($s$);
3  $r' \leftarrow r(s)$;
4  repeat
5      $a \leftarrow$ getNextActor($s, \bar{a}_P$);
6      $P \leftarrow$ getProcessor($s, a$);
7      $k \leftarrow$ incrementFiringCounter($P$);
8      $D_{(P,k)} \leftarrow$ computeDependencySet($P, k$);
9      $D \leftarrow D \cup \{D_{(P,k)}\}$;
10     $\boldsymbol{c}(P,k) =$ computeSymbolicCompletionTime($(P,k), D, \Phi, a, \zeta_P, s$);
11     $\Phi \leftarrow \Phi \cup \{\boldsymbol{c}(P,k)\}$;
12     $r'(a) \leftarrow r'(a) - 1$;
13 until $r'$ has no positive elements;
14 $\boldsymbol{G}_s \leftarrow$ collect the time-stamp vectors of final tokens and the latest processor
     availability times from $\Phi$;
```

is obtained as follows.

$$\begin{bmatrix} t'_x \\ t'_y \\ t'_z \\ t'_{P_1} \\ t'_{P_2} \end{bmatrix} = \begin{bmatrix} 7 & 9 & 7 & 9 & 2 \\ 5 & 7 & 5 & 7 & -\infty \\ 7 & 8 & 7 & 8 & -\infty \\ 7 & 8 & 7 & 8 & -\infty \\ 7 & 9 & 7 & 9 & 2 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \\ t_{P_1} \\ t_{P_2} \end{bmatrix}$$

Using this state matrix a lower bound on the throughput of the application is computed as 1/8, which is a tighter bound compared to the bound 1/12 obtained in Section 7.2.4.

Algorithm 5 sketches the construction of the state matrix based on separate analysis per dependency for a given scenario s which is running on a set Π of processors, where every processor $P \in \Pi$ has a WCRC ζ_P. The algorithm stores the dependency sets and completion times of all firings in sets D and Φ, respectively. D and Φ are computed following a symbolic simulation of the scenario. D is initially empty and Φ is initialized with the symbolic completion times of initial dependencies (Lines 1,2). The repetition vector is copied to r' in Line 3. The algorithm picks actor a, such that a is an enabled actor

ALGORITHM 6: Compute an upper bound on symbolic completion times of firings

```
procedure computeSymbolicCompletionTime($(P, k), D, \Phi, \bar{a}, \zeta_P, \bar{a}_P, s$);

$\boldsymbol{c}(P, k) \leftarrow [\ -\infty \quad \cdots \quad -\infty\ ]$;
repeat
    $(P', k') \leftarrow$ pick a dependency from $D_{(P,k)}$;
    if $P' \neq P \vee k' \leq 0$ then
        $\boldsymbol{c}(P', k') \leftarrow$ getCompletionTime($(P', k'), \Phi$);
        $\tilde{\boldsymbol{c}}(P, k) \leftarrow \boldsymbol{c}(P', k') +$ getWorstCaseAccumulatedRT($\bar{a}, \zeta_P, s$);
    else
        $\bar{a} \leftarrow$ prepend $\bar{a}_P(k')$ to $\bar{a}$;
        $\tilde{\boldsymbol{c}}(P, k) \leftarrow$
          computeSymbolicCompletionTime($(P', k'), D, \Phi, \bar{a}, \zeta_P, \bar{a}_P, s$);
    end
    $\boldsymbol{c}(P, k) = \tilde{\boldsymbol{c}}(P, k) \oplus \boldsymbol{c}(P, k)$;
    $D_{(P,k)} \leftarrow D_{(P,k)} \setminus (P', k')$;
until $D_{(P,k)}$ is empty;
return $\boldsymbol{c}(P, k)$;
```

in the next position of the static order schedule of a processor (Line 5). The processor which executes actor a is obtained from the scenario in Line 6. The firing counter on this processor is incremented (Line 7). The dependency set of the firing is constructed in Line 8 and added to D in Line 9. Line 10 calls Algorithm 6 to compute the upper bound on the symbolic completion time of (P, k), which is a firing of a. This algorithm computes the upper bounds by constructing the dependency graph as follows. First, node (P, k) is created. Then the dependency nodes of (P, k) are added to the graph by connecting the nodes representing the firings in the dependency set $D_{(P,k)}$ to this node. Then, the dependency nodes of the nodes in $D_{(P,k)}$ are added to the graph in a similar way and so on. This process continues until all source nodes of the graph (the ones without input edges) are either representing initial dependencies or firings on other processors. Then, the symbolic completion time of the firing is obtained by adding the maximum accumulated response times among all paths that connect the source nodes to (P, k), to the symbolic completion times of firings represented by source nodes. The calculated symbolic completion time of the firing is added to the set of symbolic completion times Φ (Line 11). The repetition count of actor a in r' is reduced by one in Line 12. Lines 5-12 are repeated until vector r' has no positive elements, i.e. all actor firings in the scenario are completed. In Line 14, the algorithm constructs the state matrix by obtaining the time-stamp vectors of final tokens and the latest processor availability times from the completion times in Φ.

Algorithm 6 computes the upper bound on the symbolic completion time of firing (P, k) in a recursive fashion from the completion times of its dependencies. The inputs of this algorithm are: firing (P, k), set D of the stored dependency sets, set Φ of all symbolic completion times of stored firings, sequence $\bar{a}$ of actors that corresponds to the consecutive sequence of firings which will be identified recursively by the algorithm (the first invocation always takes the sequence that includes only the actor of firing (P, k)), the WCRCs of the processors, the static order schedules $\bar{a}_P$ and scenario s. Initially, the algorithm assumes that $\boldsymbol{c}(P, k)$ completes at $-\infty$, as it has not considered any dependencies yet. Therefore, $\boldsymbol{c}(P, k)$ is set to a vector of an appropriate size with all of its elements equal to $-\infty$ (Line 2). The algorithm picks a dependency from $D_{(P,k)}$ (Line 4). A symbolic completion time for (P, k) is computed from this dependency as follows: if the dependency is either initial, i.e., (P', k') s.t. $k' \leq 0$ or from a different processor, i.e., (P', k') s.t. $P' \neq P$, the symbolic completion time from this dependency is calculated by adding $\omega(\bar{a})$ to the symbolic completion time of that dependency (Eq. 7.4) (Lines 6-7); if the dependency is such that $P' = P$, then $\bar{a}_{P'}(k')$ is identified as a dependency of $\bar{a}_P(k)$ in the same processor, therefore, considering only this dependency, $\bar{a}_{P'}(k')\bar{a}_P(k)$ is a consecutive sequence and $\bar{a}_{P'}(k')$ is prepended to $\bar{a}$ (Line 9). Then, Algorithm 6 is invoked recursively for (P', k') and new $\bar{a}$ to obtain the symbolic completion time from this dependency (Line 10). The symbolic completion time of (P, k) is the maximum symbolic completion time computed from all dependencies (Line 12). The algorithm terminates when the completion times from all dependencies in all invocations are computed.

7.4 Evaluation

We have implemented our timing analysis method in the SDF3 tool. We compared throughput lower bounds obtained by our approach with the state of the art analysis given by Siyoum et al. [88] for three realisitic applications: H.263 encoder, H.263 decoder and sample rate converter, all available in the SDF3 tool. For each application, we used SDF3 to realize it on a multi-processor system with four processors such that the total work load is evenly distributed between processors as much as possible. We set the replenishment intervals

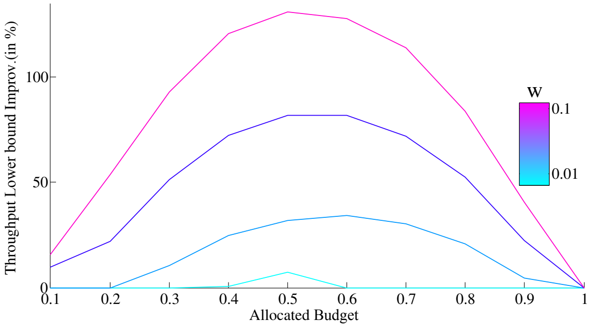


Figure 7.5: Average improvements in throughput lower bounds compared to the results given by Siyoum et al. [88]

to $0.01 \times C_f \leq w \leq 0.1 \times C_f$ where C_f is the cycle time of the application when all processors are fully allocated to the application. Large replenishment intervals cause huge delays in execution of the application, which is not desired; small replenishment intervals are less useful because of the context switch overhead. Figure 7.5 shows the average relative improvements in the throughput lower bounds of applications for different replenishment intervals and allocated budgets. For each combination of replenishment interval and allocated budget, state matrix is constructed by Algorithm 5. Then a lower bound on throughput is computed by obtaining the $(\max, +)$ eigenvalue of this matrix. As shown in the figure, the improvement ratio decreases when the application gets smaller or larger processor shares. In these cases using the accumulated worst-case response times does not have much improvements over WCRTs. When the application share is too small, the response times of tasks are very long, therefore the response time improvements which are in the order of the time wheel size is relatively small. When the application share is too big, then the penalty of assuming WCRTs is small, and therefore the improvements are also small. The average analysis run-time for the mentioned applications on a core i7, windows machine is 320 milliseconds which is 17% longer compared to that state of the art [88]. The longer run-time is caused by the calculation of the symbolic completion times per dependency separately, rather than computing the worst-case once for all dependencies.

7.5 Related Work

Analysing the timing behaviour of streaming applications is the subject of many works. The analysis methods in different works depend on the model considered for the application. As we are interested in data-driven execution, we consider tasks models with data dependencies. We classify these models into two different groups: single-rate and multi-rate. In the single-rate models all tasks have the same input and output rate which means that they all produce and consume the same amount of data. However, multi-rate models allow different tasks to have different rates. Representative works that use single-rate models are [90, 91, 69, 92, 93]. Kim et al. [90] propose an analytical method to provide tight bounds for applications modelled by a set of task graphs. The application of real-time calculus and minimum cycle mean analysis to Homogeneous Synchronous DataFlow Graphs (HSDFG) are investigated by Thiele et al. [91] and Karp et al. [69], respectively. HSDFG are a subset of SDFG in which all tasks have the same rates. Nielsen et al. [93] address the problem of finding the critical cycle of the concurrent systems modelled by an extension of marked graphs, called signal graphs. Hulgaard et al. [92] provide an algorithm to compute the exact bounds on the time separation of events for cyclic directed graphs. Application of these approaches to multi-rate models is also possible by converting these models to single-rate models. However, we provide an analysis technique that can be directly applied to multi-rate models. Applications that are multi-rate by nature, can be analysed more accurately and faster if we use analysis techniques that can be directly applied to multi-rate models.

There are several approaches that can be directly applied to multi-rate applications realized on multi-processor systems. Wiggers et al. and Lele et al. [94, 87] embed the behaviour of the scheduler into the SDFG model of the application and use existing analysis methods for the self-timed execution of SDFG. These approaches result in an SDFG which is more complicated than the model of the application itself. Consequently, it takes much time to analyse [88]. Wiggers et al. [94] use latency-rate servers to model the resources. This model gives pessimistic bounds, even for a sequence of consecutive firings. The method of Lele et al. [87] improves the response times compared to [94] by exploiting the fact that the completion times of consecutive iterations of tasks scheduled using budget schedules display a cyclic pattern, as long as the size of allocated slice and replenishment interval are rational numbers. The authors construct a dataflow model that precisely models this cyclic pattern, thus accurately mimicking the worst-case behaviour per iteration, rather than generalizing over all iterations. However, for a range of combinations of replenishment interval and allocated budget sizes, the model is not scalable. It requires additional conservative approximations to become scalable. This results in poor estimations on the bounds [87]. Stuijk et al. [85] provide an operational semantics for SDFG realized on deterministic shared resources. Then, it uses explicit simulation of the resources to find the periodic phase. The analysis time therefore is longer than the approaches that use a symbolic analysis method [88, 31]. The tightness of the bounds obtained by explicit simulation and symbolic simulation is similar but better than the method of Wiggers et al. [94].

The work most related to ours done by Siyoum et al. [88]. The authors have combined symbolic simulation in $(\max, +)$ algebra with worst-case resource availability curves. Then, an approach is provided to improve the WCRT of the tasks for consecutive task executions on the same resource by providing a rule that allows symbolic identification of the consecutive executions. We found that this rule is often too restrictive in the sense that it does not allow the identification of consecutive executions when the task has dependencies other than those from the previous tasks. In this chapter we provided an alternative approach that more often identifies consecutive executions.

7.6 Conclusion

We have proposed an analysis technique that provides conservative, but tight timing bounds for application scenarios running on shared resources. Our approach uses WCRCs to obtain the WCRT of tasks and then uses them in the symbolic simulation of the scenario. It results in a state matrix that mimics the worst-case execution of the application. We obtain tighter bounds than the state of the art by identifying consecutive firings on the processors. For consecutive firings, we are able to use the accumulated worst-case response times that are less pessimistic than worst-case response times per individual task.

Chapter 8

Concluding Remarks

This chapter concludes the thesis. A summary of the main findings is presented in Section 8.1. Section 8.2 contains suggestions for future research.

8.1 Conclusions

This thesis has focused on modelling and analysis of the timing behaviour of dynamic real-time streaming applications. This section summarizes the main conclusions. Chapters 1 and 2 are not discussed as they do not contain new research material.

In Chapter 3, we first provided an approach to model (static or dynamic) multi-scale applications by Scenario-Aware Dataflow (SADF) models. This approach is based on the observation that the timing behaviour of multi-scale applications follows a periodic pattern of smaller, static behaviours. This pattern includes many repetitions of sub-patterns associated with finer scale behaviours. By perceiving each distinctive, smaller behaviour as a scenario, the behaviour of the application is described as a periodic sequence of scenarios, some of which are repeated many times. The language of this sequence is represented by a regular expression for compactness. We used this technique to model two realistic real-time applications and showed that the throughput analysis can be performed in a short amount of time for these models compared to SDF and CSDF models.

Second, we provided an approach to quickly generate the execution traces of SADF sequences, the language of which is represented by a regular expression. We showed that using the regular expression representation, the execution traces of a particular scenario within the sequence can be generated without performing a detailed simulation of the sequence that is followed leading up to that scenario. This method reduces the run-time of programs that generate execution traces of multi-scale applications.

In Chapter 4, we provided a compositional approach for exact, worst-case throughput and maximum latency analyses for SADF with regular expression representation. We showed that the proposed throughput analysis for multi-scale dataflow models scales better compared to the state of the art. The latency analysis proposed in this thesis is the first exact latency method for SADF models.

In Chapter 5, we provided a method to compose SADF models of cyclo-static application modules to automatically generate SADF models for complex applications that are composed of multiple cyclo-static modules. In the core of our composition method, is an algorithm that, given the periodic scenario sequences of the modules by regular expressions, generates a sequence of composite scenarios in a compact regular expression representation. We used this technique to generate SADF models of complex applications such as multi-resolution filters from the models of its constituent modules in a short time.

In Chapter 6, we provided the first throughput-buffering trade-off analysis for applications whose behaviour is captured by SADF models. We developed a guided design-space exploration to obtain the trade-offs. The exploration space is pruned without losing any optimal points as the exploration continues. This is achieved by a novel throughput computation method that identifies the buffers that limit the throughput. The exploration continues only on these buffers, and consequently, the analysis is terminated in a reasonably short time. We applied our analysis on SADF models of several applications and obtained the optimal trade-off points in a short amount of time.

In Chapter 7, we provided an analysis method to obtain conservative, but tight timing bounds for application scenarios running on shared multi-processor systems. Such analysis method is often used in iterative design schemes to find required processor shares for all applications that are running on the same multi-processor system. We obtain tighter bounds with respect to the state of the art by finding less conservative, but still guaranteed upper bounds on the response times of application tasks that are executed on the processors. We showed that our method improves, i.e. increases, the lower bounds on the throughput obtained for applications running on multi-processor systems compared to the state of the art. This leads to reduced resource allocation when used in automated design flows. The level of improvement depends on the scheduler and application parameters.

Overall, the goal of this thesis was to provide modelling and analysis techniques for the timing behaviour of real-time streaming applications with dynamic, multi-scale, composite behaviours, such that they can be optimally designed by model-based design procedures. A fast and accurate modelling and analysis is important in model-based design schemes. We showed that we can model these applications in such a way that they can be quickly and accurately analysed. Moreover, we provided a few important analysis techniques that can be used in model-based design schemes to realize real-time applications on multi-processor systems.

8.2 Future Work

In Chapter 3 we introduced a method to represent real-time streaming applications using SADF models, and in Chapter 4 we showed that they can be analysed faster than SDF and CSDF models. Many multi-scale streaming applications have already been modelled by SDF or CSDF graphs. It is not straight forward how we can transform existing SDF and CSDF application models to their equivalent SADF models (equivalent in the sense that they represent the same timing behaviour), to exploit the scalable analysis. There exists a transformation from CSDF to SADF [55]; however, the resulting SADF lacks the properties that the proposed scalable analysis takes advantage of. For instance, it generates scenarios with a huge number of actor firings which results in long analysis time. A future work is to find an algorithmic way to transform SDF and CSDF models to SADF models to which we can apply the more scalable analysis.

In Section 2.8 we discussed that the throughput analysis of Geilen et al. [55], and the consequence that the throughput analysis proposed in Chapter 4 may be conservative when the FSM representation of the language of the SADF model has acceptance conditions. In the same section we also discussed the cases for which the analysis may be conservative. In addition, we suggested a way to obtain exact throughput for those cases by performing a throughput preserving transformation from an FSM with acceptance conditions to an FSM without an acceptance condition. A future work is to find an algorithmic method to perform such a throughput-preserving transformation.

In Chapter 5 we introduced an approach to generate SADF models of cyclo-static applications, compositionally, from the SADF models of their modules. A future work is to generalize the proposed composition method to SADF models with non-deterministic scenario transitions. A naive approach is to convert the regular expressions of the modules to FSMs, compute the product of these FSMs, and convert the product FSM to a regular expression. A major drawback for this approach is that the repetitive patterns of the given regular expressions will not be preserved, since the product FSM combines all states and transitions of the input FSMs in a flat FSM. Therefore, the end result would be a flat regular expression which takes a long time to generate and analyse.

In Chapter 6 we introduced a throughput-buffering trade-off analysis. The analysis uses the proposed extended matrix representation of scenarios with the throughput computation method of Geilen et al. [55] to obtain the throughput and at the same time identify the critical buffers. In Chapter 4 we introduced a scalable throughput analysis. Using the extended matrix representations with the scalable analysis will benefit the scalability of the throughput-buffering trade-off analysis. Such an approach requires the generalization of the technique we proposed for generating the abstract scenarios in Chapter 4. This means we need to compute the extended matrix representations for abstract scenarios instead of state matrices.

Another future work for the throughput-buffering trade-off analysis is to apply this analysis on the models generated by the compositional technique introduced in Chapter 5. This requires the computation of the extended matrix representations for the composite scenarios, to be able to identify critical buffers in composite scenarios. A challenge is to deal with shared links which are included in composite scenarios. Since their behaviour is different form dataflow channels, we need to investigate under what circumstances they can become critical.

In Chapter 7 we introduced a technique to compute the state matrices of application scenarios that are running on a shared multi-processor system. A future work is to compute the extended matrix representations of scenarios running on multi-processor systems. We can use such matrices in a design space exploration scheme to generate optimal trade-offs in the space of throughput, buffer size and processor shares. This analysis provides more design options by allowing the buffer space to be traded for processor shares and vice versa. The augmented symbolic simulation introduced in Chapter 7 can be integrated with the one introduced in Chapter 6 to generate extended matrix representations of scenarios that are running on shared multi-processor systems. To compute the extended matrix representations we need to find the dominating tokens for every token during the simulation. It is not clear how to identify these tokens in a shared multi-processor setting since there is uncertainty in the production times of tokens.

Bibliography

[1] P. Marwedel. *Embedded system design.* Springer, 2006.

[2] R. Raman. Image processing data flow in digital cameras. In *Digital Solid State Cameras: Designs and Applications*, pages 83–90. International Society for Optics and Photonics, 1998.

[3] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (ICCC)*, pages 179–196. Springer, 2002.

[4] A. Tekalp. *Digital video processing.* Prentice Hall Press, 2015.

[5] A. Jerraya and W. Wolf. The what, why, and how of mpsocs. *Multiprocessor Systems-on-Chips*, pages 1–18, 2005.

[6] S. Stuijk. *Predictable mapping of streaming applications on multiprocessors.* PhD thesis, Technische Universiteit Eindhoven, 2007.

[7] F. Yin, D. Makris, and S. Velastin. Performance evaluation of object tracking algorithms. In *Proceedings of International Workshop on Performance Evaluation of Tracking and Surveillance*, pages 1–25. IEEE, 2007.

[8] S. van der Vlucht, H. Alizadeh Ara, R. de Jong, M. Hendriks, R. Guerra Marin, M. Geilen, and D. Goswami. Modeling and analysis of FPGA accelerators for real-time streaming video processing in the healthcare domain. *Journal of Signal Processing Systems*, 2018.

[9] P. Corke and M. Good. Dynamic effects in visual closed-loop systems. *IEEE Transactions on Robotics and Automation*, 12(5):671–683, 1996.

[10] S. Skogestad and I. Postlethwaite. *Multivariable feedback control: analysis and design*, volume 2. Wiley, 2007.

[11] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. PhD thesis, Stanford University, 1987.

[12] V. Bhaskaran and K. Konstantinides. *Image and video compression standards: algorithms and architectures*. Springer, 1997.

[13] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[14] M. Quinn and R. Miller. Parallel processing. In *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., 2003.

[15] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[16] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *Proceedings of the European Symposium on Programming*, pages 319–334. Springer, 2003.

[17] T. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995.

[18] S. Eggers and R. Katz. *Evaluating the performance of four snooping cache coherency protocols*. ACM, 1989.

[19] M. Hübner and J. Becker. *Multiprocessor system-on-chip: hardware design and tool integration*. Springer, 2010.

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

[21] J. Keinert, C. Haubelt, and J. Teich. Modeling and analysis of windowed synchronous algorithms. In *International Conference on Acoustics Speech and Signal Processing Proceedings*, volume 12. IEEE, 2006.

[22] S. Bhattacharyya, P. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(2):151–166, 1999.

[23] E. A. Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[24] M. Wiggers, M. Bekooij, and G. Smit. Monotonicity and run-time scheduling. In *Proceedings of the 7th international conference on Embedded software*, pages 177–186. ACM, 2009.

[25] F. Siyoum. *Worst-case Temporal Analysis of Real-time Dynamic Streaming Applications.* PhD thesis, Technische Universiteit Eindhoven, 2014.

[26] M. Sen, I. Corretjer, F. Haim, S. Saha, J. Schlessman, T. Lv, S. Bhattacharyya, and W. Wolf. Dataflow-based mapping of computer vision algorithms onto FPGAs. *EURASIP Journal on Embedded Systems*, 2007(1), 2007.

[27] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Trans. on signal processing*, 44(2):397–408, February 1996.

[28] D. Goswami S. Mohamed, D. Zhu and T. Basten. Optimising quality-of-control for data-intensive multiprocessor image-based control systems considering workload variations. In *Digital System Desing.* IEEE, 2018.

[29] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[30] M. Bamakhrama. *On Hard Real-Time Scheduling of Cyclo-Static Dataflow and its Application in System-Level Design.* PhD thesis, Universiteit Leiden, 2014.

[31] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 125–134, 2010.

[32] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of the 4th International Conference on Formal Methods and Models for Co-Design*, pages 185–194. IEEE, 2006.

[33] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proceedings of The International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 63–72. ACM, 2003.

[34] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. Bhattacharyya. A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Proceedings of the 8th International Conference on Embedded Software*, pages 189–198. ACM, 2008.

[35] J. L. Pino, S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling system for dsp applications. In *Conference Record of The 29th Asilomar Conference on Signals, Systems and Computers*, pages 122–126. IEEE, 1995.

[36] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs. *ACM Transations on Embedded Computing Systems (TECS)*, 12(3), 2013.

[37] M. Skelin and M. Geilen. Compositionality in scenario-aware dataflow: A rendezvous perspective. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2018.

[38] G. Nicolescu and P. Mosterman. *Model-based design for embedded systems.* CRC Press, 2009.

[39] H. Alizadeh Ara, A. Behrouzian, M. Hendriks, M. Geilen, D. Goswami, and T. Basten. Scalable analysis for multi-scale dataflow models. *Transactions on Embedded Computing Systems*, 17(4), 2018.

[40] H. Alizadeh Ara, A. Behrouzian, M. Geilen, M. Hendriks, D. Goswami, and T. Basten. Analysis and visualization of execution traces of dataflow applications. In *Proceedings of the 2nd Embedded computing and Architecture (IDEA) Workshop on Integrating Dataflow*, pages 19–20. ESR-2017-01, 2016.

[41] H. Alizadeh Ara, M. Geilen, A. Behrouzian, T. Basten, and D. Goswami. Compositional dataflow modelling for cyclo-static applications. In *Proceedings of the 21st Euromicro Conference on Digital System Design (DSD)*, pages 121–129. IEEE, 2018.

[42] H. Alizadeh Ara, M. Geilen, A. Behrouzian, and T. Basten. Throughput-buffering trade-off analysis for scenario-aware dataflow models. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 265–275. ACM, 2018.

[43] H. Alizadeh Ara, M. Geilen, T. Basten, A. Behrouzian, M. Hendriks, and D. Goswami. Tight temporal bounds for dataflow applications mapped onto shared resources. In *Proceedings of the 11th Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2016.

[44] F. Baccelli, G. Cohen, G. Olsder, and J. Quadrat. *Synchronization and linearity*, volume 2. Wiley New York, 1992.

[45] B. Heidergott, G. J. Olsder, and J. Van der Woude. *Max Plus at work: modeling and analysis of synchronized systems: a course on Max-Plus algebra and its applications*, volume 48. Princeton University Press, 2014.

[46] S. Sriram and S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization.* CRC press, 2000.

[47] I. Liu, J. Reineke, and E. A. Lee. A pret architecture supporting concurrent programs with composable timing properties. In *Conference Record of the 44th Asilomar Conference on Signals, Systems and Computers*, 2010.

[48] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, et al. Virtual execution platforms for mixed-time-criticality systems: the compsoc architecture and design flow. *ACM SIGBED Review*, 2013.

[49] A. H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij. Throughput analysis of synchronous data flow graphs. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 25–36, 2006.

[50] A. Bonfietti, M. Lombardi, M. Milano, and L. Benini. Throughput constraint for synchronous data flow graphs. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 26–40. Springer, 2009.

[51] S. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, et al. System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1), 2009.

[52] M. Harrison. *Introduction to formal language theory.* Addison-Wesley Longman Publishing Co., 1978.

[53] J Richard Büchi. On a decision method in restricted second order arithmetic. In *The Collected Works of J. Richard Büchi*, pages 425–435. Springer, 1990.

[54] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*. Springer Berlin Heidelberg, 1981.

[55] M. Geilen, J. Falk, C. Haubelt, T. Basten, B. Theelen, and S. Stuijk. Performance analysis of weakly-consistent scenario-aware dataflow graphs. *Signal Processing Systems*, 87:157–175, 2017.

[56] L. Zadeh and C. Desoer. *Linear system theory: the state space approach.* Courier Dover Publications, 2008.

[57] M. Skelin and M. Geilen. Towards component-based (max,+) algebraic throughput analysis of hierarchical synchronous data flow models. In *Computer Safety, Reliability, and Security.* Springer, 2017.

[58] A. H. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. Theelen. Latency minimization for synchronous data flow graphs. In *Proceedings of The 10th Euromicro Conference on Digital System Design*, pages 189–196, 2007.

[59] R. de Groote, P. K. F. Hölzenspies, J. Kuper, and H. Broersma. Back to basics: Homogeneous representations of multi-rate synchronous dataflow graphs. In *Porceedings of 11th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 35–46. IEEE, 2013.

[60] S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala. Innovation and intellectual property rights. In *Handbook of signal processing systems*, chapter 21, pages 751–786. Springer International Publishing, 2019.

[61] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197 – 213, 1993.

[62] S. Gaubert. Performance evaluation of (max,+) automata. *IEEE Transations on automatic control*, 40(12), 1995.

[63] C. Hagenah and A. Muscholl. Computing ϵ-free NFA from regular expressions in $\mathcal{O}(n \log^2 n)$ time. *RAIRO Inform. Théor*, pages 257–277, 2000.

[64] T. Cormen. *Introduction to algorithms.* MIT press, 2009.

[65] S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF For Free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 276–278. IEEE, 2006.

[66] J. Keinert, H. Dutta, F. Hannig, C. Haubelt, and J. Teich. Model-based synthesis and optimization of static multi-rate image processing algorithms. In *Proceedings of the 9th Design, Automation and Test in Europe conference*, pages 135–140. IEEE, 2009.

[67] R. de Groote, P. Hölzenspies, J. Kuper, and G. Smit. Single-rate approximations of cyclo-static synchronous dataflow graphs. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, pages 11–20. ACM, 2014.

[68] M. Gondran and M. Minoux. *Graphs and algorithms.* Wiley, 1984.

[69] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309 – 311, 1978.

[70] O. Moreira and H. Corporaal. *Scheduling real-time streaming applications onto an embedded multiprocessor*, pages 67–75. Springer, 2014.

[71] A. Nakamura, T. Saito, I. Takigawa, M. Kudo, and H. Mamitsuka. Fast algorithms for finding a minimum repetition representation of strings and trees. *Discrete Applied Mathematics*, 2013.

[72] H. Deroui, K. Desnos, J. F. Nezan, and A. Munier-Kordon. Throughput evaluation of dsp applications based on hierarchical dataflow models. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS).* IEEE, 2017.

[73] J. Piat, S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *Proceedings of the Workshop on Signal Processing Systems.* IEEE, 2009.

[74] M. Geilen, S. Tripakis, and M. Wiggers. The earlier the better: A theory of timed actor interfaces. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, pages 23–32. ACM, 2011.

[75] J.P.H.M. Hausmans, S. J. Geuns, M. Wiggers, and M. Bekooij. Compositional temporal analysis model for incremental hard real-time system design. In *Proceedings of the 10th International Conference on Embedded Software*, pages 185–194. ACM, 2012.

[76] Praveen K. Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow.* PhD thesis, EECS Department, University of California, Berkeley, 1996.

[77] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proceedings of the 44th Design Automation Conference.* IEEE, 2007.

[78] M. Hendriks, H. Alizadeh Ara, M. Geilen, T. Basten, R. Guerra Marin, R. de Jong, and S. van der Vlugt. Monotonic optimization of dataflow buffer sizes. *Journal of Signal Processing Systems*, 2018.

[79] R. van Kampenhout, S. Stuijk, and K. Goossens. Programming and analysing scenario-aware dataflow on a multi-processor platform. In *Proceedings of the 17th Design, Automation Test in Europe conference*, pages 876–881, March 2017.

[80] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 4, pages 2651–2654, May 1995.

[81] C. T. Hwang, J. H. Lee, and Y. C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, Apr 1991.

[82] R. Govindarajan, Guang R. Gao, and Palash Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI signal processing systems for signal, image and video technology*, 31(3), 2002.

[83] Kristof Denolf, Adrian Chirila-Rus, Paul Schumacher, Robert Turney, Kees Vissers, Diederik Verkest, and Henk Corporaal. A systematic approach to design low-power video codec cores. *EURASIP J. Embedded Syst.*, 2007(1):42–42, January 2007.

[84] B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. Periodic schedules for cyclo-static dataflow. In *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 105–114, 2013.

[85] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th Design Automation Conference*, DAC '07, pages 777–782, New York, NY, USA, 2007. ACM.

[86] A.J.M. Moonen, M. Bekooij, and J. Meerbergen, van. Timing analysis model for network based multiprocessor systems. In *Proceedings of the 15th ProRISC, Annual Workshop on Circuits, Systems and Signal Processing*. STW Technology Foundation, 2004.

[87] A. Lele, O. Moreira, and P. Cuijpers. A new data flow analysis model for tdm. In *Proceedings of the 10th international conference on Embedded software*, pages 237–246. ACM, 2012.

[88] F. Siyoum, M. Geilen, and H. Corporaal. Symbolic analysis of dataflow applications mapped onto shared heterogeneous resources. In *Proceedings of the 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[89] Wolfgang Stadje. The busy period of the queueing system $m/g/\infty$. *Journal of Applied Probability*, 22(3):697–704, 1985.

[90] J. Kim, H. Oh, J. Choi, H. Ha, and S. Ha. A novel analytical method for worst case response time estimation of distributed embedded systems. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013.

[91] L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *Proceedings of the 7th international conference on Embedded software*. ACM, 2009.

[92] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *Computers, IEEE Transactions on*, 1995.

[93] C. D. Nielsen and M. Kishinevsky. Performance analysis based on timing simulation. In *Proc. 31st annual Design Automation Conference*. ACM, 1994.

[94] M. Wiggers, M. Bekooij, and G. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proceedings of the 10th international workshop on software and compilers for embedded systems.* ACM, 2007.

[95] A. Behrouzian, D. Goswami, T. Basten, M. Geilen, H. Alizadeh Ara, and M.Hendriks. Firmness analysis of real-time applications under static-priority preemptive scheduling. In *Proceedings of Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 2018.

[96] A. Behrouzian, D. Goswami, M. Geilen, H. Alizadeh Ara, E.P. van Horssen, W.P.M.H. Heemels, and T. Basten. Sample-drop firmness analysis of TDMA-scheduled control applications. In *Proceedings of the 11th International Symposium on Industrial Embedded Systems (SIES).* IEEE, 2016.

[97] M. Hendriks, M. Geilen, A. Behrouzian, T. Basten, H. Alizadeh Ara, and D. Goswami. Checking metric temporal logic with TRACE. In *Proceedings of the 16th International Conference on Application of Concurrency to System Design (ACSD).* IEEE, 2016.

[98] S. Adyanthaya, H. Alizadeh Ara, J. Bastos, A.R.B. Behrouzian, R. Medina Sánchez, J. van Pinxten, B. van der Sanden, U. Waqas, T. Basten, H. Corporaal, R. Frijns, M. Geilen, D. Goswami, M. Hendriks, S. Stuijk, M. Reniers, and J. Voeten. xCPS: A tool to explore cyber physical systems. *ACM SIGBED Review*, 14(1), 2017.

[99] A. Behrouzian, D. Goswami, T. Basten, M. Geilen, and H. Alizadeh Ara. Multi-constraint multi-processor resource allocation. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS).* IEEE, 2015.

[100] S. Adyanthaya, H. Alizadeh Ara, J. Bastos, A. Behrouzian, R. Medina Sánchez, J. van Pinxten, B. van der Sanden, U. Waqas, T. Basten, H. Corporaal, R. Frijns, M. Geilen, D. Goswami, S. Stuijk, M. Reniers, and J. Voeten. xCPS: A tool to explore cyber physical systems. In *Proceedings of the 15th Workshop on Embedded and Cyber-Physical Systems Education (WESE15).* ACM, 2015.

Acronyms

CF Convolution Filter.

CNNs Convolutional Neural Networks.

CSDF Cyclo-Static Dataflow.

DDG Deadlock Dependency Graph.

DS Down-Sampler.

DSE Design Space Exploration.

EMPA Extended $(\max, +)$ Automaton.

EMR Extended Matrix Representation.

EST Earliest Start Time.

ESTV Extended Symbolic Time-stamp Vector.

FIFO First-In-First-Out.

FSM Finite State Machine.

FSM-SADF Finite-State-Machine Scenario-Aware Dataflow.

FTs Final Tokens.

HSDF Homogeneous Synchronous Dataflow.

IDCT Inverse Discrete Cosine Transformation.

IQ Inverse Quantization.

ITs Initial Tokens.

KPNs Kahn Process Networks.

MCM Maximum Cycle Mean.

MCR Maximum Cycle Ratio.

MoC Model of Computation.

MoCs Models of Computation.

MPA $(\max, +)$ Automaton.

MRCF Multi-Resolution Convolution Filter.

PC Producer-Consumer.

RC Reconstruction.

RTOSs Real-Time Operating Systems.

SADF Scenario-Aware Dataflow.

SDF Synchronous Dataflow.

SDFG Synchronous Dataflow Graph.

TDMA Time Division Multiple Access.

US Up-Sampler.

VLD Variable Length Decoding.

WCRC Worst-Case Resource Curve.

WCRT Worst-Case Response Time.

Acknowledgments

Undertaking this PhD has been a life-changing experience for me, a period of intense learning, not only in the scientific arena, but also on a personal level. I could have not been able to complete this journey without the support and guidance that I received from many people.

Firstly, I would like to thank my promoter, Twan Basten, from whom I received constant support and guidance through my entire PhD journey. My weekly meetings from 11:00 hrs to 12:00 hrs on Thursdays, with Twan and other members of the ALMARVI team were among the best moments of the week. Every meeting instance, which was about one or two hours, started with Twan saying “ALMARVI” in a low tone voice. These meetings were fruitful, encouraging and most importantly, fun.

Marc Geilen, my co-promoter, invested a great deal of effort in me. I owe him a great debt of gratitude for being my teacher and my friend. His office door was always open whenever I ran into trouble or had a question about my research. He steered me in the right direction whenever he thought I needed it, until the very last moments of my PhD. He helped me to get the theorems, proofs and all technical details sound. Thank you Marc for all the help and support.

I wish to thank the members of my thesis committee: Edward Lee, Bernd Heidergott, Kees van Berkel and Jeroen Voeten for generously offering their time and good will throughout the review process of this document. I offer my special thanks to Edward for his detailed feedback and suggestions.

As always, I felt blessed to have my life-long friend, Amir Behrouzian, next to me in the past four years. If it was not for him, perhaps I was never in the Netherlands and this thesis was never written. He always encouraged and inspired me. Thank you Amir for your brotherly support.

I would like to express my gratitude to my fellow PhD students, Rasool Tavakoli, João Bastos, Bram van der Sanden, Joost van Pinxten, Róbinson Medina Sánchez, Umar Waqas, Juan Valencia and Reinier van Kampenhout. I enjoyed keeping your company and I am grateful for the help I got from

you all during my PhD. Special thanks to Joost for his technical and social support, especially in the beginning of my PhD, when I needed it the most. As a foreigner, I got a lot of tips about daily life in the Netherlands from Rasool, as he moved to the Netherlands a few months before I did. Thank you Rasool for your tips on various subjects, including but not limited to food, medicine, travel, cars, Ikea and plants.

I was pleased to work with a group of talented individuals in the context of the ALMARVI project. Dip Goswami, Martijn Hendriks, Steven van der Vlugt and Rob de Jong. I would like to thank them for their wonderful collaboration.

I had a great experience being a member of the ES group. A warm word for my colleague and great friend Mladen Skelin, with whom I had the best lunch breaks. I would like to thank Andrew Nelson for his support with the Computation course. I would like to thank Marja de Mol, Margot Gordon and Rian van Gaalen. Their presence was very important in a process that is often felt as tremendously solitaire. I would like to thank them for organizing ES-days, *gezellige jaarlijkse teamuitjes*, fun annual team outings, which I am going to remember for the rest of my life.

I would like to thank my friends in TU/e, Samaneh Tadayon Mousavi, Reyhaneh Mahlouji and Amir Ghahremani for all the joyful times we had together.

I would like to thank Ali Doost Mohammadi, my supervisor during my master studies. He introduced me to the subject of discrete event systems and $(\max, +)$ algebra, and helped me in completing my master thesis.

I deeply thank my parents, Mansoureh Elmi and Mirghasem Alizadeh Ara for their unconditional trust, timely encouragement, and endless patience despite the long distance between us. It was their love that brought me to this point in my life. My brother Hessam has also been generous with his love and encouragement. Thank you Hessam for being there for my parents when I was absent.

Last but not least my regards goes to my girlfriend Roos van den Bekerom. She has been my best friend and great companion, supported, encouraged, entertained, and helped me get through this agonizing, last period in the most positive way.

Curriculum Vitae

Seyedhadi Seyedalizadeh Ara was born on 19-07-1987 in Tabriz, Iran. After finishing college in 2005 at Dabirestan-e-Seghatoleslam in Tabriz, Iran, he studied Bachelor of Science at Sahand University of Technology in the same city. He Studied Master of Science at Amirkabir University (Tehran Polytechnic) in Tehran, Iran. In 2013 he graduated within the Control Systems group on Modelling and Control of Manufacturing Systems. From October 2014 he started a PhD project at Technische Universiteit Eindhoven in Eindhoven, the Netherlands, of which the results are presented in this thesis. Since November 2018, he is employed at Océ.

Publication List

Publications covered in the thesis

1. H. Alizadeh Ara, M. Geilen, A. Behrouzian, and T. Basten. Throughput-buffering trade-off analysis for scenario-aware dataflow models. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 265–275. ACM, 2018

2. H. Alizadeh Ara, M. Geilen, A. Behrouzian, T. Basten, and D. Goswami. Compositional dataflow modelling for cyclo-static applications. In *Proceedings of the 21st Euromicro Conference on Digital System Design (DSD)*, pages 121–129. IEEE, 2018

3. H. Alizadeh Ara, A. Behrouzian, M. Hendriks, M. Geilen, D. Goswami, and T. Basten. Scalable analysis for multi-scale dataflow models. *Transactions on Embedded Computing Systems*, 17(4), 2018

4. H. Alizadeh Ara, A. Behrouzian, M. Geilen, M. Hendriks, D. Goswami, and T. Basten. Analysis and visualization of execution traces of dataflow applications. In *Proceedings of the 2nd Embedded computing and Architecture (IDEA) Workshop on Integrating Dataflow*, pages 19–20. ESR-2017-01, 2016

5. H. Alizadeh Ara, M. Geilen, T. Basten, A. Behrouzian, M. Hendriks, and D. Goswami. Tight temporal bounds for dataflow applications mapped onto shared resources. In *Proceedings of the 11th Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2016

Publications not covered in the thesis

1. A. Behrouzian, D. Goswami, T. Basten, M. Geilen, H. Alizadeh Ara, and M.Hendriks. Firmness analysis of real-time applications under static-priority preemptive scheduling. In *Proceedings of Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018

2. S. van der Vlucht, H. Alizadeh Ara, R. de Jong, M. Hendriks, R. Guerra Marin, M. Geilen, and D. Goswami. Modeling and analysis of FPGA accelerators for real-time streaming video processing in the healthcare domain. *Journal of Signal Processing Systems*, 2018

3. M. Hendriks, H. Alizadeh Ara, M. Geilen, T. Basten, R. Guerra Marin, R. de Jong, and S. van der Vlugt. Monotonic optimization of dataflow buffer sizes. *Journal of Signal Processing Systems*, 2018

4. A. Behrouzian, D. Goswami, M. Geilen, H. Alizadeh Ara, E.P. van Horssen, W.P.M.H. Heemels, and T. Basten. Sample-drop firmness analysis of TDMA-scheduled control applications. In *Proceedings of the 11th International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2016

5. M. Hendriks, M. Geilen, A. Behrouzian, T. Basten, H. Alizadeh Ara, and D. Goswami. Checking metric temporal logic with TRACE. In *Proceedings of the 16th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 2016

6. S. Adyanthaya, H. Alizadeh Ara, J. Bastos, A.R.B. Behrouzian, R. Medina Sánchez, J. van Pinxten, B. van der Sanden, U. Waqas, T. Basten, H. Corporaal, R. Frijns, M. Geilen, D. Goswami, M. Hendriks, S. Stuijk, M. Reniers, and J. Voeten. xCPS: A tool to explore cyber physical systems. *ACM SIGBED Review*, 14(1), 2017

7. A. Behrouzian, D. Goswami, T. Basten, M. Geilen, and H. Alizadeh Ara. Multi-constraint multi-processor resource allocation. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 2015

8. S. Adyanthaya, H. Alizadeh Ara, J. Bastos, A. Behrouzian, R. Medina Sánchez, J. van Pinxten, B. van der Sanden, U. Waqas, T. Basten, H. Corporaal, R. Frijns, M. Geilen, D. Goswami, S. Stuijk, M. Reniers,

and J. Voeten. xCPS: A tool to explore cyber physical systems. In *Proceedings of the 15th Workshop on Embedded and Cyber-Physical Systems Education (WESE15)*. ACM, 2015