Timing analysis of synchronous data flow graphs

Citation for published version (APA):
Ghamarian, A. H. (2008). *Timing analysis of synchronous data flow graphs*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR635718

DOI:
10.6100/IR635718

Document status and date:
Published: 01/01/2008

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: 17. Nov. 2023

Timing Analysis of Synchronous Data Flow Graphs

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op vrijdag 4 juli 2008 om 16.00 uur

door

Amir Hossein Ghamarian

geboren te Teheran, Iran

Dit proefschrift is goedgekeurd door de promotor:

prof.dr.ir. R.H.J.M. Otten

Copromotoren:
dr.ir. T. Basten
en
dr.ir. M.C.W. Geilen

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Ghamarian, Amir Hossein

Timing Analysis of Synchronous Data Flow Graphs / by Amir Hossein Ghamarian.
Eindhoven: Technische Universiteit Eindhoven, 2008.
Proefschrift. - ISBN 978-90-386-1315-4
NUR 980
Trefw.: petri-netwerken / multiprogrammeren / parallelle processen /
ingebedde systemen.
Subject headings: data flow analysis / data flow graphs
multiprocessing systems / embedded systems.

Timing Analysis of Synchronous Data Flow Graphs

Amir Hossein Ghamarian

Committee:

prof.dr.ir. R.H.J.M. Otten (promotor, TU Eindhoven)
dr.ir. T. Basten (copromotor, TU Eindhoven)
dr.ir. M.C.W. Geilen (copromotor, TU Eindhoven)
prof.dr. J.C.M. Baeten (TU Eindhoven)
prof.dr. H. Corporaal (TU Eindhoven)
prof.dr.ir. H.J. Sips (TU Delft)
prof.dr. S.S. Bhattacharyya (University of Maryland, USA)

NWO
Netherlands Organisation for Scientific Research

The work in this thesis is supported by the Dutch government in their NWO research program within the PROMES project 612.064.206.

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 165.

© Amir Hossein Ghamarian 2008. All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Printing: Printservice Technische Universiteit Eindhoven

Abstract

Timing Analysis of Synchronous Data Flow Graphs

Consumer electronic systems are getting more and more complex. Consequently, their design is getting more complicated. Typical systems built today are made of different subsystems that work in parallel in order to meet the functional requirements of the demanded applications. The types of applications running on such systems usually have inherent timing constraints which should be realized by the system. The analysis of timing guarantees for parallel systems is not a straightforward task.

One important category of applications in consumer electronic devices are multimedia applications such as an MP3 player and an MPEG decoder/encoder. Predictable design is the prominent way of simultaneously managing the design complexity of these systems and providing timing guarantees. Timing guarantees cannot be obtained without using analyzable models of computation. Data flow models proved to be a suitable means for modeling and analysis of multimedia applications. Synchronous Data Flow Graphs (SDFGs) is a data flow model of computation that is traditionally used in the domain of Digital Signal Processing (DSP) platforms. Owing to the structural similarity between DSP and multimedia applications, SDFGs are suitable for modeling multimedia applications as well. Besides, various performance metrics can be analyzed using SDFGs. In fact, the combination of expressivity and analysis potential makes SDFGs very interesting in the domain of multimedia applications.

This thesis contributes to SDFG analysis. We propose necessary and sufficient conditions to analyze the integrity of SDFGs and we provide techniques to capture prominent performance metrics, namely, throughput and latency. These performance metrics together with the mentioned sanity checks (conditions) build an appropriate basis for the analysis of the timing behavior of modeled applications.

An SDFG is a graph with actors as vertices and channels as edges. Actors represent basic parts of an application which need to be executed. Channels represent data dependencies between actors. Streaming applications essentially continue their execution indefinitely. Therefore, one of the key properties of an SDFG which models such an application is liveness, i.e., whether all actors can run infinitely often. For example, one is usually not interested in a system which

completely or partially deadlocks. Another elementary requirement known as boundedness, is whether an implementation of an SDFG is feasible using a limited amount of memory. Necessary and sufficient conditions for liveness and the different types of boundedness are given, as well as algorithms for checking those conditions.

Throughput analysis of SDFGs is an important step for verifying throughput requirements of concurrent real-time applications, for instance within design-space exploration activities. In fact, the main reason that SDFGs are used for modeling multimedia applications is analysis of the worst-case throughput, as it is essential for providing timing guarantees. Analysis of SDFGs can be hard, since the worst-case complexity of analysis algorithms is often high. This is also true for throughput analysis. In particular, many algorithms involve a conversion to another kind of data flow graph, namely, a homogenous data flow graph, whose size can be exponentially larger than the size of the original graph and in practice often is much larger. The thesis presents a method for throughput analysis of SDFGs which is based on explicit state-space exploration, avoiding the mentioned conversion. The method, despite its worst-case complexity, works well in practice, while existing methods often fail. Since the state-space exploration method is akin to the simulation of the graph, the result can be easily obtained as a byproduct in existing simulation tools.

In various contexts, such as design-space exploration or run-time reconfiguration, many throughput computations are required for varying actor execution times. The computations need to be fast because typically very limited resources or time can be dedicated to the analysis. In this thesis, we present methods to compute throughput of an SDFG where execution times of actors can be parameters. As a result, the throughput of these graphs is obtained in the form of a function of these parameters. Calculation of throughput for different actor execution times is then merely an evaluation of this function for specific parameter values, which is much faster than the standard throughput analysis.

Although throughput is a very useful performance indicator for concurrent real-time applications, another important metric is latency. Especially for applications such as video conferencing, telephony and games, latency beyond a certain limit cannot be tolerated. The final contribution of this thesis is an algorithm to determine the minimal achievable latency, providing an execution scheme for executing an SDFG with this latency. In addition, a heuristic is proposed for optimizing latency under a throughput constraint. This heuristic gives optimal latency and throughput results in most cases.

Contents

Abstract **i**

Acknowledgments **v**

1 Introduction **1**

1.1 Trends in Embedded Systems 1
1.2 Predictable Design 2
1.3 Models of Computation for Embedded Systems 3
1.4 Problem Statement 5
1.5 Bibliography 6
1.6 Contributions 7
1.7 Thesis Overview 8

2 Preliminaries **9**

2.1 Overview 9
2.2 Synchronous Data Flow Graphs 9
2.3 Timed Synchronous Data Flow Graphs 10
2.4 Static Properties 11
2.5 Dynamic Properties 12
2.6 Homogeneous SDF 16
2.7 Summary 18

3 Liveness and Boundedness **19**

3.1 Overview 19
3.2 Boundedness Definitions 20
3.3 Boundedness 22
3.4 Strict Boundedness 25
3.5 Self-timed Boundedness 26
3.6 Related Work 37
3.7 Summary 38

4 Throughput 39
- 4.1 Overview 39
- 4.2 Throughput Analysis of SDF Graphs 40
- 4.3 Max-Plus Algebraic Characterization 44
- 4.4 Experimental Results 47
- 4.5 Throughput Analysis of Arbitrary SDFGs 53
- 4.6 Related Work 54
- 4.7 Summary 54

5 Parametric Throughput 55
- 5.1 Overview 55
- 5.2 Parametric SDFGs and Throughput Analysis Methods 56
- 5.3 HSDFG Method 60
- 5.4 State-Space Method 60
- 5.5 Divide-and-Conquer Method 66
- 5.6 Experimental Results 71
- 5.7 Parametric Throughput Analysis of Arbitrary SDFGs 73
- 5.8 Related Work 73
- 5.9 Summary 74

6 Latency 75
- 6.1 Overview 75
- 6.2 Latency 76
- 6.3 Minimum Latency Executions 79
- 6.4 Static Order Scheduling Policies 82
- 6.5 Throughput Constraints 84
- 6.6 Experimental Results 89
- 6.7 Related Work 91
- 6.8 Summary 92

7 Conclusions and Future Work 93
- 7.1 Conclusions 93
- 7.2 Open Problems and Future Research 94

Curriculum Vitae 105

List of Publications 107

Acknowledgments

This thesis puts a formal end to twenty three years of my life as a student; Twenty three years filled with memories; Memories identified with people; People who played a significant role in my student life. The tradition calls for me to begin the thesis with acknowledgements, even though it would not account for how gross an understatement that acknowledgement can be.

First and foremost, I would like to thank Twan Basten for giving me the opportunity to work with him and under his supervision. I am very grateful, not only for his invaluable guidance, our discussions and his insightful comments on my manuscripts, but also for his friendship and concern for things not related to work. I have never seen anybody as enthusiastic as Twan about his work. He also communicates his enthusiasm to those connected to him. I am sure I would not have been able to finish this thesis without his help and remarkable ideas.

Furthermore, Twan initiated the weekly PROMES meetings where we discussed together with Marc Geilen, Sander Stuijk, and Bart Theelen the ongoing PhD research of Sander and me, which sparked many ideas and directed our research to its current form. For all this and more, I gratefully thank him.

Twan was so kind to create a position for me in the group for one more year. I am looking forward to more and more fruitful cooperations with him.

Marc Geilen, my daily supervisor, is one of the brightest people I have ever come across in my life. His creativity, his precise way of thinking, and his extraordinary ability to find counter-examples for my proofs, have always improved the state of almost all my research activities during the last few years.

Also, I would like to thank Prof. Ralph Otten, for reading my thesis as my promotor, and for creating a very flexible environment for the members of his group.

The other members of my thesis committee are gratefully acknowledged for reading the thesis, providing useful comments and being present in my defense session. It is my privilege to have Jos Baeten, Shuvra Bhattacharya, Henk Corporaal, and Henk Sips in my thesis committee.

My special thanks go to the programming guru, Sander Stuijk, who was working in the same project as I was. His friendship and support during the last five years have been invaluable.

Right in front of the elevators on the ninth floor of the PT building there is a cozy place, with a table and few chairs sitting on a red carpet. This place, our coffee corner, although small, is truly great. It always provided me a wonderful opportunity for me to learn about different cultures. There, people from different countries and cultures, regardless of their background, can sit for a few minutes every day and talk about any subject, without the slightest problem of any kind. Let us hope and pray for a day that man-made borders are only part of history books and let us pray and hope for a world that will look like our coffee corner. I would like to thank all the members of the Electronic Systems group for the great time and fruitful discussions we had together.

Marja de Mol and Rian van Gaalen, the secretaries of our group, have always been very kind, hospitable; they warmly received me at my arrival, and have always been helpful throughout all these years. Thank you very much!

I would also like to highly thank my sympathetic and patient house-mates, Mohammad Mousavi and Mohammad Abam, the so-called ”Mohammads”, for their warm company in the last four years. I had a great time with them which is certainly never forgotten. Mohammad Mousavi was the person who actually introduced me to Holland and the possibility of doing a PhD in this country.

My dearest friends Navid Kabiri, the Rahmani family, Reyhaneh Nikzad, Maryam Shahvelayati and Arash Jalali always remained close while thousands of kilometers away. Hereby, I would like to thank their emotional support.

I also would like to thank my brother Ehsan, and my sister Mojgan and my dearest friend Marzieh Rafieenia whose constant supports have been so heartwarming through many cold moments.

My special appreciation goes to Ehsan Baha for being a good friend and designing the elegant cover of this thesis. I express my best thanks to the Iranian families Baha, Farshi, Fatemi, Gargari, Mousavi, Malakpour, Nikbakht, Rezaeian, Sakian, Sarhangi, Talebi and Vahedi, for their help and invitations which have always given me a taste of home. Special thanks go to the members of the Saturday’s soccer team, and the other friends who were too lazy to join.

My supervisors and professors at Shairf University of Technology introduced me to the wonderful world of academic research. For that, my best thanks go to Prof. Amir Daneshgar, whose patience, hard work and intelligence made him a role model for me. I also would like to thank Rasool Jalili for the opportunity that he gave me to work in his laboratory.

Last but certainly not least comes my family. I think that now, at the end of my PhD studies, would be the right moment to express my deepest gratitude to them for their unconditional support, encouragement and faith in me throughout my whole life and in particular, during the last few years. I can only hope that one day I would be able to return, even if only in part, the love and kindness they have extended to me. I dedicate this thesis to them, *with love and gratitude.*

Amir Hossein Ghamarian
May 2008

Chapter 1

Introduction

1.1 Trends in Embedded Systems

A popular element in the James Bond franchise is the exotic equipment that is critically useful on his missions. Most of this equipment is no longer just an excitement factor of fictional novels and movies. Devices like cell phones, MP3 players, PDA's, etc are inseparable part of our daily lives. Most of these systems contain one or more processors which realize the functionality of the device. These devices are called *embedded systems*.

Not only the number of embedded systems has increased exponentially, but also the last few years witnessed a tremendous growth in their complexity as well. Just by comparing the complexity of the functionality of embedded systems built today, and those built a decade ago, one can get a good idea about the complications involved in the design of new systems. For example, a simple cell phone today, is not only a cell phone anymore; it usually works as a camera, an MP3 player, a video displayer and even an internet browser.

One important category of embedded systems are *multimedia embedded systems* in which combinations of different content forms of media are processed. Examples of such systems are cell phones, game consoles, PDA's and digital cameras. In general, multimedia includes a combination of text, audio, animation, video, and interactivity content forms. These types of data are inherently streaming, i.e., they consist of streams of data. Therefore, multimedia embedded systems typically perform regular sequences of operations on large (virtually infinite) streams of data.

Next to functional properties, multimedia applications inherently require some non-functional properties to be fulfilled as well. For example, timing constraints are usually part of the specification of embedded systems, i.e., the correctness of the system depends not only on the functional results, but also on the time at which the results are produced. A similar story is also true for energy consumption, namely, only low powered systems are desirable, and energy consumption exceeding a certain limit is not acceptable. In addition to the increase in complexity and the number of different gadgets, costs need to be reduced and time-to-market has been shortened due to the ever-increasing demand of consumers.

1.2 Predictable Design

All the points explained in the previous section in the design of embedded systems, make the design of such systems very difficult. In fact, designers are expected to design multi-functional embedded systems with correct functional and non-functional properties in a very short time and at a very low cost.

Applications are the motivation for embedded computing, i.e., computation is not the primary goal of multimedia embedded systems. Therefore, only a fraction of the total price of the products can go into the computation part. As a result, embedded systems involve computation that is subject to resource limitations such as limited memory, processor power and energy consumption.

The various issues mentioned so far have been addressed by splitting activities into different tasks and distributing them among different specialized processing cores that work in parallel. Consequential to fulfilling all the requirements of a system, a very large design space needs to be explored which is very time consuming. Designers try to use techniques that allow them to predict some of the properties of the final result in the early stages of the design. In this way a large part of the state space gets pruned, which leads to a shorter design time, and consequently shorter time-to-market and lower costs.

As mentioned in the previous section, providing guarantees is one of the most prominent non-functional features of multimedia embedded systems. An effective approach for designing systems which realize timing constraints, is to design systems that provide predictable results in a predictable amount of time. This method of design is called *predictable design*. In other words, the main goal in predictable design is to provide timing guarantees while avoiding the overdimensioning of the system. A system is considered to be predictable in case its functional behavior as well as non-functional properties (such as performance) can be forecasted based on models or specifications developed prior to actually realizing the system with hardware and software.

But predictability is harder to achieve in a multiprocessor system because the processors typically lack a global view of the system. Therefore, a design method with the primary goal of achieving end-to-end predictability in a distributed system is required which can manage the complexity of the design on the one hand and provides the timing guarantees while using resources efficiently on the other hand.

1.3 Models of Computation for Embedded Systems

Many different activities are required to carry a complex electronic system from initial idea to physical implementation. Functional modeling captures the specification of the functional behavior of the system. Performance modeling helps to understand the non-functional characteristics of the product. Validation and verification ensure that the final implementation behaves according to its specifications and expectations.

All these activities operate on models and not on the real physical object. The most important reason for using a model is that the real product is not available before the development task is completed. Achieving the optimum solution which addresses the timing correctly and uses the least amount of resources requires design-space exploration which in most cases is very time consuming and error-prone. Today's short time-to-market usually prohibits the manufacturing of a complete prototype as part of the development. Besides that, prototypes cannot replace the hundreds or thousands of different models that are routinely developed for an average electronic product. Furthermore, realizing predictable designs cannot be done without using mathematical models, because a model efficiently determines whether specific constraints are met. In other words, formal features provided by mathematical models are desirable in predictable design because they guarantee certain properties or allow us to deploy efficient synthesis procedures.

Various models of computation (MoCs) are used in designing multimedia applications [34]. MoCs can be compared based on different aspects like expressiveness and succinctness as well as analyzability. It is obvious that there is a trade-off involved between the expressiveness and succinctness of a model as opposed to its analyzability potential.

Among MoCs used for modeling multimedia applications, data flow models in general proved to be a very successful means to capture properties of such applications [34, 53, 61]. Typical multimedia applications consist of a set of tasks (or operations) that need to be performed, while data is transferred or communicated among those tasks. This set of tasks needs to be performed iteratively, while consuming and producing fixed amounts of data for each task execution. This structure of tasks connected with communication channels can be naturally modeled by data flow graphs. The nodes of a data flow graph are called *actors*, modeling tasks, while the edges, called *channels*, represent FIFO (first-in-first-out) buffers. They typically model data transfers or control dependencies among actors. The execution of an actor is referred to as a *(actor) firing*, the data items communicated between actors are called *tokens*, and the amounts of tokens produced and consumed in a firing are referred to as *rates*.

Several data flow models with different analyzability potential and expressiveness have been proposed. In general, some models give up some descriptive power in exchange for properties which enable automated analysis. Data flow models may be distinguished according to whether they allow or disallow branching, i.e., actors which perform a decision as to which successor actors take part in the execution. In cases where branching is permitted, for instance, such as Boolean Data Flow Graphs (BDFGs) [42], they are more expressive. BDFGs are Turing complete; no a priori execution schedule can be determined, as this may be dependent on the initial data which leads to undecidability of for example throughput analysis for BDFGs. In case where branching is not permitted, some expressivity is lost in exchange for automated analysis. An example of such a MoC is the model of *Synchronous Data Flow Graphs (SDFGs).*

In this thesis, we focus on SDFGs. SDFGs have been traditionally used for modeling of Digital Signal Processing (DSP) applications [40, 57]. Due to structural similarity between DSP and multimedia applications, SDFGs provide a good degree of expressiveness for modeling of multimedia applications as well [63, 61, 53]. Besides, SDFGs have a lot of analysis potential for measuring exact performance metrics which are very important in evaluating the worst-case behavior of systems. This combination of expressivity and analysis potential makes SDFGs very interesting in the domain of multimedia applications for embedded systems.

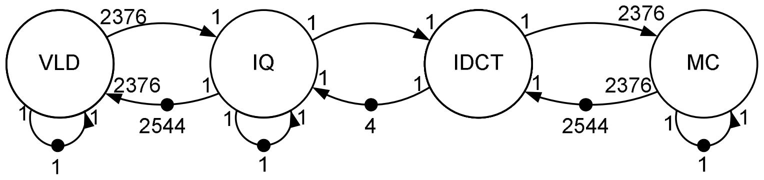


Figure 1.1: The SDFG model of an H.263 decoder

Figure 1.1 shows the SDFG model of an H.263 decoder. H.263 is a video codec standard for a low-bitrate compressed format for videoconferencing. The decoder consists of four actors VLD, IQ, IDCT and MC. Every of the four actors performs part of the frame decoding. The frame decoding starts in the actor VLD (variable length decoding) and a complete frame is decoded when the data is processed by actor MC (motion compensation). Associated with the source and destination of each channel edge are the rates which are determined by the numbers written next to edges. Communicated data and control signals are modeled as tokens, denoted with a black dot and an attached number defining the number of tokens present in the channel. Channel capacities are unbounded, i.e., channels can contain arbitrarily many tokens. Channel capacity limitations need to be modeled explicitly. In the example of Figure 1.1, the partially decoded data is communicated via the channels at the top (left-to-right). The channels at the bottom (right-to-left) model the storage-space constraints on the top edges. For example, the buffer size between VLD and IQ is 2544 tokens. In this case, each token is one block of data which is in turn 64 pixels. Data that must be preserved between subsequent firings of an actor is modeled with an initial token on the self-loop channels (channels with the same source and destination) of the actors.

Actors are typically annotated with execution times to make SDFGs amenable to timing analysis. The execution of an SDFG is defined in terms of its actor firings. An actor is enabled when there are sufficiently many (at least as many as indicated by the rates) tokens on all of its input channels. An enabled actor can start its firing and by doing so the number of tokens on each input channel gets reduced by the number indicated by the rate of that channel. The firing of an actor is atomic and cannot be interrupted. The duration of the firing of an actor is determined by its execution time. In case of untimed SDFGs the execution times of all actors will be considered equal to one. By finishing of the firing of an actor, the number of tokens of each output channel of the actor is increased by the rate of the output channel. This example shows that SDFGs are sufficiently expressive for streaming multimedia applications, which are typically similar to the H.263 decoder. In the next section, the SDFG analysis potential together with the limitations of the traditional methods are discussed.

1.4 Problem Statement

In the previous section, we argued that SDFGs are a good means of modeling multimedia applications as they combine good levels of expressivity and analyzability. Although SDFGs are amenable to many analysis techniques, not many techniques exist that are practically feasible for multimedia applications. Mainly, analysis techniques for SDFGs are categorized into two main groups. The first group, which are exact algorithms, do not often directly work on SDFGs (e.g. [8, 41, 16]), requiring a potentially costly conversion, which makes them unpractical. The second group consists of heuristics and approximation algorithms (e.g. [70, 1]) which are either not precise enough or do not provide precise bounds for the errors made. In the remainder of the section, we explain in more detail the analysis techniques required for checking some crucial properties as well as calculating the performance metrics of SDFGs. These properties together with the performance metrics required for realizing predictable designs are explained.

An inherent property of multimedia applications is that they process very large (virtually arbitrarily long) streams of data. Therefore, SDFGs which deadlock or cannot execute indefinitely are considered faulty. Also in the definition of SDFGs, channel buffers are unlimited; however, only SDFGs which use limited amounts of channel buffers can be implemented in practice. Therefore, we need a method to be able to check the sanity of constructed SDFGs, i.e., whether they can run indefinitely using a finite amount of memory. This thesis develops such sanity checks.

The most prominent metric which has been studied extensively for SDFGs in the literature is throughput [16, 56, 60], i.e., the average number of actor firings per time unit. All throughput calculation methods use analysis techniques which work on a subclass of SDFGs, which is called Homogeneous SDFGs (HSDFG, for short). All rates of an HSDFG are one. Every arbitrary SDFG can be converted to an equivalent HSDFG [41] which makes these methods applicable to SDFGs as well. The problem with the conversion is that it very often results in a dramatic increase in the size of the graph [51]. Therefore, the analysis of the huge graph resulting from the conversion is very time consuming. This thesis develops a throughput analysis technique that avoids the conversion to an HSDFG.

Another issue with throughput analysis techniques for multimedia applications is that actor execution times are usually worst-case estimates of the real execution time of the actor. Therefore, because of the dynamic behavior of software (e.g. data dependent execution) the estimation of software execution times is often not tight. To achieve higher levels of precision in throughput prediction, sometimes, the estimations of actor execution times may change during design-space exploration at design time. Also at run-time, a system may need to reconfigure itself because of various reasons like when an application starts its execution at the same platform. In all throughput calculation methods, changing the execution time of a single actor implies the need for the total recalculation of the throughput.

Although throughput calculation is relatively fast, in some cases huge numbers of different throughput calculations are required for design-space exploration. At run time, for reconfiguration purposes, very limited time and resources are available for calculating the throughput. This thesis investigates parametric throughput analysis techniques. By assuming parametric execution times for actors, we can assume a range of values for execution times instead of only fixed numbers. In this way, the throughput of an SDFG can be specified in terms of a function of the parameters. As a result, a throughput recalculation will be only the evaluation of the function for the new execution times. Then, the throughput calculation becomes very fast using only limited resources.

Furthermore, although throughput is a salient performance metric for multimedia applications, certain timing features of such applications cannot be specified using only throughput. Other performance metrics like latency are also required to specify for example the time difference between executions of different parts of application. A formal definition of latency was only defined for HSDFGs in the literature. Consequential to the lack of a formal definition of latency for SDFGs, no analysis technique for latency calculation of SDFGs were provided. This thesis develops latency analysis for SDFGs and studies the relation with throughput analysis.

1.5 Bibliography

Synchronous data flow graphs are essentially *Computation Graphs*. Computation graphs were first introduced by Karp and Miller in 1966 [38]. Their work concentrates on fundamental properties like determinacy, stating that any admissible execution yields the same ultimate result or termination (deadlock) conditions. A large part of their analysis techniques is dedicated to terminating graphs, and therefore, not directly applicable to multimedia applications. Computation graphs are further explored by Reiter [56]. The term synchronous data flow graphs was first introduced by Lee and Messerschmitt in [43] where they concentrated on the properties of the model related to digital signal processing (DSP) applications. One of the main advantages of SDFGs over other models of computations is that the buffer sizes required to implement channels in SDFGs can be determined at compile time [24, 64]; consequently, static allocations for buffers become possible avoiding the overhead of dynamic memory allocation.

There are interesting similarities between SDFGs and Petri nets [48]. In particular, there is a straightforward translation from SDFGs to a subclass of Petri nets, called weighted Marked Graphs and vice versa, where actors are transitions, and channels are places. Marked Graphs, also called T-Graphs, are known to be the subclass of Petri nets that is most amenable to rigorous analysis [12, 14].

SDFGs have been used for modeling DSP applications (e.g. [40, 57]). Also, in recent years, they have been used in many publications to model multimedia streaming applications [63, 61, 53]. Until recently, analysis techniques for the various mentioned models of computation have not been adapted to and targeted to the needs for modern embedded multimedia systems.

1.6 Contributions

This thesis makes several contributions to the state-of-the-art of timing analysis techniques for SDFGs.

- Liveness refers to the fact that an SDFG can execute indefinitely; boundedness refers to the fact that memory needs are finite. Liveness and boundedness of SDFGs are formally defined in this thesis and necessary and sufficient conditions are provided for checking whether an SDFG is live and bounded. Three useful interpretations of boundedness are discussed, and these conditions are supported with algorithms to perform the checks (Chapter 3). This work has been published in [27, 28].

- A new approach for throughput calculation of SDFGs based on state-space exploration is proposed. This approach, unlike all other existing algorithms, works directly on SDFGs avoiding the conversion to HSDFGs (Chapter 4). The method turns out to be fast in practice, and shows much less variation in execution time than the traditional methods. An earlier version of this work has been published in [29].

- Three different methods for parametric throughput analysis of SDFGs with parameters as execution times are presented and compared. The throughput is given as one over a linear function of parameters (Chapter 5). This work was published in [26, 25].

- The latency definition known from HSDFGs is extended to arbitrary SDFGs. A class of scheduling algorithms is proposed which results in the minimum achievable latency. A heuristic algorithm for minimizing latency under a throughput constraint is also given (Chapter 6). An earlier version of this work has been published in [30, 31].

1.7 Thesis Overview

This thesis is organized as follows. The next chapter discusses the preliminary definitions of SDFGs. The formal definition of actor throughput is also given in this chapter. Chapter 3 presents the necessary and sufficient conditions characterizing when SDFGs are live and bounded. Three definitions of boundedness are discussed in this chapter. All conditions are supported with decision algorithms. A new approach is presented for throughput calculation of SDFGs in Chapter 4. Chapter 5 discusses parametric throughput analysis of SDFGs in which execution times are parameters. In Chapter 6 the definition of latency is generalized to arbitrary SDFGs, a scheduling scheme is proposed for minimizing the latency. Furthermore, a heuristic algorithm is presented for obtaining minimum latency under certain throughput constraints. Chapter 7 concludes this thesis and gives recommendations for future work.

Chapter 2

Preliminaries

2.1 Overview

This chapter formally defines synchronous data flow graphs (Section 2.2) and a timed variant of them (Section 2.3). The static structural properties of SDFGs are discussed in Section 2.4. An operational semantics for SDFGs that formalizes their execution is given in Section 2.5. Dynamic behavioral properties of SDFGs are discussed in this section as well. Homogeneous SDF is a subset of SDF and traditionally most analysis techniques like throughput or latency analysis have been defined based on this subset. HSDF and the relation between SDFGs and HSDFGs are explained in Section 2.6. Finally, Section 2.7 summarizes.

2.2 Synchronous Data Flow Graphs

Let $\mathbb{N}$ and $\mathbb{R}$ denote the (non-negative) natural numbers (including 0), and real numbers respectively. $\mathbb{N}^+$ and $\mathbb{R}^+$ denote the set of positive natural and real numbers (excluding 0). We also denote the set of non-negative natural and real numbers including ∞ and 0 by $\mathbb{N}^\infty$ and $\mathbb{R}^\infty$ respectively.

Formally, an SDFG is defined as follows. We assume a set $Ports$ of ports, and with each port $p \in Ports$ we associate a positive finite rate $Rate(p) \in \mathbb{N}^+$.

Definition 2.1. (Actor) *An actor a is a tuple (In, Out) consisting of a set $In \subseteq Ports$ of input ports (denoted by $In(a)$), a set $Out \subseteq Ports$ of output ports ($Out(a)$) with $In \cap Out = \varnothing$.*

Definition 2.2. (Synchronous Data Flow Graph (SDFG)) *An SDFG is a tuple (A, C) with a finite set A of actors and a finite set $C \subseteq Ports \times Ports$ of (directed) channels. The source p of every channel (p, q) is an output port of some actor; the destination q is an input port of some actor. All ports of all actors are connected to precisely one channel. The associated actor of each port p*

is denoted by $Act(p)$. *For every actor* $a = (I, O) \in A$, *the set of all channels that are connected to ports in* I (O) *is denoted by* $InC(a) = \{(p, q) \in C \mid q \in In(a)\}$ $(OutC(a) = \{(p, q) \in C \mid p \in Out(a)\})$ *and we address them as* input (output) channels of a. *We call a channel from actor* a *to itself a self-loop channel. The set of all self-loop channels of an actor* a *is denoted by* $SLC(a) = InC(a) \cap OutC(a)$. *The predecessors of* a, $\mathrm{Pred}(a) = \{b \in A \mid OutC(b) \cap InC(a) \neq \varnothing\}$, *are the actors that are the source of a channel of which* a *is the destination and similarly* $\mathrm{Succ}(a) = \{b \in A \mid InC(b) \cap OutC(a) \neq \varnothing\}$ *are the actors that are the destination of a channel for which* a *is the source.*

The execution of an actor is defined in terms of *firings*. When an actor a starts its firing, it removes $Rate(q)$ tokens from all $(p, q) \in InC(a)$ and produces $Rate(p')$ tokens on every $(p', q') \in OutC(a)$. These rates are therefore also referred to as input resp. output rates, or consumption resp. production rates. The details of SDFG execution are formalized in Section 2.5.

Figure 2.1 shows an example SDFG, consisting of four actors (circles a through d) and six channels (arrows between actors). The number annotations inside actors are explained below. Associated with the source and destination ports of each channel edge are the rates. Channels may contain tokens, the black dots. Channels can contain arbitrarily many tokens. Capacity limitations can be modeled explicitly. For example, in Figure 2.1, the channels from left to right can be interpreted as data connections, transporting data between actors and the edges in the opposite direction with initial tokens, model available buffer space. In this way, the difference between the number of firings of each actor and that of its successors can be controlled, and this leads to a limited number of permitted tokens on the output channels for an actor. Thus, the SDFG of Figure 2.1 can be interpreted as a model of a multimedia application with four tasks, a through d, to be executed iteratively in a pipelined manner. The three channels from left to right correspond to FIFO buffers with limited sizes of 1, 5, and 1, respectively, as modeled by the channels in the opposite direction. The example SDFG is in fact very similar to the SDFG model of the H.263 decoder discussed in Chapter 1.

2.3 Timed Synchronous Data Flow Graphs

The classical SDFG model is untimed. In fact, it assumes unit time [43] for all actor execution times. However, there is a natural extension to SDFGs in which a fixed execution time is associated with each actor [60]. This extension makes the model amenable to timing analysis such as throughput or latency analysis. Each actor models a task or an operation and its execution time captures the amount of time the execution of that task may take. In reality, actor execution times may vary during the execution, and our choice of constant execution times for actors are for worst-case or best-case analysis of applications.

Definition 2.3. (Execution Time) *An execution time models the execution*

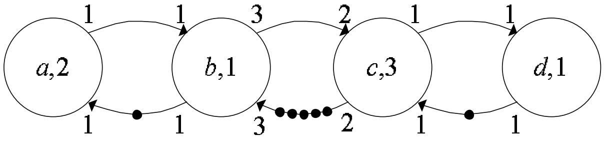


Figure 2.1: An example SDFG

duration of actors of an SDFG. In an SDFG (A, C)*, the execution time is a function* $E : A \mapsto \mathbb{R}^{\infty}$ *that assigns to each actor the amount of time it takes to fire. For* $a \in A$, $E(a)$ *is referred to as the execution time of* a*.*

The infinite execution times are used later on to model deadlocks. Normally, SDFGs do not have infinite actor execution times. Similarly, an execution time of zero is sometimes convenient. Real data transformations typically do not have zero execution times.

Definition 2.4. (Timed SDFG) *A timed SDFG is a triple* (A, C, E) *denoting an SDFG* (A, C) *with execution time* E*.*

The SDFG depicted in Figure 2.1 is in fact a timed SDFG and the numbers in actor nodes denote their execution times.

2.4 Static Properties

SDFGs are also directed (multi-)graphs; therefore, some structural properties similar to those of graphs can be defined here. Note that all structural properties are valid for both SDFGs and timed SDFGs.

Definition 2.5. (Path and Cycle) *A (n undirected) path* p *is a sequence of actors* $a_1 a_2 \dots a_l$ *such that* $a_{i+1} \in \mathrm{Succ}(a_i)$ $(\mathrm{Pred}(a_i) \cup \mathrm{Succ}(a_i))$ *for all* $1 \leq i < l$*. Path* p *is simple iff* $a_i \neq a_j$ *for all* $1 \leq i, j \leq l, i \neq j$*. If in path* p, $a_1 = a_l$ *and* $l \geq 2$*, then* p *is said to be a* cycle*. A* simple cycle *is a cycle* $p = a_1 a_2 \dots a_l$ *such that* $a_1 a_2 \dots a_{l-1}$ *is simple.*

Definition 2.6. (Connected SDFG) *SDFG* (A, C) *is said to be* connected *iff an undirected path exists between all pairs of actors.*

We assume all SDFGs are connected; SDFGs which are not connected consist of separate, completely independent graphs, which can be analyzed separately. A well-known stronger form of connectivity is given by the following definition.

Definition 2.7. (Strongly Connected SDFG) *An SDFG is* strongly connected *iff there exists a directed path from any actor to any other actor. Any subgraph of an SDFG which is strongly connected is called a strongly connected component (SCC, for short). An SCC* κ *is maximal iff there is no SCC* κ' *where* κ *is a strict subgraph of* κ'*.*

The SDFG of Figure 2.1 consists of one maximal strongly connected component as there is a path between any two actors.

Not all SDFGs are meaningful. Inappropriate rates can lead to undesirable effects. If, for example, in the SDFG of Figure 2.1, the input rate of actor b of the c-b channel is changed from 3 to 4, this would result in a guaranteed deadlock after only a few actor firings (2 times a, and all other actors once); if this rate is set to 2, it would result in an unbounded increase of tokens in the channel from b to c. There is a simple property, called consistency, of SDFGs that is necessary to avoid these kinds of effects [43], although it does not guarantee absence of deadlocks. Consistency is a structural property of SDFGs which concerns the correspondence between production and consumption rates.

Definition 2.8. (Consistent SDFG, repetition vector) *A* repetition vector *q of an SDFG (A, C) is a function $A \rightarrow \mathbb{N}$ such that for each channel $(o, i) \in C$ from actor $a \in A$ to $b \in A$, $Rate(o) \cdot q(a) = Rate(i) \cdot q(b)$. A repetition vector q is called non-trivial if and only if $q(a) > 0$ for all $a \in A$.*

An SDFG is called consistent *iff it has a non-trivial repetition vector. For a consistent graph, there is a unique smallest non-trivial repetition vector, which is designated as* the *repetition vector of the SDFG.*

The repetition vector of the SDFG of Figure 1 is $\{(a, 2), (b, 2)(c, 3)(d, 3)\}$ (in vector notation: $[2\ 2\ 3\ 3]^T$). The equations $Rate(o) \cdot q(a) = Rate(i) \cdot q(b)$ are called the *balance equations*. The solution to these equations determines how many times each actor should fire till all the tokens produced by the firings of actors get consumed by some other actors. Therefore, from these equations, it follows that firing all actors in an SDFG precisely as often as specified by a repetition vector has no net effect on the distribution of tokens over all channels. Consistency can be verified in linear time (a linear function of the number of channels in the graph) (See e.g. [43]).

Definition 2.9. (Iteration) *Assume SDFG (A, C) has repetition vector q. An* iteration *is a collection of actor firings such that for each $a \in A$, the collection contains $q(a)$ firings of a.*

2.5 Dynamic Properties

We define the behavior (operational semantics) of a timed SDFG formally in terms of a labeled transition system following [24]. For this, we need appropriate notions of states and of transitions. The behavior of untimed SDFGs can be easily deduced from the timed version.

The behavior of an SDFG consists of firings of its actors during which they consume input data and produce output data. By repeated firings, actors process streams of data. A firing is enabled by the presence of sufficient tokens on all of its input channels. An actor consumes its required input tokens at the start of its firing, and output is produced at the end of that firing. Channels have unbounded capacity, which means that sufficient space is always available. Since we are interested in timing analysis, and not, for example, in functional analysis, we abstract from the actual data that is being communicated.

To capture the timed behavior of an SDFG, we need to keep track of the distribution of tokens over the channels, the start and end of actor firings, and the progress of time. For distributions of tokens on channels, we define the following concept.

Definition 2.10. (CHANNEL QUANTITY) *A* channel quantity *on the set C of channels is a mapping $\delta : C \rightarrow \mathbb{N}$. If δ_1 is a channel quantity on C_1 and δ_2 is a channel quantity on C_2 with $C_1 \subseteq C_2$, we write $\delta_1 \preceq \delta_2$ if and only if for every $c \in C_1$, $\delta_1(c) \leq \delta_2(c)$. $\delta_1 + \delta_2$ and $\delta_1 - \delta_2$ are defined by pointwise addition resp. subtraction of δ_1 and δ_2 resp. δ_2 from δ_1; $\delta_1 - \delta_2$ is only defined if $\delta_2 \preceq \delta_1$.*

The amount of tokens read at the beginning of a firing of some actor a can be described by channel quantity $Rd(a) = \{((p, q), Rate(q)) \mid (p, q) \in InC(a)\}$, produced tokens by channel quantity $Wr(a) = \{((p, q), Rate(p)) \mid (p, q) \in OutC(a)\}$.

Definition 2.11. (TIMED STATE) *The state of a timed SDFG (A, C, E) is a pair (γ, υ). γ is a channel quantity, referred to as a channel state, which associates with each channel the amount of tokens present in that channel in that state. To keep track of time progress, an actor status $\upsilon : A \rightarrow \mathbb{N}^{\mathbb{R}^{\infty}}$ associates with each actor $a \in A$ a multiset of numbers representing the remaining durations of different firings of a. Each timed SDFG has an* initial timed state *which is given by some initial token distribution γ_0, denoting the number of tokens that are initially stored in the channels and $\upsilon_0 = \{(a, \{\}) \mid a \in A\}$ (with $\{\}$ denoting the empty multiset).*

In case of untimed SDFGs, states only consist of a channel state γ with the initial state γ_0.

By using a multiset of numbers to keep track of actor progress instead of a single number, multiple simultaneous firings of the same actor (auto-concurrency) are explicitly allowed. This is in line with the standard semantics for SDFGs [43]. If desirable, auto-concurrency can be excluded or limited by adding self-loops to actors each with a number of initial tokens equal to the desired maximal number of concurrent actor firings.

The dynamic behavior of the timed SDFG is described by transitions that can be of any of three forms: start of actor firing, end of firing, or time progress.

Definition 2.12. (TRANSITIONS) *A transition of a timed SDFG (A, C, E) from state (γ_1, υ_1) to state (γ_2, υ_2) is denoted by $(\gamma_1, \upsilon_1) \xrightarrow{\alpha} (\gamma_2, \upsilon_2)$ where label $\alpha \in (A \times \{start, end\}) \cup (\{clk\} \times \mathbb{R}^{+})$ denotes the type of the transition.*

- *Label $\alpha = (a, start)$ corresponds to the firing start of actor a. This transition is enabled if $Rd(a) \preceq \gamma_1$ and results in $\gamma_2 = \gamma_1 - Rd(a)$, $\upsilon_2 = \upsilon_1[a \mapsto$*

$\upsilon_1(a) \uplus \{E(a)\}]$, *i.e.,* υ_1 *with the value for* a *replaced by* $\upsilon_1(a) \uplus \{E(a)\}$ *(where* $\uplus$ *denotes multiset union).*

- *Label* $\alpha = (a, end)$ *corresponds to the firing end of* a*. This transition is enabled if* $0 \in \upsilon_1(a)$ *and results in* $\gamma_2 = \gamma_1 + Wr(a)$ *and* $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \backslash \{0\}]$ *(where* $\backslash$ *denotes multiset difference).*
- *Label* $\alpha = (clk, l)$ *denotes a clock transition and* $l \in \mathbb{R}^+$ *specifies the length of the clock transition.* l *is the minimum remaining execution time of all the ongoing actor firings. More precisely,* $l = \min\{r \in \mathbb{R}^+ \mid a \in A, r \in \upsilon(a)\}$.

  A clock transition is enabled only if no end transition is enabled. Also, at most one clock transition is enabled and results in $\gamma_2 = \gamma_1$, $\upsilon_2 = \{(a, \upsilon_1(a) \ominus l) \mid a \in A\}$ *with* $\upsilon_1(a) \ominus l$ *a multiset of real numbers containing the elements of* $\upsilon_1(a)$ *(which are all positive, and at least* l*) reduced by* l*.*

Definition 2.13. (Execution and Maximal Execution) *An* execution *of a timed SDFG is an alternating sequence of states and transitions* $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ *starting from the initial state* s_0 *of the graph, such that for all* $n \geq 0$, $s_n \xrightarrow{\alpha_n} s_{n+1}$*. An execution is maximal if and only if it is finite with no transitions enabled in the final state, or if it is infinite. The execution of an untimed SDFG is similar to that of a timed SDFG except that it consists only of channel states and lacks the clock transitions.*

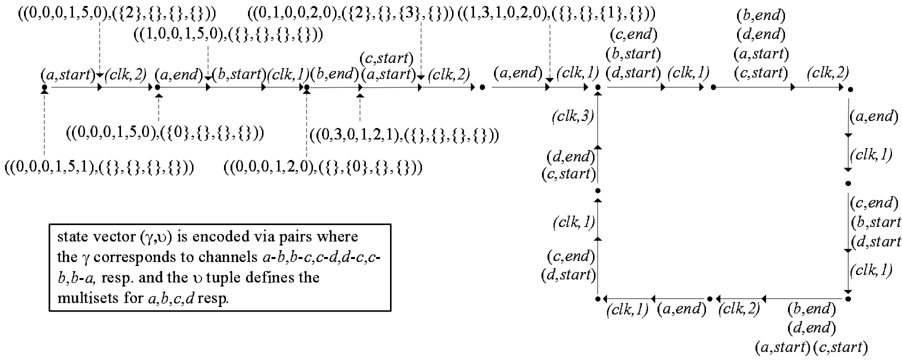


Figure 2.2: The self-timed execution of our running example

Figure 2.2 illustrates an execution of the example SDFG of Figure 2.1. Every state (γ, υ) is encoded via pairs where γ corresponds to channels *a*-*b*, *b*-*c*, *c*-*d*, *d*-*c*, *c*-*b*, *b*-*a*, and υ defines the multisets for a, b, c, and d respectively. The execution starts with the initial state $((0, 0, 0, 1, 5, 1), (\{\}, \{\}, \{\}, \{\}))$ where no actor is firing and the token distribution is determined by the initial tokens depicted in the graph. The only enabled actor at this point is a. When a starts its firing, the state becomes $((0, 0, 0, 1, 5, 0), (\{2\}, \{\}, \{\}, \{\}))$ where the token on the input channel of a is consumed and its execution time has been added to $\upsilon(a)$. No other actor can fire before a finishes its firing. Since the only element in any of the υ is 2, time progresses for 2 time units changing the state to $((0,0,0,1,5,0),(\{0\},\{\},\{\},\{\}))$. Then, the firing of a ends which changes the state to $((1,0,0,1,5,0),(\{\},\{\},\{\},\{\}))$ enabling actor b. Similarly actor b starts its firing and time progresses for 1 time unit and consequently the state changes to $((0,0,0,1,2,0),(\{\},\{0\},\{\},\{\}))$. By the finishing of actor b the state changes to $((0,3,0,1,2,1),(\{\},\{\},\{\},\{\}))$ which enables both a and c and they start their firings simultaneously. This creates a new state $((0,1,0,0,2,0),(\{2\},\{\},\{3\},\{\}))$. Again time progresses for 2 time units as it is the smallest value among all elements in υ, which leads to the end of firing of actor a and state $((1,3,1,0,2,0),$ $(\{\},\{\},\{1\},\{\}))$. This process continues in the same manner.

The order of start and end transitions between two clock transitions is often irrelevant; therefore, sometimes start and end transitions are conveniently omitted in the notation of an execution and only the states immediately after clock steps are shown. Note that these states are always the same, independent of the order of state and transitions preceding a clock transition. The steps in such an execution are referred to as *macro steps*. Hence, the execution of a (timed) SDFG is also denoted as: $\sigma = S_0, S_1, \ldots$ where the S_i are states obtained from macro steps.

Not all SDFGs are considered to be useful in practice. One normally seeks a system that is live or at least deadlock-free, as defined below.

Definition 2.14. (DEADLOCK AND LIVENESS) *An SDFG has a deadlock if and only if it has a maximal execution of finite length. An SDFG is live if and only if it has an execution in which all actors fire infinitely often.*

It is known [38] that the execution of an SDFG is determinate, which means that the order of execution does not affect the states that can eventually be reached. Thus, if one execution of an SDFG deadlocks (is maximal and finite), then all executions deadlock. Absence of deadlock does not imply liveness. It is possible that only infinite executions exist in which not all actors fire infinitely often. The SDFG in Figure 2.1 is live which can be seen from Figure 2.2.

The maximal throughput (a precise definition of throughput is given in Chapter 4) of an SDFG is known to be obtained from a specific type of execution, namely self-timed execution [60], which means that actors fire as soon as they are enabled.

Definition 2.15. (SELF-TIMED EXECUTION) *An execution is* self-timed *if and only if clock transitions only occur when no start transitions are enabled.*

Based on the above observation about macro steps, it can be seen that self-timed SDFG behavior is deterministic in the sense that all the states immediately before and after clock transitions are completely determined and independent of the selected execution. Thus, it is meaningful to refer to *the* self-timed execution of an SDFG.

Figure 2.2 in fact shows the self-timed execution of our running example. The self-timed execution of the graph of Figure 2.1 consists of a periodic phase preceded by a so-called transient phase. We show in Chapter 4 that if actor execution times are rational numbers, then the behavior of the self-timed execution of strongly connected graphs is always eventually periodic (similar to the figure).

In the following, we define some notations related to the execution of an SDFG, which are used later for the definition of performance metrics like throughput and latency. At this point, we already give a definition of *actor* throughput.

Definition 2.16. (Firing Functions) *Given a timed SDFG $G = (A, C, E)$ and an execution σ, let $S^{\sigma}_{a,k}$ ($F^{\sigma}_{a,k}$) denote the start (end) time of the k-th firing with $k \in I\!N$ of any actor $a \in A$ in execution σ, i.e., the sum of the length of clock transitions up to the k-th appearance of $\overset{(a,start)}{\rightarrow}$ ($\overset{(a,end)}{\rightarrow}$) in σ. If σ is clear from the context, we write $S_{a,k}$ and $F_{a,k}$ for denoting* start *and* end firing functions*.*

Opposite to the start firing function, $N^{\sigma}_{a,t}$ denotes the number of occurrences of the transition $\overset{(a,start)}{\rightarrow}$ up to time t.

$$N^{\sigma}_{a,t} = \max\{k \in I\!N \mid S^{\sigma}_{a,k} \leq t\}.$$

Using the above notations, now we can define the actor throughput of an SDFG.

Definition 2.17. (Actor throughput) *The throughput of an actor a for execution σ of an SDFG is defined as the average number of firings of a per time unit in σ. Since executions can be infinite, this average is defined as the following limit:*

$$Th(\sigma, a) = \lim_{t \to \infty} \frac{N^{\sigma}_{a,t}}{t}.$$

It is easy to see that when the execution includes an infinite number of start transitions, then this is equal to

$$Th(\sigma, a) = \lim_{k \to \infty} \frac{k}{S^{\sigma}_{a,k}}.$$

Note that this definition expresses the throughput of an SDFG actor for a particular execution σ. With $Th(a)$, the actor throughput of a, we denote the throughput of actor a of the self-timed execution, which is known to be maximal among all executions.

2.6 Homogeneous SDF

SDFGs in which all rates associated to ports equal 1 are called *Homogeneous Synchronous Data Flow Graphs* (HSDFGs, [43]). As all rates are 1, any HSDFG is consistent and the repetition vector for an HSDFG associates 1 to all actors.

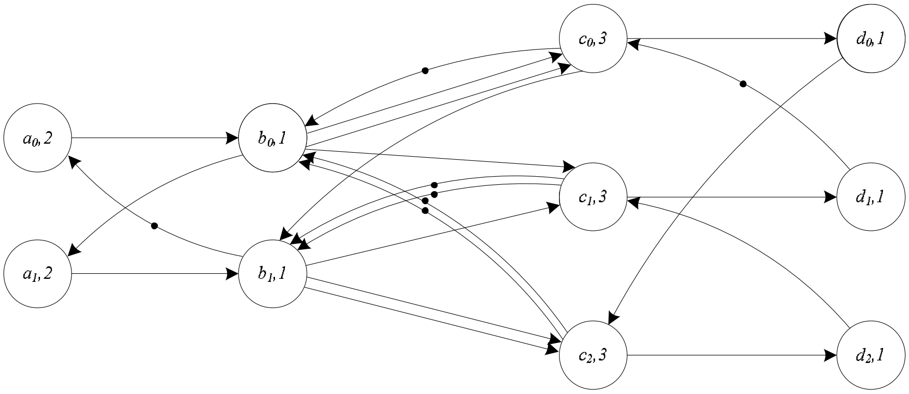


Figure 2.3: The HSDFG equivalent to our running example SDFG.

Every (timed) SDFG $G = (A, C, E)$ can be converted to an equivalent HSDFG $G_H = (A_H, C_H, E_H)$ ([43, 60]) which mimics the execution of G. This conversion is done by using the conversion algorithm in [60, Section 3.8]. Figure 2.3 shows the equivalent HSDFG of the SDFG of Figure 2.1. In the conversion, every actor is copied as many times as its entry in the repetition vector. For example, actor b has two copies b_0 and b_1. Every copy receives as many input (output) ports as the sum of the rates of its input (output) ports in the original SDFG. We can see in the figure that, for example, all copies of actor b have four output channels and four input channels. Every channel (p, q) is translated into $\mathit{Rate}(p) \cdot \mathit{Rate}(q)$ channels connecting the copies of source and destination actors. The channel between b and c with rates of 3 and 2 has been replaced by $3 \times 2 = 6$ channels. The total number of tokens remains the same but the tokens of an SDFG channel get distributed evenly over all the replacement channels between the copies of the source and the destination of the channel. Initial tokens also determine the source and destination ports to which channels should connect as they are depicted in Figure 2.3.

The equivalence notion between SDFGs and HSDFGs means that there exists a bijection relation between the SDFG and HSDFG actor firings and it can be made precise as follows.

For every actor $a \in A$ of an SDFG $G = (A, C, E)$, with repetition vector q, the conversion algorithm creates $q(a)$ copies, $a_0 \ldots a_{q(a)-1}$, all with execution time $E(a)$. The correspondence between an SDFG and its equivalent HSDFG is as follows: the k-th firing of a_r in the HSDFG corresponds to firing $k \cdot q(a) + r$ of a in the original SDFG. It can be shown ([33]) that for any execution σ of the SDFG, there is an execution σ_H of the equivalent HSDFG such that, for the firing start times of a and its copies, and for all $r, k \in \mathbb{N}$ with $0 \leq r < q(a)$,

$$S^{\sigma}_{a,k \cdot q(a)+r} = S^{\sigma_H}_{a_r,k} \tag{2.1}$$

Note that the $k \cdot q(a) + r$-th firing of actor $a \in A$, is also the r-th firing of a in iteration k of G. Also, since actor a and all its copies in the HSDFG have the same execution time, there exists a similar equation for the end times of actor firings in the SDFG and the equivalent HSDFG actor firings.

$$F^{\sigma}_{a,k \cdot q(a)+r} = F^{\sigma_H}_{a_r,k} \tag{2.2}$$

2.7 Summary

This chapter formalizes the SDFG model and it extends the model to take time into account. Different properties of SDFGs are discussed in two categories of static and dynamic properties. Static properties explain the structural properties of SDFGs. Dynamic properties are defined formally in terms of a labeled transition system (operational semantics). Most traditional timing analysis techniques have been defined on a special type of SDFG called an HSDFG. The relation between SDFGs and their equivalent HSDFGs is also discussed in this chapter.

Chapter 3

Liveness and Boundedness

3.1 Overview

As explained in Chapter 2, an execution of an SDFG is a sequence of actor firings which respects data dependencies. As long as these dependencies are satisfied, the exact order of actor firings is not determined. Consequently, several executions exist for an SDFG. Because of the usage of SDFGs for modelling streaming applications, typically, only those SDFGs which have executions in which all actors are fired infinitely often are of interest. This property of SDFGs is called liveness. Furthermore, only executions that require a finite amount of storage for the channels are of interest. This chapter formally studies this property, called boundedness, in combination with liveness.

The chapter investigates two common interpretations, namely 'normal' boundedness which requires that there exists a bounded execution of an SDFG, and strict boundedness which is whether all executions are bounded. We prove necessary and sufficient conditions guaranteeing that an SDFG is live and (strictly) bounded. For strict boundedness, these conditions follow immediately from a similar result known for Petri nets.

A natural way of scheduling applications on multiprocessors is self-timed as no extra control mechanism is required for scheduling the processors except the readiness of the necessary data for each processor. Self-timed schedule is also desirable because it achieves the maximum attainable throughput of an SDFG. Therefore, it raises an interesting question of whether the self-timed execution is feasible in practice using a finite amount of memory for channels. To answer this question, a new notion of boundedness, namely self-timed boundedness is introduced. This notion requires that self-timed execution of SDFGs is bounded. Necessary and sufficient conditions for the liveness and self-timed boundedness of SDFGs are proved. In this chapter, an algorithm is proposed that determines the liveness and self-timed boundedness of an SDFG.

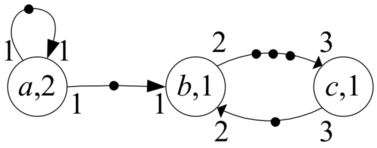


Figure 3.1: An example timed SDFG G_{ex}.

The rest of this chapter is organized as follows. Section 3.2 formally introduces different definitions of boundedness for SDFGs to allow studying liveness and boundedness in a rigorous way. Sections 3.3 and 3.4 present necessary and sufficient conditions for liveness and (strict) boundedness plus algorithms for verifying these conditions. Section 3.5 identifies conditions for self-timed boundedness of SDFGs and presents an algorithm for verifying the combination of liveness and this type of boundedness. Section 3.6 discusses related work. Section 3.7 summarizes the conclusions of the chapter. This chapter is based on publication [28].

3.2 Boundedness Definitions

Different useful notions of boundedness can be defined for SDFGs. To enable identifying these forms, we first define boundedness for a given execution.

Definition 3.1. (Bounded Channel and Bounded Execution) *Let* $\gamma_0, \gamma_1, \ldots$ *represent the sequence of channel states of an execution* σ *of a (timed) SDFG. We call a channel ch bounded under* σ *iff there exists some* $B \in \mathbb{N}$ *such that* $\gamma_i(ch) \leq B$ *for all* $i \geq 0$*. If all channels of the SDFG are bounded under* σ *then* σ *is bounded.*

Now, we give a definition for the boundedness of an SDFG which intuitively means that it can be implemented using a finite amount of memory.

Definition 3.2. (Bounded SDFG) *A (timed) SDFG is called* bounded *iff there exists a bounded maximal execution. It is unbounded otherwise.*

A stronger form of boundedness is *strict boundedness.*

Definition 3.3. (Strictly Bounded Channel, Strictly Bounded SDFG) *A channel of a (timed) SDFG G is* strictly bounded *iff it is bounded under all executions of G. A (timed) SDFG is called strictly bounded iff all of its channels are strictly bounded.*

Note that this definition allows that each execution can have a different bound.

Figure 3.1 shows a simple example of SDFG G_{ex} which is consistent with the repetition vector $[3\ 3\ 2]^T$. We use G_{ex} as the running example throughout

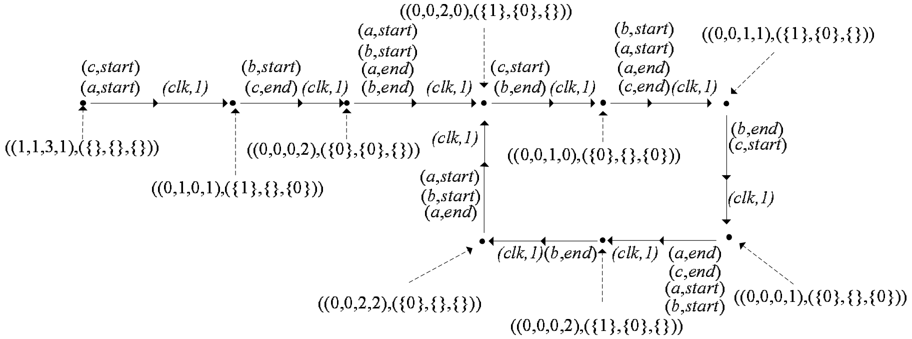


Figure 3.2: Self-timed execution of G_{ex}.

this chapter. G_{ex} is bounded but not strictly bounded because a can be fired indefinitely without firing b and c.

Note that any strictly bounded SDFG is also bounded. We finally define another form of boundedness, which only considers self-timed execution of timed SDFGs.

Definition 3.4. (SELF-TIMED BOUNDED SDFG) *A timed SDFG is* self-timed bounded *iff the self-timed execution is bounded. A channel in a timed SDFG is self-timed bounded iff it is bounded under self-timed execution.*

Figure 3.2 illustrates the self-timed execution of the example SDFG G_{ex} of Figure 3.1. The state contains a channel component with the distribution of tokens over the channels *a*-*a*, *a*-*b*, *b*-*c*, *c*-*b*, respectively, and a time component as described in Chapter 2.

G_{ex} is self-timed bounded, as Figure 3.2 illustrates. Hence, it is also bounded. In fact, self-timed bounded SDFGs are by definition bounded. They are not necessarily strictly bounded. SDFG G_{ex} is not strictly bounded, which follows for example from the execution that fires actor a infinitely often. It is not difficult to construct bounded SDFGs that are not self-timed bounded. If the execution times of actors b and c in G_{ex} are changed to 3, for example, then the SDFG remains bounded but it is no longer self-timed bounded. This example graph and its variant show that the notion of self-timed boundedness does not coincide with other notions of boundedness. Given the importance of self-timed execution, it is worth investigating this notion of boundedness in some detail.

Figure 3.3 shows the three-way relations between different notions of boundedness, liveness and deadlock-freeness together with consistency. In fact, only SDFGs which can be categorized in the fraction in dark gray are of interest or considered well-constructed SDFGs. The light gray fractions are empty; the white fractions are not empty. The correctness of this diagram follows from the definitions and examples given so far, as well as from results proven in the remainder of this chapter.

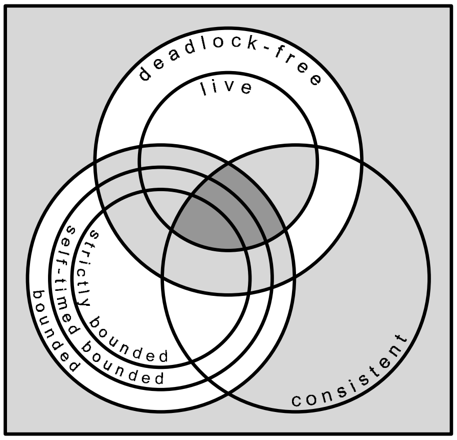


Figure 3.3: Liveness and boundedness diagram.

3.3 Boundedness

In this section, we study necessary and sufficient conditions under which an SDFG is live and bounded.

Theorem 3.1. *A live SDFG is bounded iff it is consistent.*

Proof. Let G be a live SDFG. The sufficient (if) part: If the graph is consistent, then there exists a non-trivial repetition vector q for G. So, starting from the initial state s_0, if every actor $a \in A$ fires $q(a)$ times, then according to the definition of the repetition vector the channel state of G goes back to s_0. According to [41], an SDFG is live (called deadlock free in [41]) iff it is possible to execute every actor as many times as indicated by its repetition vector entry. As the number of initial tokens, the number of firings and the rates are bounded, therefore the number of produced tokens during such an iteration is limited. So, we conclude that the required memory under these firings is bounded. Therefore, the execution consisting of repeating the same actor firing pattern is bounded.
The necessary (only if) part: If G is live and bounded, then there exists an infinite execution which is bounded. This implies that also an infinite and bounded sequential execution σ exists in which no two actor firings are simultaneously active. Execution σ visits some state in which no actors are firing and with channel state γ repeatedly because the execution is infinite and a bounded SDFG can only have a finite number of different token distributions. Let $\gamma_0, \gamma_1, \ldots$ be the sequence of channel states resulting from σ. Let $F^{\#}_{a,k}$ for some actor a denote the number of (a, end) transitions that have occurred up-to and including channel state γ_k; Note that by assumption $F^{\#}_{a,k}$ equals the number of (a, start) transitions that occurred up-to that point.

Suppose $\gamma_n = \gamma_{n'}$. We can calculate the number of tokens on every channel ch from a to b with production and consumption rates of p and c respectively in any state with channel state γ_n in which no actors are firing by the following expression

$$\gamma_k(ch) = \gamma_0(ch) + pF^{\#}_{a,k} - cF^{\#}_{b,k}.$$

Assume without loss of generality that $n' > n$ and that there is at least one actor firing between them. Since G is connected, it is impossible to return to the same state without having fired every actor at least once. Therefore, $F^{\#}_{a,n'} > F^{\#}_{a,n}$ and $F^{\#}_{b,n'} > F^{\#}_{b,n}$. Since we have $\gamma_n = \gamma_{n'}$, it follows that

$$(F^{\#}_{a,n'} - F^{\#}_{a,n})p = (F^{\#}_{b,n'} - F^{\#}_{b,n})c.$$

Hence, if we take for all $a \in A$, $q(a) = F^{\#}_{a,n'} - F^{\#}_{a,n}$ then q is a non-trivial solution for the balance equations, which means G is consistent. $\square$

Theorem 3.1 states the consistency of an SDFG as a necessary and sufficient condition for boundedness of live SDFGs. If a subgraph of an SDFG deadlocks (which means that the SDFG is not live) then the consistency of an SDFG is not sufficient for boundedness. For example, consider G_{ex} of Figure 3.1 without the initial token in the *c*-*b* channel. Execution times may be ignored. The resulting SDFG is consistent (consistency is independent of the number of tokens) but not bounded, because the SCC of the graph that consists of actors b and c deadlocks after the first firing of both actors. However, actor a can continue its firing, and must do so in any maximal execution, which leads to an unbounded channel between a and b.

According to Theorem 3.1 we cannot have neither consistent and live graphs which are not bounded nor bounded and live graphs which are not consistent. These two fractions are in fact empty and shown with the light gray inside the live circle in Figure 3.3. Similarly, we cannot have consistent SDFGs which are neither deadlock-free nor bounded.

Proposition 3.1. *[68] A strongly connected SDFG is live iff it is deadlock-free.*

The definition of liveness states that a live SDFG has an execution in which all actors fire infinitely often. If a live SDFG is strongly connected, then all actors fire infinitely often in *all maximal* executions.

Lemma 3.1. *If one maximal SCC in an SDFG G deadlocks then either G deadlocks or it is unbounded.*

Proof. If G consists of only one maximal SCC then the lemma follows immediately. In case it consists of multiple SCCs, at least one deadlocked and at least one deadlock-free, then there exists an SCC (possibly a single actor) which is deadlock-free and connected to an actor in a deadlocked SCC (an SCC is said to be deadlocked when it deadlocks in isolation of the rest of the SDFG). This connecting channel must necessarily go from the deadlock-free SCC to the deadlocked SCC. Since in a deadlock-free SCC all actors must necessarily fire infinitely often, this channel must be unbounded. The case where the graph consists of multiple SCCs and all of them deadlocking is trivial. □

This lemma implies that in a deadlock-free and bounded SDFG no maximal SCCs deadlock and so the SDFG is live.

Corollary 3.1. *An SDFG is live and bounded iff it is deadlock-free and bounded.*

As a consequence, we cannot have SDFGs which are deadlock-free and bounded but not live. Therefore, the light gray fraction in Figure 3.3 inside the deadlock-free circle and outside the live circle denotes the lack of existence of these types of SDFGs.

The following theorem follows from Theorem 3.1, Proposition 3.1, Lemma 3.1, and Corollary 3.1.

Theorem 3.2. *An SDFG is live and bounded iff it is consistent and all its maximal SCCs are deadlock-free.*

Proof. For the necessary (only if) part, note that Theorem 3.1 states that an SDFG which is live and bounded is also consistent. Liveness and boundedness together with Lemma 3.1 show that all maximal SCCs must be deadlock-free. For the sufficient (if) part, observe that the fact that all maximal SCCs are deadlock-free implies liveness of the SDFG, because the maximal SCCs of the SDFG without input channels from other maximal SCCs can, by Proposition 3.1, always continue feeding tokens into the SDFG, which again by Proposition 3.1 implies all actors in all maximal SCCs can fire infinitely often and hence the SDFG is live. Theorem 3.1 then implies that the SDFG is also bounded. □

The example SDFG G_{ex} is live and bounded because it is consistent and all its maximal SCCs are deadlock-free. Next, we give an algorithm to check liveness and boundedness of an SDFG.

Algorithm *isLive&Bounded*(G)
Input: A connected (timed) SDFG G
Output: "live and bounded" or "either deadlock or unbounded"

```
if $G$ is inconsistent
    then return "either deadlock or unbounded"
for each maximal SCC $S$ in $G$
    do if $S$ deadlocks
        then return "either deadlock or unbounded"
return "live and bounded"
```

Algorithm *isLive&Bounded* first checks the consistency of the graph and then verifies the deadlock-freeness of all of its maximal SCCs in isolation. If the graph is consistent and all of its maximal SCCs are deadlock-free then the graph is announced live and bounded. Consistency of SDFGs can be verified efficiently as explained in [8]. Maximal SCCs of a graph can also be computed efficiently [15]. Algorithms for detecting deadlock for consistent strongly connected SDFGs that are efficient in practice are given in [29, 41]. The algorithm in [29] is the throughput analysis algorithm discussed also in Chapter 4 of this thesis. An SDFG can be checked for deadlock by a straightforward state-space exploration. Note that it is in this way also straightforward to distinguish deadlock cases from unbounded ones, but as they are both uninteresting, they are not identified in the algorithm.

3.4 Strict Boundedness

This section identifies necessary and sufficient conditions for the liveness and strict boundedness of an SDFG.

Theorem 3.3. *[68, Theorem 4.11] A live SDFG is strictly bounded iff it is consistent and strongly connected.*

This theorem in combination with Proposition 3.1 implies the following theorem.

Theorem 3.4. *An SDFG is live and strictly bounded iff it is deadlock-free, consistent and strongly connected.*

Proof. First we prove the sufficient (if) part. According to Proposition 3.1, a deadlock-free strongly connected SDFG is live. Therefore, according to Theorem 3.3 a deadlock-free, consistent and strongly connected SDFG is live and strictly bounded. The necessary (only if) part follows directly from Theorem 3.3. ☐

```
Algorithm isLive&StrictBounded(G)
Input: A connected (timed) SDFG $G$
Output: “live and strictlyBounded” or “either not strictly bounded or deadlock”
1.  if $G$ is consistent & strongly connected & deadlock free
2.      then return “live and strictly bounded”
3.      else  return “either not strictly bounded or deadlock”
```

So the algorithm for checking liveness and strict boundedness first checks whether the SDFG is strongly connected and consistent, and then whether it is deadlock-free using the algorithms from [8, 15, 29, 41]. The example of Figure 3.1 is not strictly bounded because it is not strongly connected. An execution which only fires a forever fills up channel a-b.

3.5 Self-timed Boundedness

In this section, we investigate the liveness and self-timed boundedness of timed SDFGs. A self-timed execution of a live and self-timed bounded SDFG uses a finite amount of memory and all actors fire infinitely often in such an execution. Necessary and sufficient conditions are given, and an algorithm for checking these conditions. As self-timed boundedness is directly related to the periodic behavior of the self-timed execution, we only allow rational numbers for the execution times of actors in this section. As already mentioned in Chapter 2, and shown in Chapter 4, Proposition 4.1, self-timed execution of an SDFG with rational execution times is periodic, as illustrated in for example Figures 2.1 and 3.1. It is unknown whether an SDFG with real actor execution times exhibits the same periodic behavior. The next subsection defines a concept on which self-timed boundedness heavily depends, namely local throughput of an actor and investigates some basic properties related to self-timed boundedness.

3.5.1 Local Throughput and Basic Self-timed Boundedness Properties

We define the *local* throughput of an actor as the throughput of an actor when it does not need to wait for data from other actors; in other words, the throughput of that actor in the self-timed execution where all non-self-loop input channels are removed.

Definition 3.5. (Local Throughput) *The* local throughput $LTh(a)$ *of an actor a for a self-timed execution with initial channel state γ_0 of a timed SDFG (A, C, E) is defined as*

$$LTh(a) = \begin{cases} 0, & \text{if there is a } ch = (p,q) \text{ in } SLC(a) \\ & \quad \text{such that } Rate(p) < Rate(q) \text{ or } \gamma_0(ch) < Rate(q) \\ \displaystyle\min_{\substack{ch = (p,q) \in SLC(a) \\ with Rate(p) = Rate(q)}} \lfloor \gamma_0(ch) / Rate(q) \rfloor / E(a), & \text{otherwise.} \end{cases}$$

If an actor has a self-loop channel with a lower production rate than consumption rate or insufficient tokens for an initial firing, it deadlocks at some point in time, i.e., its local throughput is zero. Otherwise, the local throughput is determined by the self-loop channels with equal production and consumption rates. If there are no such channels, i.e., there are no self-loop channels or all self-loop channels have a higher production than consumption rate, local throughput is by definition infinite (A min qualification over an empty domain results in ∞).

Consider again SDFG G_{ex} of Figure 3.1. The local throughput of actor a is $\frac{1}{2}$, whereas it is ∞ for b and c. The regular throughput of the three actors, as depicted in Definition 2.17 equals $\frac{3}{6} = \frac{1}{2}$, $\frac{3}{6} = \frac{1}{2}$, and $\frac{2}{6} = \frac{1}{3}$, respectively, as can be seen from the self-timed execution in Figure 3.2.

In the following, some properties for the throughput as well as the relation between boundedness and throughput of timed SDFGs are given which are used later for checking the self-timed boundedness of an SDFG. The following lemma specifies that the throughput of an actor is determined by the throughput of its predecessors and its local throughput.

Lemma 3.2. *The throughput of an actor $b \in A$ of a timed SDFG $G = (A, C, E)$ satisfies the equation*

$$Th(b) = \min\{ \min_{(p,q) \in InC(b) \setminus SLC(b)} \frac{Rate(p)}{Rate(q)} Th(Act(p)), LTh(b).\} \tag{3.1}$$

Proof. It is not difficult to see that the local throughput of an actor is an upper bound for its throughput. So, we prove the theorem for the case

$$\min_{(p,q) \in InC(b) \setminus SLC(b)} (Rate(p)/Rate(q))\, Th(Act(p)) \leq LTh(b).$$

Let channels $ch_i = (p_i, q_i)$ be all input channels of b. Suppose $a_m = Act(p_m)$ is a predecessor of actor b for which $(Rate(p_m)/Rate(q_m))\, Th(a_m)$ is minimal. First, consider the case that $Rate(q_m)\, Th(b) - Rate(p_m)\, Th(a_m) = k > 0$. By substituting the definition of actor throughput we get

$$\lim_{t \to \infty} \frac{Rate(q_m) N_{b,t} - Rate(p_m) N_{a_m,t}}{t} = k.$$

According to the definition of a limit, for any $\epsilon > 0$, there exists a time T where for all $t > T$, we have

$$|\frac{Rate(q_m) N_{b,t} - Rate(p_m) N_{a_m,t}}{t} - k| < \epsilon.$$

If we set ϵ to $k/2$, we can conclude that for $t > T$,

$$Rate(q_m) N_{b,t} - Rate(p_m) N_{a_m,t} > \frac{k}{2} t,$$

which means that regardless of the number of initial tokens on the channel, there exists a T, such that for all $t > T$ the number of produced tokens plus the number of initial tokens is less than the consumed ones, which is impossible. Hence, $Rate(q_m)\, Th(b) - Rate(p_m)\, Th(a_m) \leq 0$.

Second, similarly, we can show that if $Rate(q_m)\, Th(b) - Rate(p_m)\, Th(a_m) < 0$, then there exists a time T, where for times $t > T$, there were enough tokens on ch_m to fire and due to the choice for a_m (minimality of $(Rate(p_m)/Rate(q_m))\, Th(a_m)$) on all other input channels of b, while b did not start a new firing, which contradicts the self-timed execution scheme.

Thus, $Rate(q_m)\, Th(b) - Rate(p_m)\, Th(a_m) = 0$, and $Th(b) = \frac{Rate(p_m)}{Rate(q_m)} Th(a_m) = \min_{(p,q) \in InC(b) \setminus SLC(b)} (Rate(p)/Rate(q))\, Th(Act(p))$ which completes the proof. □

The throughput of actor b of G_{ex}, for example, is $\frac{1}{2}$, because its predecessor a has that throughput, the rates of channel a-b are 1 and its local throughput is ∞.

Corollary 3.2. *If actors* $a, b \in A$ *of an SDFG* G *are connected by a channel* (p, q) *then* $Th(b) \leq (Rate(p)/Rate(q))\,Th(a)$.

After having illustrated the factors that are involved in calculating the throughput of an actor, we now show that the only case that a channel is not self-timed bounded, is when the production of tokens into one channel is larger than the consumption of tokens out of that channel.

Lemma 3.3. *SDFG* (A, C, E) *is self-timed bounded iff* $Th(b) \geq (Rate(p)/Rate(q))\,Th(a)$ *for every channel* $(p, q) \in C$ *connecting* a *to* b.

Proof. We know that there is a time t_p such that for all $t \geq t_p$, the execution of G is in the periodic phase. Let d be the amount of time that one period of a self-timed execution σ of G takes. Then for any actor $a \in A$ and time $t \geq t_p$ we have

$$N_{a,t+d} - N_{a,t} = d\,Th(a).$$

Although the number of firings of one actor in one period is always fixed, the firings of a in one period can be spread over the period. Therefore, we have the following inequality, where $k = \lfloor (t - t_p)/d \rfloor$,

$$kd\,Th(a) \leq N_{a,t} - N_{a,t_p} \leq (k + 1)d\,Th(a). \tag{3.2}$$

Let $\gamma_t(ch)$ be the number of tokens on channel $ch = (p, q) \in C$ from a to b at time t; then $\gamma_t(ch)$ can be bounded as follows

$$\gamma_t(ch) \leq \gamma_0(ch) + Rate(p)N_{a,t} - Rate(q)N_{b,t}. \tag{3.3}$$

Note that $\gamma_t(ch)$ is strictly less than the right-hand side if at time t actor b has active firings. For $t > t_p$, by using Equation (3.2), we have

$$\gamma_t(ch) \leq \gamma_0(ch) + Rate(p)(N_{a,t_p} + (k+1)d\,Th(a)) - Rate(q)(N_{b,t_p} + kd\,Th(b)).$$

Since the only part of the above inequality that depends on k is $kd(Rate(p)\,Th(a) - Rate(q)\,Th(b))$, $\gamma_t(ch)$ is bounded if $Rate(p)\,Th(a) \leq Rate(q)\,Th(b)$.

Using a similar argument, looking at the number of firing starts of actors a and b instead of ends, we can conclude that ch is not bounded if $Rate(p)\,Th(a) > Rate(q)\,Th(b)$. □

The next proposition gives necessary and sufficient conditions for self-timed boundedness of a live strongly connected SDFG.

Proposition 3.2. *A live and strongly connected SDFG* G *is self-timed bounded iff it is consistent.*

Proof. The sufficient (if) part can be deduced directly from Theorem 3.3 as strict boundedness ensures self-timed boundedness. For the necessary (only if) part, the same argument as used in the proof of Theorem 3.1 is valid. ☐

Lemmas 3.4 and 3.5 and Proposition 3.3 prove some useful properties about the relation between the throughput of different actors of the same graph. Lemma 3.4, which follows immediately from Corollary 3.2 and Lemma 3.3, shows the relation between producer and consumer actors of an arbitrary self-timed bounded channel. Lemma 3.5 shows the relation between the actor throughputs for any two actors in an SCC of an SDFG. Proposition 3.3 gives the relation between the throughput of two arbitrary actors in consistent self-timed bounded SDFGs.

Lemma 3.4. *If a channel (p, q) connecting a to b is self-timed bounded then* $Th(b) = (Rate(p)/Rate(q))\,Th(a)$.

Lemma 3.5. *If a and b are two actors of an SCC of a consistent SDFG with repetition vector q, then* $Th(a)/q(a) = Th(b)/q(b)$.

Proof. We know that actors a and b are on a cycle. Let $a = i_1, i_2, \ldots, i_k = b, \ldots, i_l = a$ denote this cycle. If actors i_1 and i_2 are connected by channel (p_1, q_1), and i_2 and i_3 by (p_2, q_2) and so forth, then, by Corollary 3.2, we know that

$$\begin{aligned}
Th(a) &\leq \frac{Rate(p_{l-1})}{Rate(q_{l-1})}\,Th(i_{l-1}),\\
Th(i_{l-1}) &\leq \frac{Rate(p_{l-2})}{Rate(q_{l-2})}\,Th(i_{l-2}),\\
\ldots &\leq \ldots,\\
Th(i_{k+1}) &\leq \frac{Rate(p_k)}{Rate(q_k)}\,Th(b),\\
Th(b) &\leq \frac{Rate(p_{k-1})}{Rate(q_{k-1})}\,Th(i_{k-1}),\\
\ldots &\leq \ldots,\\
Th(i_2) &\leq \frac{Rate(p_1)}{Rate(q_1)}Th(a).
\end{aligned}$$

By combining the above equations, we obtain

$$Th(a) \leq \frac{Rate(p_{l-1})Rate(p_{l-2})\ldots Rate(p_k)}{Rate(q_{l-1})Rate(q_{l-2})\ldots Rate(q_k)}\,Th(b), \tag{3.4}$$

and

$$Th(b) \leq \frac{Rate(p_{k-1})Rate(p_{k-2})\ldots Rate(p_1)}{Rate(q_{k-1})Rate(q_{k-2})\ldots Rate(q_1)}\,Th(a). \tag{3.5}$$

Because q satisfies the balance equations, we can also write

$$\begin{aligned}
Rate(q_{l-1})q(a) &= Rate(p_{l-1})q(i_{l-1}) \Rightarrow \frac{Rate(p_{l-1})}{Rate(q_{l-1})} = \frac{q(a)}{q(i_{l-1})},\\
Rate(q_{l-2})q(i_{l-1}) &= Rate(p_{l-2})q(i_{l-2}) \Rightarrow \frac{Rate(p_{l-2})}{Rate(q_{l-2})} = \frac{q(i_{l-1})}{q(i_{l-2})},\\
\ldots &= \ldots,\\
Rate(q_k)q(i_{k+1}) &= Rate(p_k)q(b) \Rightarrow \frac{Rate(p_k)}{Rate(q_k)} = \frac{q(i_{k+1})}{q(b)},\\
Rate(q_{k-1})q(b) &= Rate(p_{k-1})q(i_{k-1}) \Rightarrow \frac{Rate(p_{k-1})}{Rate(q_{k-1})} = \frac{q(b)}{q(i_{k-1})},\\
\ldots &= \ldots,\\
Rate(q_1)q(i_2) &= Rate(p_1)q(a) \Rightarrow \frac{Rate(p_1)}{Rate(q_1)} = \frac{q(i_2)}{q(a)}.
\end{aligned}$$

By substitution into Inequalities (3.4) and (3.5) we have $Th(a) \leq (q(a)/q(b))Th(b)$ and $Th(b) \leq (q(b)/q(a))Th(a)$. Rewriting this result yields $Th(a)/q(a) \leq Th(b)/q(b)$ and $Th(b)/q(b) \leq Th(a)/q(a)$, which means that $Th(a)/q(a) = Th(b)/q(b)$, completing the proof. □

Proposition 3.3. *If a and b are two actors of a consistent self-timed bounded SDFG G with repetition vector q then $Th(a)/q(a) = Th(b)/q(b)$.*

Proof. Assume that a and b are actors of a consistent self-timed bounded SDFG. Suppose path $a = i_1, i_2, \ldots, i_k = b$ is an undirected path connecting a to b. If actors i_1 and i_2 are connected by (p_1, q_1) and i_2, i_3 are connected by (p_2, q_2) and so on, then according to Lemma 3.4 we have

$$\begin{aligned}
Th(b) &= \frac{Rate(p_{k-1})}{Rate(q_{k-1})}Th(i_{k-1}),\\
Th(i_{k-1}) &= \frac{Rate(p_{k-2})}{Rate(q_{k-2})}Th(i_{k-2}),\\
\ldots &= \ldots\\
Th(i_2) &= \frac{Rate(p_1)}{Rate(q_1)}Th(a).
\end{aligned}$$

The proof can now be completed along the lines of the proof of Lemma 3.5. □

3.5.2 Reduction to an HSDFG

In this section, we propose a method for reducing a consistent SDFG G to an HSDFG G_H which preserves (non-)liveness and self-timed (un)boundedness of

G. In G_H, every actor has a self-loop channel with one initial token, rates of all channels are one (i.e., it is an HSDFG), and, ignoring self-loops, it is acyclic. Because of these simple properties, we use the reduced graph for verifying the liveness and self-timed boundedness of the original SDFG.

The reduction requires the notion of local throughput of an SCC of an SDFG, and it is illustrated in Figure 3.4 which provides the reduced graph for the running example.

Definition 3.6. (Local Throughput of an SCC) *The* local throughput *$LTh(\kappa)$ of an SCC $\kappa = (A_\kappa, C_\kappa, E_\kappa)$ in a consistent SDFG $G = (A, C, E)$ with repetition vector q is defined as the actor throughput of an arbitrary actor $a \in A_\kappa$ when all input channels from $A \backslash A_\kappa$ to A_κ are removed, divided by $q(a)$.*

Lemma 3.5 implies that this definition is sound.

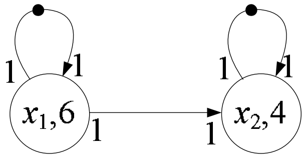


Figure 3.4: The reduced HSDFG for G_{ex}.

Definition 3.7. (Reduced Graph) *Let a consistent SDFG $G = (A, C, E)$ contain n maximal SCCs $\kappa_1 = (A_{\kappa_1}, C_{\kappa_1}, E_{\kappa_1}), \ldots, \kappa_n = (A_{\kappa_n}, C_{\kappa_n}, E_{\kappa_n})$. Suppose q is the repetition vector of G. We define the reduced SDFG $G_H = (A_H, C_H, E_H)$ as follows: $A_H = \{x_i | 1 \le i \le n\}$ (which means one actor for each maximal SCC in G); C_H contains a channel ch connecting actor x_i to actor x_j with production and consumption rates of one for every channel $ch' \in C$ connecting actor a to b where $a \in A_{\kappa_i}$, $b \in A_{\kappa_j}$, $i \ne j$; C_H also contains self-loop channels for every actor x_i with production and consumption rates of one; the execution time $E_H(x_i)$ equals $1/LTh(\kappa_i)$ if κ_i does not deadlock and ∞ if it does. Note that we obtain an HSDFG as the result. Finally, every self-loop channel in G_H contains one initial token, and all the other channels are empty.*

Since the HSDFG resulting from the reduction is acyclic when ignoring self-loops, the preservation of (non-)liveness and self-timed (un-)boundedness that we are aiming at, is independent of the number of initial tokens on the non-self-loop channels. Hence, we choose to leave those channels empty.

Consider the reduced graph shown in Figure 3.4. The original graph G_{ex} has two maximal SCCs, containing actor a, and actors b and c, respectively. These SCCs are reduced to actors x_1 and x_2. Since actor a has throughput $\frac{1}{2}$ and repetition-vector entry 3, the local throughput is $(1/2)/3 = 1/6$ and the execution time of x_1 is set to 6, illustrating that 3 firings of a take 6 time units. Considering the other SCC in isolation, it can be verified that one period of this

SCC containing 3 firings of b and 2 of c consists of 4 time units. Given the repetition vector of G_{ex} and Definition 3.6, this gives a local throughput of $\frac{1}{4}$ and an execution time of 4 for x_2.

The following proposition shows the relation between the throughput of actors in a maximal SCC of an SDFG and the throughput of the actor corresponding to that SCC in the reduced SDFG.

Proposition 3.4. *Let G_H be the reduced SDFG of a consistent timed SDFG G with repetition vector q. If a maximal SCC $\kappa = (A_\kappa, C_\kappa, E_\kappa)$ in G is replaced by actor x in G_H, then for any $a \in A_\kappa$, $Th(a) = q(a)\,Th(x)$.*

Proof. For proving this proposition, first we define an intermediate reduced graph in which only one of the SCCs is reduced. Then, we prove the proposition for this intermediate SDFG; Finally, the result is proven for the entire reduction. Let $\kappa = (A_\kappa, C_\kappa, E_\kappa)$ be a maximal SCC in G, q be the repetition vector of G and $LTh(\kappa)$ be the local throughput of κ. For any fresh actor name $x \notin A$, we define the κ-reduced SDFG $G_{\kappa\to x} = (A_x, C_x, E_x)$ as follows: $A_x = (A \backslash A_\kappa) \cup \{x\}$; C_x equals $C \backslash C_\kappa$ with every input channel connecting a to k with production and consumption rates of p and c where $a \in A \backslash A_\kappa$ and $k \in A_\kappa$ is replaced by a channel connecting a to x with production rate p and consumption rate $cq(k)$. Similarly, every output channel connecting k to a with production and consumption rates of p and c where $a \in A \backslash A_\kappa$ and $k \in A_\kappa$ replaced by a channel connecting x to a with production rate $pq(k)$ and consumption rate c. An extra self-loop channel for x with production and consumption rates of one and one initial token is added to C_x. The execution time $E_x(a)$ equals $E(a)$ for all actors $a \in A \backslash A_\kappa$, and for actor x it is set to $1/LTh(\kappa)$ if κ does not deadlock and ∞ if it does. The channels of $G_{\kappa\to x}$ contain the same number of initial tokens as the corresponding channels in G.

Next, for $G_{\kappa\to x}$, we prove the following equation, for every $a \in A_\kappa$.

$$Th(a) = q(a)\,Th(x). \tag{3.6}$$

First, assume κ has only self-timed unbounded input channels, if it has any input channels at all. Since κ is a maximal SCC, the actor firings in κ do not have impact on the production of any tokens in the input channels of κ. By the construction of $G_{\kappa\to x}$, any part of the graph producing tokens into input channels of κ remains unchanged in $G_{\kappa\to x}$. Furthermore, because all input channels of κ are self-timed unbounded, κ is not constrained by its input channels realizing a throughput equal to $LTh(\kappa)$, and because $Th(x) \leq LTh(x) = LTh(\kappa)$, x consumes tokens in a self-timed execution at most as fast as κ consumes the corresponding tokens. Therefore, also all input channels of x in $G_{\kappa\to x}$ are self-timed unbounded. This means that at some point in time x never has to wait for input tokens, which implies that $Th(x) = LTh(x)$. By the definition of the local throughput of an SCC, $LTh(x) = LTh(\kappa) = Th(a)/q(a)$, for some arbitrary a in A_κ, which proves Equation (3.6).

Second, we may assume that not all input channels of κ are unbounded. By Lemma 3.5, it suffices to prove Equation (3.6) for an arbitrary actor of κ.
Let channels ch_i be all channels connecting some actor b_i in $A \setminus A_\kappa$ to an actor a_i in A_κ with production and consumption rates of p_i and c_i. Denote this set as $InC(\kappa)$. Based on Corollary 3.2, $Th(a_i) \leq (p_i/c_i)Th(b_i)$ for all i. The definition of C_x in $G_{\kappa\to x}$ implies that for all i,

$$Th(x) \leq \frac{p_i}{c_i q(a_i)} Th(b_i).$$

Let ch_m be a channel from actor $b_m \in A \backslash A_\kappa$ to actor $a_m \in A_\kappa$ with production and consumption rates of p_m and c_m, such that, for all i,

$$\frac{p_m}{c_m q(a_m)} Th(b_m) \leq \frac{p_i}{c_i q(a_i)} Th(b_i),$$

i.e., ch_m is a channel which constrains the throughput of x the most. We continue to prove Equation (3.6) for actor a_m, i.e., we prove that $Th(a_m) = q(a_m)Th(x)$.
We show that ch_m is bounded. To show this by contradiction we assume that ch_m is unbounded. According to Lemma 3.3 we have

$$p_m Th(b_m) > c_m Th(a_m) \Rightarrow \frac{p_m}{c_m q(a_m)} Th(b_m) > \frac{Th(a_m)}{q(a_m)}.$$

Since we assumed that not all ch_i are unbounded, there exists a bounded channel $ch_k \in InC(\kappa)$. Therefore using first Lemma 3.5 and then Lemma 3.4 we can conclude that

$$\frac{p_m}{c_m q(a_m)} Th(b_m) > \frac{Th(a_m)}{q(a_m)} = \frac{Th(a_k)}{q(a_k)} = \frac{p_k}{c_k q(a_k)} Th(b_k),$$

which means that

$$\frac{p_m}{c_m q(a_m)} Th(b_m) > \frac{p_k}{c_k q(a_k)} Th(b_k), \tag{3.7}$$

which contradicts the choice of ch_m. Hence, channel ch_m must be bounded.
According to Lemma 3.2, the throughput of x can be calculated as follows:

$$Th(x) = \min\{\min_{ch_i \in InC(\kappa)} \frac{p_i}{c_i q(a_i)} Th(b_i), LTh(\kappa)\}.$$

The choice for ch_m implies that we can calculate the throughput of x as

$$Th(x) = \min\{\frac{p_m}{c_m q(a_m)} Th(b_m), LTh(\kappa)\}.$$

If the result of this minimum is $LTh(\kappa)$ then from the definition of the local throughput of an SCC and the definition of the execution time of x, Equation

(3.6) follows for a_m. In the other case, as ch_m is bounded, using Lemma 3.4, we have

$$Th(x) = \frac{p_m}{c_m q(a_m)} Th(b_m) = \frac{Th(a_m)}{q(a_m)},$$

which completes the proof for actor a_m, and hence for all actors in A_κ.
Finally, it is not difficult to see that by replacing all SCCs of SDFG G via the above intermediate reduction results in the reduced SDFG as defined in Definition 3.7 but with rates $pq(a)$ and $cq(a)$ for every channel resulting from a channel between actors a and b in G with rates p and c. Therefore, we can extend Equation (3.6) to all SCCs of G. According to the balance equations we know that for each channel in the original graph connecting actor a to b with production and consumption rates of p and c, $pq(a) = cq(b)$. Thus, the production and consumption rates for every channel obtained via the reductions so far are equal. Therefore, we can simplify the graph obtained so far to the reduced graph as defined in Definition 3.7. Since the simplification of the rates does not change the throughput of actors, the desired result follows. □

Consider for instance actor x_2 of the reduced graph of the running example. Its throughput in the reduced graph is fully determined by the throughput of x_1 and becomes therefore $\frac{1}{6}$. Proposition 3.4 states that $Th(b) = 3(\frac{1}{6}) = \frac{1}{2}$ and $Th(c) = 2(\frac{1}{6}) = \frac{1}{3}$, which agrees with the throughput values for b and c computed in Section 3.5.1.

Proposition 3.4 also implies that non-zero throughput (i.e., (non-)liveness) is preserved.

Corollary 3.3. *A consistent timed SDFG is live iff its reduced graph is live.*

The reduction also preserves self-timed (un-)boundedness.

Theorem 3.5. *A consistent timed SDFG is self-timed bounded iff its reduced graph is self-timed bounded.*

Proof. In this proof, we also use the intermediate reduced graph as explained in the proof of Proposition 3.4, in which only one of the SCCs is replaced by an actor in the intermediate reduced SDFG. Let an SCC $\kappa = (A_\kappa, C_\kappa, E_\kappa)$ of G be replaced by an actor x in $G_{\kappa \to x}$. According to Theorem 3.3, channels in a live and strongly connected SDFG are strictly bounded, which implies the self-timed boundedness of channels in C_κ. We want to prove that the input and output channels of actor x in the reduced graph are self-timed bounded iff the corresponding channels to/from κ are self-timed bounded.
First, we prove the sufficient (if) part. If an input channel connecting actor $b \in A \backslash A_\kappa$ to $a \in A_\kappa$ with production and consumption rates p and c in G is self-timed bounded, by Lemma 3.3 and Proposition 3.4 we have

$$p\,Th(b) \leq c\,Th(a) = cq(a)\,Th(x).$$

Thus, by Lemma 3.3 the channel from b to x with production and consumption rates p and $cq(a)$ is also self-timed bounded.
Now, we prove the sufficient part for output channels from κ in the same way. Suppose the channel from actor $a \in A_\kappa$ to actor $b \in A \backslash A_\kappa$ with production and consumption rates p and c is self-timed bounded. We prove that the channel connecting x to b with production and consumption rates $pq(a)$ and c is also self-timed bounded. Again using Lemma 3.3 and Proposition 3.4 we have

$$p\,Th(a) \leq c\,Th(b) \Rightarrow pq(a)\,Th(x) \leq c\,Th(b),$$

which by Lemma 3.3 proves self-timed boundedness of the channel connecting x to b.
Next, we prove the necessary (only if) part of the proof, namely, if the channel connecting an actor $b \in A \backslash A_\kappa$ to actor a inside A_κ with production and consumption rates p and c is not self-timed bounded, then the channel connecting b to x with production and consumption rates p and $cq(a)$ is also not self-timed bounded.
Again in a similar way by replacing $Th(a)$ with $q(a)Th(x)$ in the following inequality we can prove this part of theorem.

$$p\,Th(b) > c\,Th(a) = cq(a)\,Th(x).$$

Also for the case of an output channel which connects a to b with production and consumption rates p and c.

$$p\,Th(a) > c\,Th(b) \Rightarrow pq(a)\,Th(x) > c\,Th(b).$$

Thus, we may conclude that the reduction of one SCC to a single actor preserves self-timed (un-)boundedness.
Applying the intermediate reduction on G can be done iteratively in different and independent steps. Therefore, applying the intermediate reduction for each maximal SCC of G results in the reduced HSDFG as defined by Definition 3.7 when after all the intermediate reductions all rates are changed to one. As proven in this theorem, each intermediate reduction preserves self-timed (un)boundedness. Using similar arguments as those at the end of Proposition 3.4, it follows that the rate changes do not affect self-timed boundedness. Consequently, we can conclude that the reduced SDFG preserves self-timed (un)boundedness. ☐

3.5.3 Verifying Self-timed Boundedness

This section introduces an algorithm that determines whether an SDFG is live and self-timed bounded.

Algorithm *isLive&SelftimedBounded*(G=(A, C, E))
Input: A connected timed SDFG G

```
Output: "yes" if self-timed bounded and live, "no" otherwise
1.  if not $isLive\&Bounded(G)$
2.     then return "no"
3.  $G_H = (A_H, C_H, E_H) \leftarrow$reduce($G$)
4.  $AL[1..|A_H|] \leftarrow$topologicalSort($G_H$)
5.  if $|A_H| = 1$
6.     then return "yes"
7.  for $i \leftarrow 1$ to $|A_H|$
8.       do $AL[i].Th \leftarrow \frac{1}{E_H(AL[i])}$
9.          if $\mathrm{Pred}(AL[i]) = \{AL[i]\}$ and $AL[i].Th = \infty$
10.            then return "no"
11.         $maxPTh \leftarrow 0$
12.         for each $j \in \mathrm{Pred}(AL[i]) \backslash \{AL[i]\}$
13.              do $AL[i].Th \leftarrow \min(AL[i].Th, AL[j].Th)$
14.                 $maxPTh \leftarrow \max(maxPTh, AL[j].Th)$
15.         if $maxPTh > AL[i].Th$
16.            then return "no"
17. return "yes"
```

The algorithm works in two steps. The first step (lines 1 and 2) checks the liveness and boundedness (as defined by Definition 3.2) of the graph by calling algorithm *isLive&Bounded*. If the graph is not live and bounded, it cannot be live and self-timed bounded and "no" is returned. The second step (lines 3 to 17) concerns determining whether the reduced HSDFG is self-timed bounded.

If *isLive&Bounded* returns "yes", we know that the SDFG is consistent (Theorem 3.1). Then, line 3 of the algorithm reduces the SDFG according to Definition 3.7 and stores the result in G_H. Note that the reduction requires throughput calculations for all SCCs. For efficiency reasons, these throughput calculations can be delayed till the algorithm really needs this information. Calculations may then be avoided if the algorithm returns "no" early. We have not made this explicit in the algorithm. How throughput calculations are performed is the subject of Chapter 4. Since G is at this point known to be live and consistent, by Corollary 3.3, also G_H is live. It remains to determine self-timed (un-)boundedness.

Ignoring self-loops, G_H is acyclic. Line 4 topologically sorts the actors of G_H, and stores them in array AL, so that the predecessors of an actor $AL[i]$ are only among the $AL[j]$ for $j \leq i$. If G_H contains only one actor, then G is strongly connected, and hence, by Proposition 3.2, self-timed bounded, and the algorithm terminates.

Each iteration i, with $1 \leq i \leq |A_H|$, of the loop of lines 7 to 16 starts by calculating the local throughput of the corresponding actor $AL[i]$, storing the result in $AL[i].Th$. If a source actor is detected (an actor without any input channel except its self-loop channel) with an infinite throughput, the algorithm returns "no", because this implies that its output channels are unbounded. The loop continues by setting *maxPTh* to zero. This variable is a temporary variable for storing the maximum throughput of the predecessors of actor $AL[i]$. In the loop of lines 12 to 14, the minimum between the local throughput of actor $AL[i]$ and the minimum throughput of its predecessors is assigned to $AL[i].Th$. This value, according to Lemma 3.2, is the throughput of the actor $AL[i]$. Note that since the actors are topologically sorted in AL, the throughputs of all predecessors have already been calculated. The maximum throughput of the predecessors of actor $AL[i]$ is assigned to $maxPTh$.

The test of line 15 checks whether the maximum throughput of predecessors of actor $AL[i]$ (excluding $AL[i]$) is greater than the throughput of actor $AL[i]$ itself. In case it is, according to Lemma 3.3, at least one channel connecting a predecessor of actor $AL[i]$ to $AL[i]$ is unbounded.

If the algorithm reaches line 17, then no unbounded channel has been detected, and the graph is live and self-timed bounded. The following theorem follows.

Theorem 3.6. *A timed SDFG G is live and self-timed bounded iff isLive&Selftimed Bounded(G) returns "yes".*

3.6 Related Work

Both liveness and boundedness are standard properties studied for all subclasses of Petri nets [48]. Recall that SDFGs are a subclass of Petri nets. Thus, it makes sense to compare the results obtained in this chapter with the corresponding results in the literature concerning Petri nets. We studied liveness in combination with three different definitions of boundedness (Definitions 3.2, 3.3 and 3.4) for (timed) SDFGs.

We do not know of any related results for boundedness as defined by Definition 3.2. The only reference we know to this type of boundedness is in [50] which only introduces it without providing necessary and sufficient conditions, as we do.

For strict boundedness in the sense of Definition 3.3, the problem has been studied from different viewpoints in the Petri-net literature (see for an overview [20, 48]). In particular, [68] gives necessary and sufficient conditions for strict boundedness of live weighted Marked Graphs (our Theorem 3.3). Strict boundedness is also the only kind of boundedness which has been investigated formally in the literature on SDFGs themselves; Karp and Miller in their seminal paper [38] introduced computation graphs, which are slightly more general than SDFGs. They proved necessary and sufficient conditions for liveness and strict boundedness in their model. Their results as well as those in [68] are the same as those presented in this chapter.

Our third definition of boundedness, self-timed boundedness (see Definition 3.4) is defined only on timed SDFGs. Therefore, we need to compare it with time-enabled Petri nets. Petri nets have been extended with quantitative time in different ways, by adding timing information to places, transitions and/or tokens (see [10] for a survey). The timed Petri net model that comes closest to timed SDFGs is the "time Petri net" model originally defined by [45]. This extension of Petri nets associates a duration (delay) and a deadline to transitions. We are not aware of any study of the self-timed boundedness problem for the subclass of time Marked Graphs. In [55], the liveness and strict boundedness problem for time Petri nets is studied but only some sufficient conditions are given. These conditions guarantee that once a time Petri nets satisfies certain syntactic constraints, it is live and strictly bounded if the underlying untimed Petri net is live and strictly bounded. Unfortunately, the results of [55] cannot be applied in our setting since the syntactic constraints require the absence of either duration or deadline both of which are necessary for translation of timed SDFGs to time Petri nets. [36] proves a general undecidability result for strict boundedness of time Petri net of [45]. However, in [6], two sufficient conditions are given for strict boundedness of time Petri nets. We are not aware of any result about self-timed boundedness as defined in Definition 3.4. To the best of our knowledge, both the concept and the derived results are novel.

3.7 Summary

We have studied the liveness and boundedness of Synchronous Data Flow Graphs. Liveness and boundedness are prerequisites of any meaningful SDFG model of a streaming multi-media application. Two known notions of boundedness, namely boundedness and strict boundedness, have been studied rigorously, and in particular necessary and sufficient conditions for liveness in combination with these two types of boundedness have been given. For strict boundedness, these conditions are known from the Petri-net literature. Furthermore, a new notion, self-timed boundedness, is introduced. Self-timed boundedness checks whether self-timed execution of an SDFG is bounded. A self-timed execution yields the maximum throughput attainable by an SDFG. Necessary and sufficient conditions for self-timed boundedness and liveness are given and proven for SDFGs with fractional actor execution times. An algorithm for checking these conditions is presented. The results depend on the periodicity of self-timed execution, which is proven in Proposition 4.1 in Chapter 4. It is an open problem whether this periodicity result can be generalized to SDFGs with arbitrary real execution times.

Chapter 4

Throughput

4.1 Overview

The main aim of modeling applications using Synchronous Data Flow Graphs is to provide performance metrics to be used in predictable designs. Among different performance indicators, throughput is the most prominent one.

Throughput analysis of dataflow graphs has been extensively studied in the literature ([16, 17, 37, 56, 73]). All existing throughput analysis approaches for SDFGs suggest algorithms which are based on an analysis of the structure of the graphs. The drawback of these approaches is that they are not directly applicable to SDFGs, but can only be applied to homogeneous SDFGs (HSDFGs). Therefore, these methods require a conversion from an SDFG to an equivalent HSDFG, which is always possible in theory as explained in Section 2.6, but frequently leads to a prohibitively large increase in the size of the graph, causing algorithms to fail to produce a result within reasonable time.

In this chapter, we propose a different method for computing the throughput of SDFGs. This method, unlike existing methods, works directly on an SDFG, avoiding the costly conversion to an HSDFG. The method generates and analyzes the SDFG's dynamic state space by executing the graph. Although the number of states that may need to be generated can also be large in unfavorable situations, experiments show that the method performs well in practice because the number of states that actually needs to be stored is almost always very limited.

The rest of the chapter is organized as follows. Section 4.2 discusses throughput of data flow graphs and our method for analyzing throughput, initially focussing on strongly connected graphs. Our approach for throughput calculation has direct links with a method in the data flow literature called Max-Plus algebra. To show these links, Section 4.3 places the method in the context of Max-Plus algebra and shows that the results are equivalent to spectral analysis of the Max-Plus equivalent of an SDFG. Section 4.4 explains our experimental method and presents the results of the experiments, comparing the performance of the proposed method with state-of-the-art throughput analysis through minimum cycle mean algorithms. Section 4.5 extends the results to arbitrarily connected graphs. Section 4.6 discusses related work and finally Section 4.7 summarizes. This chapter is based on publication [29].

4.2 Throughput Analysis of SDF Graphs

In this section, the maximum throughput of (executions of) SDFGs is studied. First, some of the properties of SDFG state spaces are shown, then the definition of throughput for SDFGs is given. Prevailing methods for throughput analysis of SDFGs are explained and the new approach given in this thesis is also formulated in this section. Initially, the throughput analysis of strongly connected SDFGs is studied; later on in Section 4.5, this method is extended for arbitrary SDFGs. In the remainder of this section, SDFGs are assumed to be strongly connected and consistent. We also assume rational numbers for actor execution times.

4.2.1 The Self-timed Execution State-Space

The operational semantics of SDFGs with a self-timed execution policy leads to a state-space of a particular shape, illustrated in Figures 2.2 and 3.2 for graphs depicted in Figures 2.1 and 3.1, respectively. It consists of a finite sequence of states and transitions (called the *transient phase*), followed by a sequence that is periodically repeated ad infinitum (the *periodic phase*). The following proposition shows that self-timed execution of SDFGs always constructs a similar shape as the above mentioned examples.

Proposition 4.1. *For every live, consistent and strongly connected SDFG, the self-timed state-space consists of a transient phase, followed by a periodic phase.*

Proof. Self-timed execution is deterministic if we consider the execution in macro steps as explained in Section 2.5. Also, according to Proposition 3.2 in Section 3.5.1, a live, consistent and strongly connected SDFG is self-timed bounded. Therefore, the number of states of an SDFG in self-timed execution is finite. This guarantees that the execution will eventually revisit some state that was visited before, signifying the fact that (because of determinism) the execution is then in the periodic regime. □

If we have a closer look at the periodic behavior of the graph, we observe the following.

Proposition 4.2. *The periodic behavior of a live and strongly connected SDFG consists of a whole number of iterations.*

Proof. A single execution of the periodic behavior has no net effect on the number of tokens in the different channels, because it returns to the same state, which includes the amounts of tokens in channels. From this, it follows immediately that the number of actor firings (starts and ends) satisfies the SDFG's balance equations and thus must be a multiple of the repetition vector. □

4.2.2 Throughput

The maximum throughput of a timed SDFG $G = (A, C, E)$ is achieved with the self-timed execution of G [33, 60], as no actor $a \in A$ can start a firing without having enough tokens in all of its input channels and any delay in the start of firing of an actor is of no use in increasing the number of firings of a itself or any other actor in the graph. Hereafter in this thesis, the focus is on the throughput associated with the self-timed execution, unless mentioned otherwise.

Lemma 4.1. *For every live, consistent and strongly connected timed SDFG $G = (A, C, E)$, the throughput of an actor $a \in A$ is equal to the average number of firings per time unit in the periodic part of the self-timed state space.*

Proof. Considering the state space of the self-timed execution of G, there is some K such that for all $i > K$, the i-th firing of a is in the periodic phase. Let $|p|$ and $|p|_a$ respectively, be the sum of the lengths of the clock transitions and the number of $\overset{(a,start)}{\rightarrow}$ transitions in one period. The k-th firing of a, when in the periodic phase, can be decomposed as follows: $k = K + m|p|_a + r$ for some non-negative m and r where $1 \leq r \leq |p|_a$. The corresponding time of the start of that firing $S_{a,k} = T + m|p| + T_r$, where T is the start time of the k-th firing of a and T_r the time of the r-th firing of a in the period, relative to T. Then using Proposition 4.2,

$$Th(a) = \lim_{k \to \infty} \frac{k}{S_{a,k}} = \lim_{m \to \infty} \frac{K + m|p|_a + r}{T + m|p| + T_r} = \frac{|p|_a}{|p|}.$$

□

Lemma 3.5, proven in Section 3.5.1, can also be immediately concluded from Lemma 4.1 and Proposition 4.2 and the observation that it is trivial for non-live strongly connected components. This lemma implies the following proposition.

Proposition 4.3. *For a consistent and strongly connected timed SDFG (A, C, E) with repetition vector q and actors $a, b \in A$, $Th(a)/q(a) = Th(b)/q(b)$.*

This proposition means that we can define a normalized notion of (maximal) throughput, independent of any specific actor, that applies to the self-timed execution. Intuitively, it expresses the number of iterations of the graph executed per time unit.

Definition 4.1. (SDFG THROUGHPUT) *The throughput of a consistent strongly connected timed SDFG* $G = (A, C, E)$ *is defined as* $Th(G) = \frac{Th(a)}{q(a)}$, *for an arbitrary* $a \in A$.

Lemma 4.3 guarantees that the result is independent of the chosen actor a and thus the definition is well-defined. From Lemma 4.1 we know that the throughput of a strongly connected SDFG can be determined from the periodic part of the state space.

Corollary 4.1. *The throughput of a strongly connected SDFG is equal to the number of actor firings per time unit during one period normalized by the repetition vector. This in turn is equal to the number of iterations executed in one period divided by the duration (the sum of the lengths of clock transitions) of one period.*

Continuing the example of Figure 2.2 of Section 2.5, it can be seen that the periodic phase takes 12 time units and includes one iteration of the graph. Actor d executes 3 times during this period. Hence, the throughput of d equals $3/12 = 1/4$. The normalized throughput of the SDFG itself is 1/12, corresponding to the execution of one graph iteration per 12 time units.

We are now also able to express the relation between throughput of an SDFG and its equivalent HSDFG.

Theorem 4.1. *Let* G *be an SDFG and* H *the corresponding HSDFG obtained from the conversion algorithm of [60] (see Section 2.6), then* $Th(G) = Th(H)$.

Proof. Trivial in case of deadlock. Otherwise, let a be an actor of G and q the repetition vector of G. For any k, we have $i \geq 0$ and $0 \leq r < q(a)$ such that $k = i \cdot q(a) + r$ and

$$Th(G) = \frac{1}{q(a)} \lim_{i \to \infty} \frac{i \cdot q(a) + r}{S_{a, i \cdot q(a) + r}}.$$

From the correspondence between the SDFG and HSDFG discussed in Section 2.6, we have that $S_{a, i \cdot q(a) + r} = S_{a_r, i}$. Thus,

$$Th(G) = \frac{1}{q(a)} \lim_{i \to \infty} \frac{i \cdot q(a) + r}{S_{a_r, i}} = \lim_{i \to \infty} \frac{i + r/q(a)}{S_{a_r, i}} = \lim_{i \to \infty} \frac{i}{S_{a_r, i}} = Th(H).$$

□

The throughput of the SDFG can be determined from the state space. Often, it is also interesting to determine the critical components, i.e., the actors and channels that are constraining the throughput. These are candidates to improve (speed of an actor or capacity of a channel) if we need to increase throughput. This type of information can also be extracted from the state space, see [64] for an example.

Traditionally (see e.g., [60]), throughput of an SDFG is defined as 1 over the *Maximum Cycle Mean* (MCM) of the corresponding HSDFG. The cycle mean of some cycle of an HSDFG or weighted directed graph in general is defined as the total execution time or total weight of the cycle divided by the number of tokens or the number of arcs in that cycle for the HSDFG and weighted directed graph respectively. The maximum cycle mean over all cycles in the HSDFG or weighted directed graph is called the MCM of the graph. The MCM can be shown ([56],[60, Lemma 7.3]) to be equal to the average time between two firings of any of the HSDFG actors. Given Theorem 4.1, and the observation that all repetition vector entries of an HSDFG are 1, it is easy to see that Definition 4.1 of SDFG throughput is the same as the traditional definition of throughput.

Corollary 4.2. *Let G be an SDFG and H the HSDFG obtained from the conversion of [60], then $Th(G)$ is equal to $1/\lambda^*$ if λ^* is the MCM of H.*

The suggested method (see, e.g., [60]) for computing the throughput of an SDFG is as follows. First, convert the SDFG to an equivalent HSDFG and then compute the throughput on this graph. The throughput of the HSDFG can be computed through an MCM algorithm ([17, 37]). In Sections 2.5.3 and 2.5.4 of [3] an approach is described to convert an HSDFG to a weighted directed graph in which each channel contains one token and is annotated with a cost (execution time). The MCM of this graph then equals one over the throughput of the HSDFG. An alternative method to compute the throughput is the use of a Maximum Cycle Ratio (MCR) algorithm [17]. Each edge in the weighted directed graph for the MCR algorithm has a cost (execution time of the producing actor in the HSDFG) and a transit time (number of tokens on the channel in the HSDFG). Efficient algorithms for calculating MCMs/MCRs exist, which are compared in [16]. However, MCM/MCR analysis can only be applied to an HSDFG which is often exponentially larger in size than the original SDFG. This makes the approach as a whole not particularly efficient for SDFG throughput analysis, as the experiments below confirm.

4.2.3 The State-Space Exploration Method

We propose a method that calculates the throughput of an SDFG by directly executing its self-timed behavior. For our method, we enforce a deterministic order of the interleaving of concurrent transitions corresponding to simultaneous start and end transitions in between clock transitions (see Section 2.5). This has no effect on the throughput, but in this way, the entire state space becomes deterministic.

In principle, we can execute the SDFG while remembering all states we visit until we detect that we are in the periodic phase when we encounter a state that we have visited before. At that point, by Corollary 4.1, we can calculate the throughput of the graph by counting, for one period, the number of iterations that were executed and the total lengths of clock transitions. Their quotient is the throughput.

We have to store states to detect the periodic phase, but the lengths of the transient and periodic phases can be fairly long and we may need to store a large number of states. The determinism in the state space however, allows us to store only selected states. Suppose we pass a state that was visited before, but not stored. We then continue the execution in the same way as the first time, revisiting the same states. We only need to be sure that at least one of the states in the periodic part is actually stored and when we revisit it, we encountered the cycle. Knowing from Proposition 4.2 that the periodic behavior consists of a whole number of iterations, we choose to only store one state for every iteration. In this way, the periodic behavior always includes at least one state that is stored. It further allows us to know without extra cost how many iterations occurred in the period.

We can do this as follows. We pick an arbitrary actor a. Then every iteration includes $q(a)$ start (and end) transitions of a. We choose to store all the states reached immediately after every $q(a)$-th execution of a start transition of a.

Using this method, one can detect the period and also the number of iterations of the period and the length in clock transitions can be easily calculated if we additionally store the sum of clock-transition lengths between each two stored states. With this information, we can calculate the throughput of an SDFG. In this manner we can significantly decrease the number of states that need to be stored and compared in comparison to the naive approach that stores all states, and consequently we can reduce the memory and time needed for the algorithm.

Since the method is obtained by some additions (storing and comparing states) to the execution of the behavior of the SDFG, it is relatively simple to integrate the analysis method into existing simulation tools for SDFGs.

4.3 Max-Plus Algebraic Characterization

A very elegant model to reason about (H)SDFGs is the Max-Plus algebra [3, 13]. Execution of a data flow graph is captured as a linear transformation in a special algebra and linear algebra theory is used to analyze such systems. In particular, spectral analysis is directly related to the throughput analysis problem. In Section 4.2, the relation between throughput of an SDFG and the MCM of the equivalent HSDFG was shown. The relation between the MCM and Max-Plus algebra is discussed in [3]. In this section, we study directly the relation between throughput of SDFGs, our throughput analysis algorithm, and Max-Plus algebra. The discussion in this section intends to provide additional insights into the asymptotic behavior of the state-space based throughput calculation algorithm. Reading it is however not required to understand the rest of this chapter.

We first explain some basics of Max-Plus algebra and then talk about the relation between our state-space exploration method and the Max-Plus formulation of (H)SDFG.

4.3.1 The Max-Plus model of (H)SDFGs

In a self-timed execution of an HSDFG, each actor starts a firing when there is at least one token on all of its input channels. The existence of these tokens on the input channels depends in turn on the end of actor firings which provide tokens to the channels. In this way, the start times of each actor firing can be expressed in terms of the start times of certain other actor firings. In this section, we assume an HSDFG (A, C, E) with initial token distribution γ_0.

Recall that $S_{a,k}$ denotes the start time of the k-th firing of actor $a \in A$ in the self-timed execution. When it ends, it produces the $k + \gamma_0(c)$-th token on every channel c connected to one of its output ports. We additionally define $F_{c,k}$ as the time at which the k-th token is produced on channel $c \in C$ (where $F_{c,k} = 0$ for all $0 \leq k < \gamma_0(c)$, because the initial tokens are already there from the start). $S_{a,k}$ depends on the availability of tokens on all of its inputs and starts as soon as the last of the required tokens has arrived. The tokens are produced when the actor writing to that channel finishes its firing. From this, we derive the following equations for the firing times of actors. For each actor $a \in A$, we have the equations (for all $k \geq 0$):

$$S_{a,k} = \max_{c \in InC(a)} F_{c,k}.$$

For each $a \in A$ and channel $c \in OutC(a)$, we have the equations (for all $k \geq \gamma_0(c)$):

$$F_{c,k} = S_{a,k-\gamma_0(c)} + E(a).$$

Combined, this gives a set of equations in which the k-th firing time of every actor is related to the k-th or earlier firing times of other actors. Through substitution and introduction of auxiliary variables (see [13] for details), this set of equations is converted to a set of difference equations of the form:

$$S_{i,n} = \max_j S_{j,n-1} + E_{i,j} \tag{4.1}$$

where the set of variables $S_{i,n}$ includes the firing times of the actors $S_{a,n}$.

It is convenient to formulate these equations using Max-Plus algebra [3] notation. Max-Plus algebra, like conventional algebra, is defined on real numbers $\mathbb{R}$. In Max-Plus algebra the maximum operator is used in the role of addition and is denoted by $\oplus$, and addition, denoted by $\otimes$, is used instead of multiplication. From this, a linear algebra is obtained and equation (4.1) can be represented using the Max-Plus formulation as follows:

$$S_{i,n} = \bigoplus_j S_{j,n-1} \otimes E_{i,j}.$$

This set of sum-of-products equations can be encoded as a matrix equation.

$$\mathbf{t}_n = \mathbf{M}\mathbf{t}_{n-1}.$$

where vector $\mathbf{t}_n$ consists of all $S_{i,n}$. $\mathbf{M}$ is a matrix with the coefficients $E_{i,j}$. If $\mathbf{t}_0$ encodes the initial token distribution, then the sequence $\{\mathbf{t}_k \mid k \geq 0\}$, where $\mathbf{t}_k = \mathbf{M}^k\mathbf{t}_0$ describes the evolution of the HSDFG over time. The eigenvalue equation plays a special role in this.

$$\mathbf{M}\mathbf{t} = \lambda^* \otimes \mathbf{t}.$$

The solution characterizes the graph in its periodic phase. For such a vector $\mathbf{t}$ all execution times of the next iteration ($\mathbf{M}\mathbf{t}$) are equal to the corresponding execution times of the current iteration, shifted by λ^* units of time. With $\mathbf{t}$ being an eigenvector, the same shift occurs for the next iteration and so on. Hence the behavior is periodic and the corresponding throughput is $1/\lambda^*$ where λ^* equals the MCM of the graph. Note that through the connection between firing times in SDFGs and in their corresponding HSDFGs, as discussed in Section 2.6, this model also applies to the execution of SDFGs if we take all firing times of one iteration in a single vector. We use this fact in the next section to model our state space exploration method.

4.3.2 A Max-Plus Model of the State-Space Exploration

We now show how computation of throughput with the state space exploration method can be interpreted as a computation of the eigenvalue of the corresponding matrix. This is akin to the so-called power method for computing the dominant eigenvalue in conventional linear algebra (see e.g., [4]).

The vectors $\mathbf{t}_n$ of the previous section capture the absolute firing times of the actors in the execution of the graph. In the state space we defined, and our exploration of the state space, we are not concerned with the absolute firing times, but only relative times, such as remaining execution times of actors. Since we store one state for every iteration, we can build a vector of all actor firing times of an entire iteration, counting relative to the starting time of the particular actor firing used to determine which state is being stored. Assume (without loss of generality) that the time of that actor firing is the first element of the vector: $\mathbf{t}_n(1)$. Define $\mathbf{u}_k$ as the relative version of $\mathbf{t}_k$, by subtracting the first entry from each of the entries, which gives all of the firing times relative to the moment the state was last stored,

$$\mathbf{u}_k = \frac{\mathbf{t}_k}{\mathbf{t}_k(1)}.$$

(A division by a scalar t denotes a Max-Plus multiplication ($\otimes$) with the inverse of t, i.e., a subtraction of t in conventional algebra.) We can then derive the following equation.

$$\mathbf{u}_{k+1} = \frac{\mathbf{t}_{k+1}}{\mathbf{t}_{k+1}(1)} = \frac{\mathbf{M}\mathbf{t}_k}{(\mathbf{M}\mathbf{t}_k)(1)} = \frac{\frac{1}{\mathbf{t}_k(1)}\mathbf{M}\mathbf{t}_k}{\frac{1}{\mathbf{t}_k(1)}(\mathbf{M}\mathbf{t}_k)(1)}$$

$$= \frac{\mathbf{M}\frac{\mathbf{t}_k}{\mathbf{t}_k(1)}}{(\mathbf{M}\frac{\mathbf{t}_k}{\mathbf{t}_k(1)})(1)} = \frac{\mathbf{M}\mathbf{u}_k}{(\mathbf{M}\mathbf{u}_k)(1)}.$$

We now have a recursive equation which characterizes the execution of the state-space exploration method. Similarly, one can show that for any $k \geq 0$ and $d \geq 0$.

$$\mathbf{u}_{k+d} = \frac{\mathbf{M}^d\mathbf{u}_k}{(\mathbf{M}^d\mathbf{u}_k)(1)}.$$

From the fact that this execution ends in a periodic phase, we conclude that there exist m and d such that:

$$\mathbf{u}_{m+d} = \mathbf{u}_m = \frac{\mathbf{M}^d\mathbf{u}_m}{(\mathbf{M}^d\mathbf{u}_m)(1)}.$$

Hence, with $\mu = (\mathbf{M}^d\mathbf{u}_m)(1)$, we have a solution to the eigenvalue equation:

$$\mathbf{M}^d\mathbf{u}_m = \mu \otimes \mathbf{u}_m.$$

Here, μ is the total length of the d iterations in the periodic phase and hence, $\mu = \lambda^{*d}$ ($\mu = d \cdot \lambda^*$ in common algebra), i.e., d times the eigenvalue λ^* of $\mathbf{M}$ which is identical to d times the MCM of the equivalent HSDFG which is identical to d divided by the throughput of the SDFG.

4.4 Experimental Results

In this section, we discuss the experimental validation of our throughput analysis method and tool. We first discuss some implementation details of the tool. Then, the experimental setup is discussed and the benchmarks we have selected for evaluation. Subsequently, we present the results of the evaluations and conclusions.

4.4.1 The SDFG Throughput Analysis Tool

This section very briefly explains the implementation of the throughput analysis algorithm based on the method proposed in this chapter. This method has been implemented in the tool, called *SDF*^{3} [65], which takes an XML description of an SDFG as input and can then calculate among others the throughput for the supplied SDFG.

The state of an SDFG consists of a tuple (γ, υ). To implement γ, an array with the size of the number of channels can be used. The function υ associates with each actor a multiset of numbers representing the remaining times of different actor firings of the actors.

The algorithm builds the state space of the graph as outlined in Section 4.2.3. A recurrent state (i.e., a cycle) must be detected from which the throughput can be computed. States are stored on a stack, indexed using a heap. This heap reduces the number of states which must be compared for equality even further, which makes the throughput calculation faster. When a recurrent state is detected, the program computes the throughput from the period.

4.4.2 Experimental Setup

To the best of our knowledge, all existing techniques to compute the throughput use a conversion to an HSDFG, followed by MCR analysis or via an additional conversion to a weighted directed graph followed by MCM analysis. Alternatively, spectral analysis of the Max-Plus formulation of the HSDFG can be used. In [18], Dasdan et al. give an extensive comparison of existing MCM, MCR and spectral analysis algorithms. It shows that Dasdan & Gupta's algorithm (DG) [17] which is a variant of Karp's algorithm [37] and Howard's algorithm (HO) [13] that uses spectral analysis have the smallest running times when tested on a large benchmark. In [16], Dasdan shows that also Young-Tarjan-Orlin's algorithm (YTO) [73] has a very good practical running time. Originally YTO is formulated as an MCM problem, but Dasdan gives pseudo-code for an MCR formulation of the problem. In our experiments, we compare the running times of our state-space exploration method with these state-of-the-art analysis algorithms.

All algorithms are implemented in SDF^3 for comparison. For the implementation of HO, the source code offered by the authors of [13] is used. An implementation of YTO is available via [46]. It uses the MCM formulation of the problem. The implementation of the DG algorithm was provided by Sander Stuijk, the main developer of SDF^3 tool, using the pseudo-code given in [17]. These algorithms compute the minimum cycle mean of a graph, while throughput analysis requires a maximum cycle mean computation. All implementations were modified to compute this maximum cycle mean. Our comparison requires a conversion from an SDFG to the weighted directed graphs which are input for the MCM algorithms. This conversion consists of two steps. First, an SDFG is converted into an equivalent HSDFG using the algorithm proposed in [60]. Second, the HSDFG is converted into a weighted directed graph using the approach suggested in Sections 2.5.3 and 2.5.4 of [3]. This step requires the computation of the longest path through a graph from each edge with initial tokens to any node in the graph reachable from this edge without using other edges that contain initial tokens.

We measure the running times of each of the two conversion steps and the MCM algorithms individually. These three values per experiment provide insight in the contribution of the different steps to the total running time required for computing an SDFG throughput. We also measure the running time of our state-space exploration method. For this approach, the measured running time consists of the time needed for the self-timed execution, storing and comparing states and computing the throughput from the state space.

4.4.3 Benchmark

Currently no standard benchmark set of SDFGs exists. (Note that the benchmark used in [16, 18] is a set of directed graphs, and cannot be used for our purposes.) To compare the running times of existing approaches for calculating throughput to our approach, we developed specific sets of test graphs. The first set of graphs in the benchmark are actual DSP and multimedia applications, modeled as SDFGs. From the DSP domain, the set contains a modem [8], a satellite receiver [58] and a sample-rate converter [8], and from the multimedia domain an MP3 decoder [61], an H.263 decoder [61], an H.263 encoder [49], and an MP3 playback application [71]. In all graphs, a bound on the storage space of each individual channel is modeled in the graph. Their bounds are set to the minimal storage space required for a non-zero throughput and is computed using the technique from [24].

As a second set of graphs ('Mimic DSP'), the benchmark contains 100 random SDFGs generated with SDF^3 [65] in which actor ports have small rates and the actors have small execution times. These settings for the rates and execution times make the graphs representative for SDFGs of DSP applications.

The practical problem with the existing state-of-the-art algorithms for throughput of an SDFG is that the conversion to an HSDFG can lead to an exponential increase in the number of actors [51]. Our approach should not be affected by this problem. To test this hypothesis, the benchmark contains a set ('Large HSDFG') of 100 randomly generated graphs in which the rates have a large variation (which tends to cause the exponential increase in the conversion) and all actors have equal execution times (this avoids long transient phases).

A potential problem with our approach is that the self-timed execution must first go through the complete transient phase, while the existing MCM algorithms are not affected by this issue. To test the impact of this potential problem on our approach, the benchmark contains a set ('Long transient') of 100 randomly generated SDFGs in which all actors have a large execution time with a small variation. Such SDFGs typically have a transient phase with a large number of transitions. Further all ports have a rate of 1, which makes the SDFGs effectively HSDFGs. This avoids an exponential increase in the number of actors during the SDFG to HSDFG conversion, which is also favorable to traditional throughput analysis methods and thus represents the most difficult input for our algorithm.

4.4.4 Results

Using the three algorithms described in Section 4.4.2 and our state-space exploration method (SS), we computed the throughput for all SDFGs contained in the

Table 4.1: DSP and multimedia applications

	SS	DG	HO	YTO
Modem [s]	$1{\cdot}10^{-3}$	$82{\cdot}10^{-3}$	$81{\cdot}10^{-3}$	$81{\cdot}10^{-3}$
Sample rate [s]	$1.99{\cdot}10^{-3}$	> 1800	> 1800	> 1800
Satellite [s]	$54.99{\cdot}10^{-3}$	> 1800	> 1800	> 1800
MP3 decoder [s]	$<1{\cdot}10^{-3}$	$1{\cdot}10^{-3}$	$1{\cdot}10^{-3}$	$1{\cdot}10^{-3}$
MP3 Playback [s]	$12.99{\cdot}10^{-3}$	> 1800	> 1800	> 1800
H.263 decoder [s]	$6.99{\cdot}10^{-3}$	> 1800	> 1800	> 1800
H.263 encoder [s]	$1{\cdot}10^{-3}$	> 1800	> 1800	> 1800

four sets of our benchmark. Table 4.1 shows the measured running times for the real DSP and multimedia applications. The MCM algorithms can only compute the throughput for the MP3 decoder and modem within 30 minutes. They do not complete the HSDFG to weighted directed graph conversion for the other models. Our algorithm computes the throughput for all graphs within 0.1 seconds, and often finishes within a few milliseconds.

The most important characteristics of the SDFGs in the three synthetic benchmark sets are shown in the first three rows of Table 4.2. The size of transient and periodic phases of the state space of the SDFGs are given in the table in terms of the average number of iterations. For the MCM algorithms, we measured the running time of the conversion from the SDFG to the HSDFG, the running time of the conversion from the HSDFG to the weighted directed graph and the running time of DG, HO, and YTO, separately. For the state-space exploration method, we measured the total running time of the algorithm.

For some of the graphs, it was not possible to compute the throughput within 30 minutes using the HO, YTO or DG algorithms. This is caused by the exponential increase in the number of actors when converting an SDFG to an HSDFG. For these graphs, the throughput calculation is stopped and the running times are not taken into account in the results. This provides an optimistic estimate of the real average running time of the existing approaches on the benchmark.

The columns labeled 'Mimic DSP', 'Large HSDFG' and 'Long transient' in Table 4.2 show the results of our experiments for the corresponding set of SDFGs. For the MCM algorithms, two conversion steps must be performed before the actual MCM analysis can be performed. The section labeled 'SDFG to HSDFG conversion' in Table 4.2 shows the measured running time for the conversion from an SDFG to an HSDFG. For 10 graphs from the set 'large HSDFG', it was not possible to complete the conversion within 30 minutes. The second step is the conversion from an HSDFG to a weighted directed graph. The results for the step are shown in the section labeled 'HSDFG to digraph conversion' in the Table. 53 graphs fail to finish this step before the time deadline (see row '#SDFGs not

Table 4.2: Experimental results on synthetic benchmark sets

	Mimic DSP	Large HSDFG	Long transient
avg #actors (SDFG)	20	13	14
avg #actors (HSDFG)	1008	8166	14
avg #iterations transient phase	2.72	16.85	47.76
avg #iterations periodic phase	1.04	1.02	1.58
SDFG to HSDFG conversion			
avg to HSDF [s]	$242\cdot10^{-3}$	2	–
var to HSDF [s^2]	$264\cdot10^{-3}$	11	–
#SDFGs not solved	0	10	0
HSDFG to digraph conversion			
avg to digraph [s]	$479\cdot10^{-3}$	218	$<1\cdot10^{-3}$
var to digraph [s^2]	$17\cdot10^{3}$	$95\cdot10^{3}$	$<1\cdot10^{-3}$
#SDFGs not solved	9	44	0
MCM algorithms			
avg MCM (DG)[s]	$271\cdot10^{-3}$	2	$<1\cdot10^{-3}$
avg MCM (HO)[s]	$1\cdot10^{-3}$	$9\cdot10^{-3}$	$<1\cdot10^{-3}$
avg MCM (YTO)[s]	$1\cdot10^{-3}$	$8\cdot10^{-3}$	$<1\cdot10^{-3}$
var MCM (DG)[s^2]	$565\cdot10^{-3}$	120	$<1\cdot10^{-3}$
var MCM (HO)[s^2]	$<1\cdot10^{-3}$	$<1\cdot10^{-3}$	$<1\cdot10^{-3}$
var MCM (YTO)[s^2]	$<1\cdot10^{-3}$	$<1\cdot10^{-3}$	$<1\cdot10^{-3}$
MCM based throughput analysis			
avg total (DG) [s]	48	222	$<1\cdot10^{-3}$
avg total (HO) [s]	48	220	$<1\cdot10^{-3}$
avg total (YTO) [s]	48	220	$<1\cdot10^{-3}$
var total (DG) [s^2]	$17\cdot10^{3}$	$97\cdot10^{3}$	$<1\cdot10^{-3}$
var total (HO) [s^2]	$17\cdot10^{3}$	$96\cdot10^{3}$	$<1\cdot10^{-3}$
var total (YTO) [s^2]	$17\cdot10^{3}$	$96\cdot10^{3}$	$<1\cdot10^{-3}$
State-space based throughput analysis			
avg total (SS) [s]	$1.12\cdot10^{-3}$	$4.47\cdot10^{-3}$	$1.7\cdot10^{-3}$
var total (SS) [s^2]	$<1\cdot10^{-3}$	$<1\cdot10^{-3}$	$77\cdot10^{-3}$

solved' of the section 'HSDFG to diagraph conversion').

The measured running times for the MCM algorithms are shown in the section 'MCM algorithms'. The overall required running time using the existing MCM-based approaches is shown in the section 'MCM based throughput analysis' and the running time for our approach is shown in the section 'State-space based throughput analysis'.

We summarize the most important results from the experiments. The results for the set 'mimic DSP' show that our approach solves all problems in about a millisecond each, while the others did not complete 9 problems due to the conversion to the directed graph. The column 'Large HSDFG' in Table 4.2 shows the running times for SDFGs with a large increase in the number of actors when going from the SDFG to the HSDFG. The running time of the existing approaches is strongly impacted by this increase and has grown considerable with respect to the results in the previous set. In contrast, our running times are still in the order of milliseconds. It is further important to note that the set contains 10 SDFGs for which the conversion to an HSDFG does not complete, and another 44 SDFGs for which the combined conversion from the SDFG to a weighted directed graph cannot be completed. The results for the 'Long transient' set confirm our expectations that SDFGs with a long transient phase impact the running times of our algorithm while not influencing the running times of the other algorithms. The effect is visible in the increased variance. However, the running times of our algorithm are on average still in the order of milliseconds.

The conversion to a weighted directed graph is required for MCM analysis and often a bottleneck for analysis as results show. However it is not required for an MCR analysis. Dasdan gives in [16] an MCR formulation of YTO (YTO-MCR). One can argue however that the running time of the YTO-MCR algorithm will always be larger than the running time of the SDFG to HSDFG conversion (which is still required) plus the running time of the YTO algorithm used in our experiments; the graph used in YTO is never larger than the graph used in YTO-MCR. Therefore, we can conclude from the experimental results that also MCR analysis using YTO-MCR will be slower than our state-space exploration method.

Overall, the experiments show that the running time of the existing approaches is greatly impacted by the SDFG to HSDFG conversion. The results of the experiments on the real applications show also that this problem appears frequently in practice. On the other hand, our method tends to have acceptable running times even if it is confronted with adverse graphs. We observe that our method has on average better run times than the existing MCM approaches and it can compute the throughput of all tested SDFGs within milliseconds while the MCM approaches fail to produce results on a substantial number of SDFGs.

4.5 Throughput Analysis of Arbitrary SDFGs

So far, the throughput of an SDFG was defined only for strongly connected SDFGs (Definition 4.1). In this section, we first generalize the definition to an arbitrary SDFG and then we propose a method to calculate its throughput.

The normalized actor throughputs may vary in non-strongly connected SDFGs as opposed to strongly connected graphs in which they are always equal. Since throughput is the indication of the average performance of the whole model, we intuitively generalize the throughput definition by taking the throughput of the slowest strongly connected component as the throughput of the whole SDFG.

Definition 4.2. (Throughput of an Arbitrary SDFG) *The throughput of an arbitrary consistent SDFG is defined as the normalized actor throughput of the actor with minimum normalized actor throughput among all actors in the SDFG. Formally, the throughput of a consistent SDFG* $G = (A, C, E)$ *with repetition vector* q *is,*

$$Th(G) = min_{a \in A} \frac{Th(a)}{q(a)}.$$

Now we need to generalize our approach to be able to calculate the throughput of a non-strongly connected SDFG. First of all, the state-space method does not always work because non-strongly connected SDFGs may be self-timed unbounded. The unboundedness leads to an infinite number of states, implying that the state-space method cannot detect the period. For the purpose of throughput calculation, we use the same reduced graph as used in Section 3.5.2 for checking self-timed boundedness. Definition 3.7 shows the reduction of an arbitrary SDFG to a special kind of HSDFG where every actor has a self-loop channel, and disregarding self-loop channels the resulting HSDFG is acyclic. It has been shown that the reduction preserves certain properties like normalized actor throughput (Proposition 3.4) and self-timed boundedness (Theorem 3.5). The throughput calculation of the resulting HSDFG can be easily done owing to its acyclicity. Let $G_H = (A_H, C_H, E_H)$ be this HSDFG. Considering Definitions 3.7 and 4.2; the throughput of G_H then is

$$Th(G_H) = \min_{x \in A_H} \frac{1}{E_H(x)}.$$

Considering Proposition 3.4 we get the following theorem.

Theorem 4.2. *Given an arbitrary consistent timed SDFG* G. $Th(G) = Th(G_H)$, *with* G_H *the HSDFG of Definition 3.7.*

To compute throughput of a non-strongly connected SDFG, we first apply the conversion of Definition 3.7, and then compute its throughput according to the formula above. The conversion uses the state-based throughput calculation method on all maximal SCCs, to compute the actor execution times in the constructed HSDFG.

4.6 Related Work

Throughput analysis of HSDFGs has been studied extensively [16, 17, 37, 56, 73]. All these studies are applicable to SDFGs only through a conversion to an HSDFG described in Section 2.6. Maximum Cycle Mean (MCM) analysis or Maximum Cycle Ratio (MCR) analysis are then used to determine throughput. Karp proposed an algorithm in [37] which forms the basis for other improved algorithms like [13, 17, 73]. An in-depth comparison of the timing behavior of different MCM (related) algorithms is given in [16, 17]. Behavior of HSDFGs and their throughput can also be analyzed using Max-Plus algebra [3, 13]. In this context, it is refered to as spectral analysis.

Throughput analysis has been studied extensively in the Petri-net literature as well. In [59], an MCM-related analytical method is presented for marked graphs (HSDFGs), which in [12] is extended to specific cases of weighted marked graphs (SDFGs) where all the initial tokens must be only on one channel on each cycle. In [11], using a linear programming approach, lower and upper bounds on throughput of a certain class of Petri nets are given. The upper bound is exact for marked graphs, and a conversion from weighted marked graphs to marked graphs similar to the conversion of SDFGs to HSDFGs is used to calculate throughput for weighted marked graphs.

Unlike all previous approaches, this thesis proposes a technique based on explicit state-space exploration for finding the throughput, which directly works on SDFGs. Because of this, we save the extra step for explicitly converting an SDFG to an HSDFG, which can be exponentially larger.

4.7 Summary

We have introduced a new approach to throughput analysis of Synchronous Data Flow Graphs. Existing methods for throughput analysis include a transformation to Homogeneous Data Flow Graphs and suffer from an exponential blowup in the number of graph nodes, which makes the approaches fail in certain cases. Our approach is based on explicit state-space exploration and avoids the translation to HSDFGs. We have studied properties of the state-space and derived a method for computing throughput based on the state-space. We have shown that the state-space-based definition of throughput corresponds to the classical definitions in terms of Maximum Cycle Mean of the equivalent HSDFG and the eigenvalue of the corresponding Max-Plus matrix equation. Experiments show that our throughput analysis method performs significantly better in practice than existing approaches which is an important enabler for extensive design-space explorations.

Chapter 5

Parametric Throughput

5.1 Overview

Throughput of SDFGs has been discussed in the previous chapter as the most prominent performance metric. Throughput is a crucial indicator of performance used both at design time (e.g., in design-space exploration, DSE) and run-time (e.g., resource management). In DSE many different settings of the system are explored [61, 62], which leads to many throughput calculations. At run-time, prediction of throughput is required for proper assignment of resources to applications during reconfigurations [54]. In both cases, throughput calculations need to be as fast as possible with very strict time and resource requirements for run-time applications.

Previously, throughput analysis has been studied with the assumption of fixed numbers for the execution times. Therefore, any change in the execution time of one or more actors of an SDFG leads to a recomputation of the throughput from scratch.

In this chapter, we consider parametric SDFGs, a generalization of SDFGs where actors can have parameters as their execution times. As a result, throughput is specified in the form of a simple function of parameters, where the resulting function gives the throughput of the SDFG for any value in the range of the parameters. Such a function can be used in DSE, or at run-time for resource or quality management. It is an important enabler for a scenario-based design approach [32] aiming at throughput predictability. Another application of having parametric throughput is for example the study of the impact on throughput of variation of execution times under production process variations [22].

We study three algorithms to calculate the throughput of a parametric SDFG. The first two algorithms are variants of the standard throughput analysis algorithms based on MCM analysis of HSDFGs and the state-space method explained in the previous chapter for SDFGs with parametric actor execution times. The

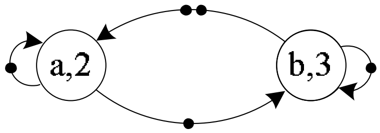


Figure 5.1: An example SDFG G_{ex}.

third algorithm is based on a divide-and-conquer (DC) strategy. In the experimental results, we compare the advantages and the drawbacks of these algorithms. The DC algorithm turns out to be the most efficient in practice.

Section 5.2 briefly explains different methods of throughput analysis and it introduces parametric synchronous data flow graphs. Sections 5.3, 5.4 and 5.5 explain the different methods for finding the throughput of parametric SDFGs, initially focussing on strongly connected graphs. Comparison of the methods is done in Section 5.6. Section 5.7 extends the results to arbitrarily connected graphs. Section 5.8 discusses the related work and finally Section 5.9 summarizes. This chapter is based on publication [26].

5.2 Parametric SDFGs and Throughput Analysis Methods

In this section, we first briefly summarize the different throughput analysis methods that we discussed in the previous chapter. We also define parametric SDFGs. Then we explain in general, how these throughput analysis methods can be used for analyzing the throughput of parametric SDFGs. For brevity, we focus on strongly connected SDFGs. The extension of the results to general graphs can be done by combining the results of the strongly connected components of the SDFG which we explain at the end of the chapter.

Generally speaking, there are two different methods for calculating throughput of an SDFG.

HSDFG method: In Chapter 4, we showed that the throughput of an SDFG is equal to the inverse of the maximum cycle mean (MCM) of the equivalent HSDFG [60]. The main problem of this method was that the conversion of an SDFG to an equivalent HSDFG may lead to an exponential explosion in the size of the graph.

State-space method: This method was explained in detail in Chapter 4, Self-timed execution of an SDFG ends in a repetitive sequence of actor firings, the periodic phase of execution (Proposition 4.1). The throughput of an actor can be calculated by dividing the length of the period by the number of firings of the actor in one period.

Throughout this chapter, we use a simple timed HSDFG G_{ex} depicted in Figure 5.1 as our running example. G_{ex}contains two actors, a and b, with execution times 2 and 3 respectively.

So far in previous chapters, actor execution times were always fixed numbers. Here we introduce a *parametric SDFG*, as an SDFG where the execution time of at least one of its actors is a parametric expression, where a parametric expression is a linear expression in terms of some parameters. In principle, parametric expressions can take any value of positive real numbers. For example, G_{ex} of Figure 5.1 becomes a parametric SDFG, G_{ex}^{par}, if we assume that the execution times of a and b are given by parameters $p, q \in \mathbb{R}^+$. We are interested in the throughput of a parametric SDFG in the form of a function of the parameters: $f(p, q)$. The domain of this function, the set of values that the parameters can take, is called the *parameter space* of the graph, which is d-dimensional when the number of parameters is d. Evaluating this function for a point in the parameter space is computationally much cheaper than redoing any of the traditional throughput calculations for those parameter values. We formally define the parametric execution times in the following definition.

Definition 5.1. (PARAMETRIC EXECUTION TIME) *Assume a d-dimensional vector of parameters. A parametric execution time models the execution duration of actors of a parametric SDFG, in terms of these parameters. In a parametric SDFG (A, C, E_p), (A, C) is an SDFG and the execution time is a function $E_p : A \rightarrow T$ that associates with each actor $a \in A$ a linear combination of parameters, formally denoted by a vector t in $T = \mathbb{Q}^{d+1}$ containing the coefficients of the linear expression. For $a \in A$, $E_p(a)$ assigns the amount of time it takes to complete a firing of actor a.*

We know that the throughput of an SDFG corresponds to the inverse of the maximum cycle mean of its equivalent HSDFG. The cycle mean of each cycle equals the sum of the execution times of actors in the cycle divided by the number of tokens on the cycle. Consequently, any cycle mean in a parametric HSDFG is a linear combination of parameters plus a constant, representing the non-parameterized actors in the cycle, which is 0 when there are no such actors. We call these linear combinations *cycle mean expressions*. $\frac{1}{3}p + \frac{1}{3}q + 0$ is a cycle mean expression of G_{ex}^{par}corresponding to the cycle through a and b. A cycle mean expression is represented by a vector $\overline{e}$ whose elements are the coefficients of the linear expression, e.g., $(1/3, 1/3, 0)$ in the example. If cycle c has cycle mean expression $\overline{e}_c$, then its cycle mean, λ_c, for each point $\overline{p} \in \mathbb{R}^d$ in the parameter space can be calculated by $\lambda_c(\overline{p}) = \overline{e}_c \cdot (\overline{p}, 1)$, where "$\cdot$" is the inner product of two vectors. For example, for G_{ex}^{par}, $\lambda(1, 2) = (1/3, 1/3, 0) \cdot (1, 2, 1) = 1$. We denote the evaluation of a point $\overline{p}$ in a cycle mean expression $\overline{e}_c$, by $\overline{e}_c(\overline{p})$. The maximum cycle mean of an HSDFG G for each point $\overline{p}$ in the parameter space, denoted by $\lambda^*(\overline{p})$, can be calculated via

$$\lambda^*(\overline{p}) = \max_{c \in cycles(G)} \overline{e}_c(\overline{p}).$$

Note that λ^* is a continuous function as it is the composition of continuous functions max and $\overline{e}_c$. Any expression that has the maximum cycle mean value for some point is called a maximum cycle mean expression (*mcme*). Any mcme that has the maximum cycle mean value for some point for which no other mcme has the maximum value is a *dominating mcme*. It is shown below that the dominating mcmes are sufficient to compute λ^*. The other cycle mean expressions, including the other mcmes, are called *redundant expressions*.

Definition 5.2. (DCMS) *Given an HSDFG, the dominating cycle mean set (DCMS) is the set of dominating mcmes.*

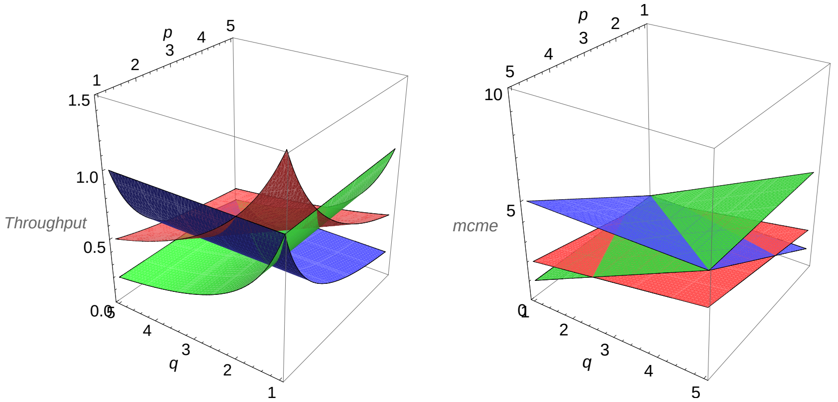
Proposition 5.1. *Given an SDFG G and a parameter space, the maximum cycle mean of G for any value of parameters can be obtained by its DCMS.*

$$\lambda^*(\overline{p}) = \max_{\overline{e} \in DCMS} \overline{e}(\overline{p}).$$

Proof. Suppose the parameter space has d dimensions. For any arbitrary point $\overline{p}$ in the parameter space, if only one mcme is maximum at $\overline{p}$ then that mcme is in DCMS by definition. If only two mcmes $\overline{e}_1$ and $\overline{e}_2$ are maximum at $\overline{p}$, then $\overline{p}$ can only lie in the plane characterized by $\overline{e}_1(\overline{p}) = \overline{e}_2(\overline{p})$ which has less than d dimensions as the mcmes are linear combinations of parameters. In other words, $\overline{p}$ is on the border of the regions in parameter space for which $\overline{e}_1$ or $\overline{e}_2$ are mcmes. So, there is a region around $\overline{p}$ in the parameter space for which either $\overline{e}_1$ or $\overline{e}_2$ is uniquely maximum. Therefore, at least one of $\overline{e}_1$ or $\overline{e}_2$ belongs to the DCMS. If more than two mcmes are valid at $\overline{p}$, we can use a similar argument to show that at least one of them belongs to the DMCS. We showed that for any point $\overline{p}$ in the parameter space at least one dominating mcme belongs to the DCMS. □

Note that when we talk about the DCMS of an SDFG we refer to the DCMS of its equivalent HSDFG. Conversion of a parametric SDFG to an equivalent HSDFG can be done using the algorithm for non-parametric graphs, since execution times have no impact on the conversion algorithm.

Thus, throughput analysis for a parametric SDFG can be done by finding its DCMS. This minimum set can be obtained from the set of all expressions/mcmes by removing all redundant expressions. Checking the redundancy of an expression is equivalent with checking the infeasibility of a system of linear equations [21]. However, since in our case the values of parameters are positive, there is a fast way to remove a large part of the redundant expressions. An expression is redundant if all of its coefficients are less than or equal to those of another expression. In other words, if we look at the vectors $\overline{e}$ of the expressions, then all points (expressions) are dominated by a subset of points with larger coefficients. Those points which are not dominated by other points using this redundancy check are called pareto points. In other words, pareto points are dominating all non-pareto points. We denote $\overline{e}_1 \preccurlyeq \overline{e}_2$ to express that $\overline{e}_1$ is dominated by $\overline{e}_2$. A pareto dominance test is much easier than the general redundancy checks. Although finding the pareto set of expressions often removes a large part of the redundant expressions, the pareto set is not necessarily the DCMS. For example, suppose our set of expressions is $\{p, q, (p+q)/3\}$, or, in terms of vectors $\{(1,0,0),(0,1,0),(1/3,1/3,0)\}$. Even though $(1/3,1/3,0)$ is a pareto point (it is not dominated by either $(1,0,0)$ or $(0,1,0)$), there is no point in the parameter space where $(p+q)/3$ has a larger value than all other expressions, making it redundant. Nevertheless, pruning the set of expressions via a pareto dominance test before applying any general redundancy test is worthwhile. Figures 5.2(a) and 5.2(b) visualize this set of throughput expressions and their related mcmes for parameter ranges of $[1,5]$. The horizontal axes are parameters and the vertical axes give throughput and mcme values for Figures 5.2(a) and 5.2(b) respectively. As it can be seen from the figure, the red plane (expression $(p+q)/3$) is always dominated by either green (p) or blue (q) planes for any point in the parameter space (similar for the throughput expressions in Figure 5.2(a)).



(a) Throughput expressions (b) mcmes

Figure 5.2: mcmes and related throughput expressions of the running example.

Existing methods of calculating throughput for SDFGs are not directly applicable to parametric SDFGs. The efficient MCM analysis algorithms which work on HSDFGs cannot be applied on parametric SDFGs. The conversion of SDFGs to HSDFGs can easily be adapted though. Therefore, a naive MCM analysis leads to enumerating all simple cycles of the HSDFG and collecting the expressions in the DCMS. Also, the state-space method, explained in the previous chapter, cannot be directly used for parametric throughput analysis, but it can be generalized. In the remainder, we introduce two variations of the existing methods and one new method for calculating the throughput of a parametric SDFG.

5.3 HSDFG Method

This section shows how a parametric throughput can be calculated using the conversion of an SDFG to an HSDFG. The MCM of an HSDFG can be found by enumerating all simple cycles. The cycle mean of each cycle can be calculated by summing up all of the execution times of actors in the cycle and dividing it by the number of tokens on the cycle. Finally, the DCMS of the parametric HSDFG is obtained by removing the redundant expressions as explained in Section 5.2. While enumerating the cycles and calculating their cycle mean expressions only the pareto points are kept.

The example in Figure 5.1 is already an HSDFG, so no conversion is needed. This graph has three simple cycles (a, a), (b, b) and (a, b, a) with one, one and three tokens respectively. The corresponding cycle mean expressions are p, q and $(p + q)/3$. Since all expressions are pareto points no points get eliminated in the pareto test.

In the next step of the algorithm, we see that $(p + q)/3$ is redundant. This follows from the infeasible set of linear equations $\{(p + q)/3 > p, (p + q)/3 > q\}$. Therefore, $DCMS(G_{ex}^{par}) = \{p, q\}$ and $\lambda^*(p, q) = \max\{p, q\}$.

```
Algorithm HSDFG Method(G)
Input: A parametric SDFG $G$
Output: DCMS of $G$
1.  $DCMS = \varnothing$
2.  Convert $G$ to equivalent HSDFG $H$
3.  for each simple cycle $c$ in $H$
4.      do if $\overline{e}_c \not\preccurlyeq \overline{e}_i$ for all $\overline{e}_i \in DCMS$
5.            then remove all $\overline{e}_i$ from $DCMS$ for which $\overline{e}_i \preccurlyeq \overline{e}_c$
6.                 insert $\overline{e}_c$ in $DCMS$
7.  Remove redundant expressions from $DCMS$
8.  return $DCMS$
```

Note that finding a set of dominating expressions among expressions (Line 7 of the algorithm) has been solved efficiently in the context of determining the upper envelope of pairwise linear functions [19]. Since the time spent on this part of the algorithm is negligible compared to the first part, we used the straightforward redundancy check explained above in our experiments using a linear programming C library (LPsolve [5]).

5.4 State-Space Method

State-space-based throughput calculation for SDFGs avoids the conversion to HSDFGs. This section generalizes the state-space based method to calculate the throughput of a parametric SDFG.

5.4.1 A Parametric State Space

In Chapter 2, we explained that the behavior of an SDFG can be defined in terms of a state transition system. For example, in Figure 5.1, the initial state is $((1,1,1,2),(\{\},\{\}))$, where the first vector, γ, shows the token distribution of channels, starting from the self-loop channel of a and continuing counterclockwise. The second vector, υ, is the vector of multisets of the remaining execution times of a and b. Initially, both multisets are empty. At this point, both a and b are enabled and they start their firings, changing the token distribution vector γ to $(0,0,0,1)$ and the vector of remaining execution times υ to $(\{2\},\{3\})$. No more actor firings can occur before actor a finishes. So the time goes forward for 2 time units. Completing the firing of a leads to state $((1,1,0,1),(\{\},\{1\}))$. Firing actor a once again then results in $((0,1,0,0),(\{2\},\{1\}))$, after which time progresses for 1 time unit, b completes its firing, and so on.

We generalize this model to parametric SDFGs. In the state space of a parametric SDFG, υ contains parametric elements, expressions in terms of the parameters. Since the relations between parameters are not known, we cannot always be sure which firing finishes first. The next example shows how we solve this problem.

The parametric state space of G_{ex}^{par} with execution times p and q for actors a and b is given in Figure 5.3. To simplify the figure, the details of states are not shown. Each dot represents a state. The start and end of firings in each state is denoted by the actor name with subscript s or e respectively. For example a_s shows the start of a firing of actor a.

After starting firings of a and b, υ changes to $(\{p\},\{q\})$. At this stage, a time step equal to the smallest among all elements in the multisets of υ must be taken, but the relation between p and q is unknown. Therefore, we split the parameter space into two mutually exclusive parts with $p < q$ and $p > q$. For each of these parts, the state space continues in a separate branch. Since our final goal is finding λ^* and since λ^* is continuous, we do not need to consider the case $p = q$ as the cycle means of this part of the parameter space are covered by expressions obtained by both the cases $p < q$ and $p > q$.

In case $p > q$, the first vertical arrow in the figure, after a time step as large as q, b finishes its firing. So γ and υ become $(0,0,1,2)$ and $(\{p-q\},\{\})$. The execution proceeds by a time step as large as $p-q$ which leads to the end of the firing of a and consequently the start of new firings of actors a and b, changing υ to $(\{p\},\{q\})$. Since in this branch we already assumed that $p > q$, no new partitioning of the parameter space is needed and the execution proceeds by a time step as large as q. The state space in this branch ends in a periodic phase repeating the last two steps. From the periodic phase, we can compute the throughput. The length of the period is $(p-q)+q = p$ and during the period only one firing of actors a and b occurs. Therefore, the throughput is $1/p$ if $p > q$.

We proceed for the case where $p < q$. After a time step as large as p, actor a finishes its firing and starts a new firing. So γ and υ become $(0,1,0,0)$ and

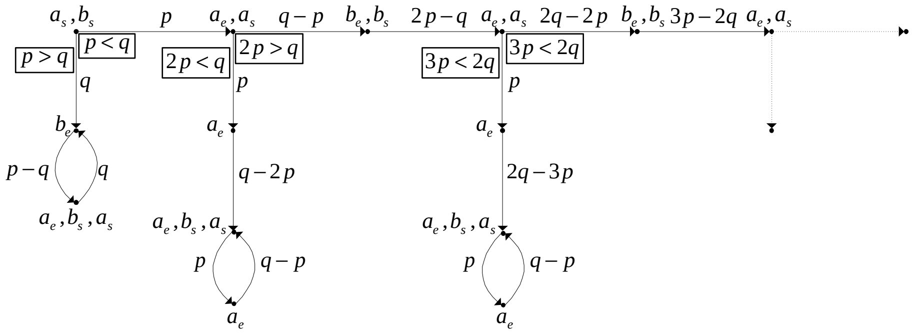


Figure 5.3: The parametric execution of G_{ex}^{par}.

$(\{p\}, \{q-p)\})$ respectively. At this state, the parameter space needs to be split again into two parts: $p < q-p$ and $q-p < p$ (or $2p < q$ and $2p > q$). Note that the state-space exploration only continues if the newly added constraints do not conflict with the previously made assumptions in the earlier states. In this case, both $2p < q$ and $2q < p$ are compatible with $p < q$. The case $2p < q$ gets periodic in a few steps with throughput $1/q$. The other case requires a new partitioning of the parameter space. Each branch continues till either it ends up in its periodic phase or the constraint set contains conflicting constraints. As shown in the figure, the state space of the example continues to repeat a similar pattern. All subsequent branches have the same throughput $1/q$. We can conclude that, as before, $DCMS(G_{ex}^{par}) = \{p, q\}$ for the whole parameter space.

From the example, we can see that the multisets in υ of remaining execution time expressions contain linear combinations of parameters throughout the execution of the graph. We also observe that the equations partitioning the parameter space need to be stored in the states.

Definition 5.3. (PARAMETRIC STATE) *The state of a parametric SDFG* (A, C, E_p) *is a tuple* (γ, υ, Φ). γ*, as in a regular state a channel state, associates with each channel the amount of tokens present in that channel in that state. Actor status* $\upsilon : A \to \mathbb{N}^T$ *associates with each actor* $a \in A$ *a multiset of linear combinations of parameters, each such combination is denoted by a vector* $\overline{t}$ *in* $T = \mathbb{Q}^{d+1}$ *containing the coefficients of the linear expression* $\overline{t}(\overline{p})$*. The state constraint set* Φ *is a subset of* T*, which contains all of the assumptions,* $\overline{t}(\overline{p}) > 0$*, in the form of inequalities on the parameters made so far.*

Similar to timed SDFGs, the dynamic behavior of a parametric SDFG is described by transitions that can be of any of three forms: start of actor firing, end of firing, or time progress. These transitions are direct translations of the concrete transitions of Definition 2.12 into parametric form.

Definition 5.4. (TRANSITIONS) *A transition of a parametric SDFG* (A, C, E_p)

from state $(\gamma_1, \upsilon_1, \Phi_1)$ *to state* $(\gamma_2, \upsilon_2, \Phi_2)$ *is denoted by* $(\gamma_1, \upsilon_1, \Phi_1) \xrightarrow{\alpha} (\gamma_2, \upsilon_2, \Phi_2)$ *where label* $\alpha \in (A \times \{start, end\}) \cup (\{clk\} \times T \times 2^T)$ *denotes the type of the transition.*

- *Label* $\alpha = (a, start)$ *corresponds to the firing start of actor* a*. This transition is enabled if* $Rd(a) \preceq \gamma_1$ *and results in* $\gamma_2 = \gamma_1 - Rd(a)$*,* $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \uplus \{E_p(a)\}]$*, and* $\Phi_2 = \Phi_1$*.*
- *Label* $\alpha = (a, end)$ *corresponds to the firing end of* a*. This transition is enabled if* $\overline{0} \in \upsilon_1(a)$ *and results in* $\gamma_2 = \gamma_1 + Wr(a)$ *and* $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \backslash \{\overline{0}\}]$ *(where* $\overline{0}$ *is a vector with all coefficients equal to zero) and* $\Phi_2 = \Phi_1$*.*
- *Label* $\alpha = (clk, l_j, \phi_j)$ *denotes a clock transition.* $l_j \in T$ *is a remaining execution time of an actor in* υ_1*, which is a vector denoting a linear combination of parameters and a constant number, and it specifies the length of the clock transition. Suppose* $l_i, 1 \leq i \leq k$*, are all the remaining execution times of ongoing actor firings in* υ_1*. For every* $l_j, 1 \leq j \leq k$ *in the range of* υ_1*,* ϕ_j *is a set of inequalities* $l_j < l_i$*, for all* i *with* $1 \leq i \leq k, i \neq j$*. In this way,* $\phi_j \subseteq T$ *is a set of assumptions imposing that* l_j *to be the smallest element of* υ_1*.*

  A clock transition is enabled only if no end transition is enabled and $\Phi_1 \cup \phi_j$ *is feasible. The clock transition results in* $\gamma_2 = \gamma_1$*,* $\upsilon_2 = \{(a, \upsilon_1(a) \ominus l_j) \mid a \in A\}$ *with* $\upsilon_1(a) \ominus l_j$ *a multiset of vectors containing the elements of* $\upsilon_1(a)$ *reduced by* l_j *(subtracting* l_j *from each of the vectors) and* $\Phi_2 = \Phi_1 \cup \phi_j$*.*

In Figure 5.3, the horizontal branches of the state space continue to partition the parameter space into ever smaller pieces ad infinitum. The further splitting of the parameter space does not result in the infeasibility of the system of inequalities. This shows that the state space can be infinite.

5.4.2 Throughput Calculation

Algorithm *coverStateSpace* shows the state-space method for the throughput calculation of a parametric SDFG. It works recursively in a depth-first-search fashion, branching the parameter space as explored. Below we assume that parameter ranges are bounded. The algorithm receives a parametric SDFG G, initial state $s = (\gamma, \upsilon, \Phi)$, and a bound on the depth of the recursion D as arguments and returns the DCMS of G. The latter is needed since a parametric state space is potentially infinite. Upon reaching the bound on the depth, the remainder of the parameter space is searched via the divide-and-conquer algorithm explained in the next section. D can thus be used as a control parameter to steer the algorithm towards a (mostly) state-space-based method or a divide-and-conquer method. We use the absence of integer points in the remainder of the parameter space to be explained as an extra criterion to switch to the divide-and-conquer method, as this avoids spending a lot of time in searching increasingly smaller parts of the parameter space.

Algorithm *coverStateSpace*(s, G, D)
Input: A parametric state $s = (\gamma, \upsilon, \Phi)$
Input: A strongly connected parametric SDFG $G = (A, C, E_p)$
Input: A bound D on the recursive depth
Output: DCMS of G

```
if ($D = 0$)
  then Divide&Conquer($G, \Phi$);
  else for all $t \in \cup_{a \in A} \upsilon(a)$
         do $n = (\gamma_n, \upsilon_n, \Phi_n)$ = nextState($t, s, G$);
            if ($n \neq INT\_INF$ and not $BS$)
              then if ($n \in nBList$)
                     then $\overline{e}$ = calcCMExp($nBList, n$);
                          insert($DCMS, \overline{e}$);
                     else push($nBList, n$);
                          coverStateSpace($n, G, D - 1$);
              else if ($n \neq INT\_INF$)
                     then reset($nBList$);
                          coverStateSpace($n, G, D - 1$);
                     else if ($n \neq INF$)
                            then Divide&Conquer($G, \Phi_n$);
```

The algorithm first checks if the maximum execution depth has been reached. If so, it calls Algorithm *Divide&Conquer* of the next section; if not, it uses procedure nextState which accepts parameter expression t (an element of a multiset in υ) as a clock step, thereby assuming that t is the minimum time that should elapse before any event can occur. Then, nextState returns the next state $n = (\gamma_n, \upsilon_n, \Phi_n)$ if Φ_n contains integer solutions within the given parameter bounds. For cases where Φ_n lacks integer solutions or it has no solution at all *INT_INF* and *INF* are returned respectively. nextState also marks the current state as a branching state (BS) or a non-branching state depending on whether the parameter space is split.

If the algorithm is invoked for state s, for every element $t \in \cup_{a \in A} \upsilon(a)$, a new branch is explored, the procedure nextState is called and the new state $n = (\gamma_n, \upsilon_n, \Phi_n)$ is created. We know that none of the states in a periodic phase can be branching since the constraints in the sets of the recurrent states should be identical. Therefore, the search for a recurrent state only occurs in *nBList*, which stores the non-branching states visited since the last branching state. If Φ_n is feasible and n is non-branching, then the algorithm checks whether the state is recurrent (has already been visited) by comparing it with already stored non-branching states in *nBList*. If the state is recurrent, the algorithm calculates the cycle mean expression, by calling calcCMExp. This function finds the length of the periodic phase $|P|$ by adding up the length of clock transitions between the two recurrent states and counts the number of firings $|a|_p$ of an arbitrary actor a

in the periodic phase. The throughput can be calculated as $|a|_p/(|P|q(a))$ where q is the repetition vector of the graph. The obtained expression, which is always a dominating mcme, is stored in DCMS. Then the algorithm returns and continues at the last stored branching state if any is left. If n is not recurrent, it is stored in *nBList* and the algorithm is invoked recursively for n, decrementing bound D.

In case n is a branching state and integer solutions are left, the algorithm is invoked recursively after clearing *nBList*. If Φ_n is not infeasible but contains no integer solutions, the *Divide&Conquer* algorithm that is explained in the next section is called. This algorithm finds the mcmes of all points in the region Φ_n.

Theorem 5.1. *Given a parametric SDFG G and parameters with bounded ranges, algorithm coverStateSpace finds all the dominating mcmes in the parameter space of the parameters of G.*

Proof. We can visualize the parametric state space as a tree with the initial state as its root and leaves which are either recurrent states, or states with depth D or states whose constraint sets lack integer solutions. Algorithm *coverStateSpace* is a depth-first traversal algorithm on this tree, where leaf detections are at Lines 1, 6, and 14 for detecting states with maximum recursive depth, recurrent states, and states whose constraint sets lack integers solution respectively. If a recurrent state is detected, then an mcme is found on the periodic phase of that branch which is valid for the region specified by the constraint set of the states. In case of a leaf without integer parameter solutions or with maximum depth, Algorithm *Divide&Conquer* whose correctness is proven in Theorem 5.2 is called to cover points in the remaining region of the parameter space. Note that the algorithm covers all the parameter space because at each branching state $bs = (\gamma_b, \upsilon_b, \Phi_b)$ with successor states $(\gamma_i, \upsilon_i, \Phi_i), i = 1 \ldots k$ when k states are feasible, then Φ_is partition Φ_b except for the borders of Φ_is. The borders are covered by expressions found for the interior, as λ^* is continuous. □

5.4.3 Discussion

In Section 5.4.1 we showed that the parametric state space of our running example SDFG of Figure 5.1 is infinite. Algorithm 4 assumes two stop criteria to avoid an unbounded number of recursive calls. The first criterion is achieved by restricting the depth of the recursion which is specified by input parameter D of the algorithm. The other criterion holds whenever a branch reaches a state whose constraint set lacks integer solutions. The reason that we have the latter criterion in the algorithm is that we conjecture that we can find the recurrent state of the branch related to any point in the parameter space, in a finite number of steps. The intuition behind this conjecture is that every branch of a parametric state space can be translated to a non-parametric state space. Every parametric state (γ, υ, Φ) of the parametric state space can be translated into a state $(\gamma, \upsilon_{\overline{p}})$ in the non-parametric state space, where $\upsilon_{\overline{p}}$ is a multiset whose parametric remaining execution times of ongoing actors are replaced by their evaluation for $\overline{p}$. The execution of both parametric and non-parametric state spaces are defined based on the same semantics. Furthermore, according to Proposition 4.1, we know that the non-parametric state space is periodic for consistent and strongly connected SDFGs, implying that the parametric state space is also periodic for consistent and strongly connected SDFGs for any concrete point in the parameter space. Our experiments are also in line with this conjecture as the recursion depth stopping criterion never executed.

In fact, if the conjecture holds, and if we limit our interest to integer parameter values, we can remove the depth criterion and stop when a branch in the parametric state space reaches a state whose constraints set lacks integer solutions. Note that the restriction to integer values is not a limitation in practice, because the ranges can be chosen such that the actor execution times are arbitrarily precise. In this way, we can also have a pure state-space method if it assumes integer parameter values. This version of the state-space method stops either when it reaches a recurrent state, in which case, it finds the related mcme of the related points of that branch, or it ends if it reaches a state whose constraint set lacks integer solutions. In this case, there is no integer solution in the interior of the region specified by the constraint set of the state. The mcme of the integer points on the border of these regions can also be found by applying the state-space method for concrete points as proposed in Chapter 4 individually on each of them.

5.5 Divide-and-Conquer Method

If we have a closer look at the parts of the parameter space that share the same mcme, we observe that these parts form convex polyhedra. In this section, using this fact, we propose a divide-and-conquer algorithm to find these regions as well as their related mcmes.

Proposition 5.2. $\{\overline{p} \in I\!R^d \mid \lambda^*(\overline{p}) = \overline{e}(\overline{p})\}$ *is a convex polyhedron for any mcme* $\overline{e}$.

Proof. To show that the throughput region associated with $\overline{e}$ is convex, we need to prove that any point $\overline{p} = t\overline{p}_1 + (1-t)\overline{p}_2$ where $0 < t \leq 1$ belongs to the throughput region of $\overline{e}$, i.e., $\overline{e}(\overline{p}) = \lambda^*(\overline{p})$ for every two points $\overline{p}_1$ and $\overline{p}_2$ belonging to the throughput region of $\overline{e}$.
Because $\overline{e}$ is a linear function, if $\overline{e}(\overline{p}_1) = \lambda^*(\overline{p}_1)$ and $\overline{e}(\overline{p}_2) = \lambda^*(\overline{p}_2)$, we have

$$\overline{e}(t\overline{p}_1 + (1-t)\overline{p}_2) = t\overline{e}(\overline{p}_1) + (1-t)\overline{e}(\overline{p}_2). \tag{5.1}$$

According to the definitions $\overline{e}(\overline{p}_1) = \lambda^*(\overline{p}_1) = \max_{\overline{e}_i \in DCMS} \overline{e}_i(\overline{p}_1)$ and $\overline{e}(\overline{p}_2) = \lambda^*(\overline{p}_2) = \max_{\overline{e}_i \in DCMS} \overline{e}_i(\overline{p}_2)$, for all $\overline{e}_i \in DCMS$, we can write

$$\begin{array}{rcl} \overline{e}_i(\overline{p}_1) & \leq & \overline{e}(\overline{p}_1) \\ \overline{e}_i(\overline{p}_2) & \leq & \overline{e}(\overline{p}_2). \end{array}$$

Therefore, for all $\overline{e}_i \in DCMS$,

$$t\overline{e}(\overline{p}_1) + (1-t)\overline{e}(\overline{p}_2) \geq t\overline{e}_i(\overline{p}_1) + (1-t)\overline{e}_i(\overline{p}_1)$$

which means that

$$\overline{e}(\overline{p}) = \max_{\overline{e}_i \in DCMS} \overline{e}_i(\overline{p}) = \lambda^*(\overline{p}).$$

□

We call these convex polyhedra *throughput regions.* Figure 5.4 shows two throughput regions for the running example, corresponding to $\overline{e}_1 = q$ and $\overline{e}_2 = p$ within a rectangular area between corner points $\overline{v}_1, \ldots, \overline{v}_4$. The following corollary directly follows from Proposition 5.2.

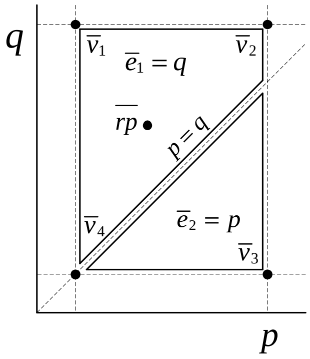


Figure 5.4: Divide-and-Conquer Method.

Corollary 5.1. *If for every corner point $\overline{v}$ of an arbitrary polyhedron of the parameter space, $\lambda^*(\overline{v}) = \overline{e}(\overline{v})$ for some mcme $\overline{e}$, then for every point $\overline{p}$ in that region, $\lambda^*(\overline{p}) = \overline{e}(\overline{p})$.*

Hence, the parameter space is composed of throughput regions. Suppose $\overline{e}$ is the mcme of an arbitrary interior point of a convex polyhedron C. By comparing the evaluation of $\overline{e}$ for every vertex (corner point) of C and the actual maximum cycle mean value of that vertex, we can detect whether C is a subset of a single throughput region or whether it is covered by parts of more regions. If for any vertex, these two compared values are different, then C is covered by more than one throughput region; otherwise it is part of a single throughput region (namely that of dominating mcme $\overline{e}$). This idea can be used in a divide-and-conquer method if we add a partitioning strategy, to be applied after detecting a region with more than one mcme. Partitioning continues till all the created regions have a single mcme. All obtained mcmes together form the DCMS.

Since λ^* is continuous, the mcmes of two neighboring regions are valid for all points on the border of the regions. This means that for any two neighboring regions with mcmes $\overline{e}_1$ and $\overline{e}_2$, their border is characterized by the equation $\overline{e}_1(\overline{p}) = \overline{e}_2(\overline{p})$. We address this (hyper) plane as the *splitting plane*; in Figure 5.4, this is the line $p = q$. In other words, if we have two mcmes for two neighboring regions, we can directly calculate the border of the two regions. The following proposition shows that a splitting plane obtained from two different mcmes of a convex region always passes through the region and splits the region into smaller ones.

Proposition 5.3. *Let $\overline{e}_1$ and $\overline{e}_2$ be mcmes associated to points $\overline{p}_1$ and $\overline{p}_2$ respectively. If $\overline{e}_1(\overline{p}_1) \neq \overline{e}_2(\overline{p}_1)$ and $\overline{e}_1(\overline{p}_2) \neq \overline{e}_2(\overline{p}_2)$, then $\overline{e}_1(\overline{p}_1) - \overline{e}_2(\overline{p}_1) > 0$ and $\overline{e}_1(\overline{p}_2) - \overline{e}_2(\overline{p}_2) < 0$.*

Proof. Considering the definition of mcme for a point, we have

$$\begin{aligned} \overline{e}_1(\overline{p}_1) &= \max_{\overline{e}_i \in DCMS} \overline{e}_i(\overline{p}_1) \\ \overline{e}_2(\overline{p}_2) &= \max_{\overline{e}_i \in DCMS} \overline{e}_i(\overline{p}_2) \end{aligned}$$

Therefore, since $\overline{e}_1(\overline{p}_1) \neq \overline{e}_2(\overline{p}_1)$ and $\overline{e}_1(\overline{p}_2) \neq \overline{e}_2(\overline{p}_2)$, $\overline{e}_1(\overline{p}_1) > \overline{e}_2(\overline{p}_1)$ and $\overline{e}_2(\overline{p}_2) > \overline{e}_1(\overline{p}_2)$. Therefore,

$$\begin{aligned} \overline{e}_1(\overline{p}_1) - \overline{e}_2(\overline{p}_1) &> 0 \\ \overline{e}_1(\overline{p}_2) - \overline{e}_2(\overline{p}_2) &< 0 \end{aligned}$$

□

Geometrically speaking, Proposition 5.3 means that the plane characterized by equation $\overline{e}_1(\overline{p}_2) - \overline{e}_2(\overline{p}_2) = 0$ intersects the line connecting $\overline{p}_1$ and $\overline{p}_2$; in other words, $\overline{p}_1$ and $\overline{p}_2$ are on the opposite sides of the plane, and the plane splits the convex region into two smaller and again convex regions. Using Proposition 5.3 and Corollary 5.1 we have our complete algorithm if we can find the mcme of a point in the parameter space. This is achieved by adapting the state-space exploration of Section 5.4. The difference with the generic parametric state-space method is that the evaluation of the expressions in the constraint set is known when searching for an mcme for a concrete point and no branching is required.

Some points in the parameter space may have more than one mcme, if different HSDFG cycles happen to be simultaneously critical. In that case, we may get an expression from this method that does not correspond to any real cycle mean expression of the graph because it contains fragments of different cycles. However, our partitioning strategy only works if the expressions relate to real cycle means. So we need to avoid obtaining such expressions. We show that this is a 'coincidence', that only happens on the border of throughput regions and can be avoided by selecting a random point from the parameter space.

Proposition 5.4. *A randomly selected point from the parameter space has only one mcme with probability one.*

Proof. Let $\overline{p}$ be a randomly selected point in the parameter space. If two different $\overline{e}_1$ and $\overline{e}_2$ are both valid mcmes for $\overline{p}$, then the evaluation of both $\overline{e}_1$ and $\overline{e}_2$ for $\overline{p}$ must be equal. Therefore, we have

$$\overline{e}_1(\overline{p}) = \overline{e}_2(\overline{p}).$$

Suppose we denote the jth coefficient of expression $e_i, i = 1, 2$, with $\overline{e}_{i,j}$ and of $\overline{p}$ with $\overline{p}_j$. Now, knowing that the constants in $\overline{e}_1$ and $\overline{e}_2$ must be equal, we can rewrite the above equation in the following form.

$$\sum_{j=1}^{d} (\overline{e}_{1,j} - \overline{e}_{2,j})\overline{p}_j = 0$$

We know that $\overline{p}$ is randomly selected from a uniformly continuous distribution. Therefore, $\sum_{j=1}^{d} (\overline{e}_{1,j} - \overline{e}_{2,j})\overline{p}_j$ is also a random variable in a continuous space. The probability of a continuous random variable being a constant number is zero. Therefore, with probability one, $\overline{e}_1(\overline{p}) \neq \overline{e}_2(\overline{p})$ and $\overline{e}_1$ and $\overline{e}_2$ are not both valid for $\overline{p}$. $\square$

Every convex region can be represented in two different ways using half spaces (H-representation) or vertices of the convex region (V-representation) and these two representations are convertible in a very efficient way [21]. In our algorithm, we use both representations, the V-representation for finding the vertices, and the H-representation for calculating the splitting plane. As before, we assume that parameter ranges are bounded. Algorithm *Divide&Conquer*, given below, receives G and a convex region CR as input. Initially, when applying the algorithm to the entire parameter space, CR is a d-dimensional box obtained by the ranges of the parameters. In Line 4 and 5, all the cycle mean values of all the vertices of CR are checked for the validity of the mcme obtained for a random point $\overline{rp}$ in the interior of CR. $Th(\overline{v}_i)$ is the throughput of G for point $\overline{v}_i$, obtained using the non-parametric state-space method of Chapter 4. In case the mcme of $\overline{rp}$ is not valid for a vertex $\overline{v}_i$, then the splitting plane obtained from mcmes of $\overline{v}_i$ and $\overline{rp}$ splits CR (illustrated in Figure 5.4 for the example with $\overline{v}_3$ in the role of $\overline{v}_i$) by adding half-spaces characterized by vectors $\overline{e}_{rp} - \overline{e}_{v_i}$ and $\overline{e}_{v_i} - \overline{e}_{rp}$ to CR, respectively. Then the algorithm is invoked for both subregions in Lines 9 and 10. Procedure ranCornerExpr receives a vertex $\overline{v}_i$, expression $\overline{e}_{rp}$, and $\overline{rp}$. It produces the mcme valid in $\overline{v}_i$ which will be different from $\overline{e}_{rp}$. Note that inside this procedure, instead of using $\overline{v}_i$ itself which is typically on the border between throughput regions, for the reason explained, a randomly selected point in its neighborhood on the line through $\overline{v}_i$ and $\overline{rp}$ is used instead. In fact, in our algorithm this point is chosen as the first point on this line with different associated mcme from that of $\overline{rp}$ by moving in a zeno-fashion toward $\overline{v}_i$. Of course, this is our choice in this algorithm and any other arbitrary point in the region with different mcme is suitable for constructing the splitting plane as well.

The algorithm is guaranteed to terminate, because there is a finite number of mcmes and hence a finite number of borders between regions exist that can be used for splitting. The correctness of the algorithm is proven in the following theorem.

Theorem 5.2. *Given a parametric SDFG G and a parameter space CR, algorithm Divide&Conquer finds all the mcmes of parameter space CR and it terminates.*

Proof. We prove the correctness of this function by induction on the number of throughput regions of CR. The base case for the induction is when all the corner points of CR have the same mcmes as a randomly generated point $\overline{e}_{rp}$ in the interior of CR. Then, due to the convexity of the throughput region, proven by Proposition 5.2 we know that only one mcme is valid for CR, i.e., CR is part of only one throughput region. In this case, Algorithm *Divide&Conquer* calculates this mcme and terminates immediately. This check is done in the for-loop of Line 4. In case that at least one of the corner points has a different mcme, then the splitting plane made by the mcme of $\overline{e}_{rp}$ and the mcme of another point in the interior of the mcme splits CR. This splitting plane due to Proposition 5.3 cuts CR into two smaller regions, CR_1 and CR_2, in Lines 7 and 8. Both CR_1 and CR_2 must have at least one throughput region less than CR, because we know that the splitting plane used for cutting CR into CR_1 and CR_2, is constructed by removing an mcme from both CR_1 and CR_2. This implies that the used mcme from CR_1 cannot be dominating in CR_2 and vice versa. In other words, CR_1 lacks the throughput region related to the mcme of CR_1 which is used for the splitting plane. Similar reasoning is valid for CR_2 as well. By induction, we know that the mcmes of both regions CR_1 and CR_2 are found in the recursive calls. The final set of mcmes is the union of the expressions found in each of the recursive calls.

We know that the HSDFG corresponding to G has only a finite number of simple cycles and that each mcme relates to a cycle in this HSDFG. Therefore, there are only a finite number of different mcmes. Therefore, the algorithm terminates. □

```
Algorithm Divide&Conquer($G$, $CR$)
Input: A strongly connected parametric SDFG $G$
Input: A convex region $CR$
Output: DCMS of $G$
1.  Let $\overline{rp}$ be a random point in $CR$;
2.  $\overline{e}_{rp} \leftarrow \mathrm{findMCME}(G, \overline{rp})$;
3.  insert($DCMS, \overline{e}_{rp}$);
4.  for all vertices $\overline{v}_i \in CR$
5.          if $(\overline{e}_{rp}(\overline{v}_i) \neq 1/Th(v_i))$
6.             then $\overline{e}_{v_i} = \mathrm{ranCornerExpr}(\overline{v}_i, \overline{e}_{rp}, \overline{rp})$;
7.                  $CR_1 \leftarrow CR \cup \{\overline{e}_{rp} - \overline{e}_{v_i}\}$;
```

```
            $CR_2 \leftarrow CR \cup \{\overline{e}_{v_i} - \overline{e}_{rp}\}$;
            Divide&Conquer($G$, $CR_1$);
            Divide&Conquer($G$, $CR_2$);
```

It remains to explain how an mcme for a given point in the parameter space can be found (findMCME in the algorithm above). As already mentioned, an adapted version of the state-space method of the previous section is used for finding the mcme of the graph for a concrete point $\overline{p}$. In fact, this method keeps two multisets for remaining execution times. One contains the parametric expressions obtained from the parametric execution times, the other contains the concrete values of the remaining execution times which are in fact the evaluation of the expressions of the first multiset for the concrete point $\overline{p}$. The modified algorithm only explores the part of the state space corresponding to concrete point $\overline{p}$. The multiset with the concrete remaining execution times is used for detecting the recurrent state, omitting the check on depth of the recursion. The multiset of expressions is used for calculating the mcme of the SDFG at point $\overline{p}$. This makes the method equivalent to the concrete state-space throughput analysis method of Chapter 4, and a recurrent state is guaranteed to be found (by Proposition 4.1).

5.6 Experimental Results

We implemented our methods in the SDF3 tool [65]. We have evaluated the execution times of our algorithms using SDFG models of seven real applications. We used the same benchmark as in Chapter 4, consisting of an H.263 decoder, an MP3 decoder, a modem, a satellite receiver, a sample-rate converter, an H.263 encoder and an MP3 playback application. In *Divide&Conquer*, the CDDLib library [21] is used for all polyhedra operations. In *coverStateSpace*, all operations related to linear inequality systems have been done using LPSolve [5]. All experiments were performed on a P4 PC running at 3.4 Ghz.

In each graph, to each actor with varying execution times a parameter has been assigned. Actors with constant execution times received fixed execution times. In cases where more than one copy of an actor exists in the SDFG, the same parameter was dedicated to all copies. Two experiments with the same parameter set and different ranges for the parameters have been carried out. We used two different ranges for parameters with the same lower-bounds and upper-bounds as large as 110% and 150% of the lower-bounds. These ranges were chosen in line with the worst-case estimates of the execution times of the benchmarks, if any were given.

One can interpret *coverStateSpace* and *Divide&Conquer* as one algorithm where the depth input parameter D of *coverStateSpace* can be used to determine the amount of parametric state-space exploration vs. the amount of calls to the concrete state-space method made by *Divide&Conquer*. To test whether true parametric state-space exploration is beneficial, we execute *coverStateSpace*

Table 5.1: Experimental results

	#pa	#act	rep	110%			150%		
				st[s]	dc[s]	#e	st[s]	dc[s]	#e
H.263 decoder	4	4	1190	0.854	0.590	1	0.862	0.589	1
H.263 encoder	5	5	201	0.119	0.212	1	0.241	0.211	1
modem	7	16	48	51	0.568	1	168	0.570	1
MP3 decoder	8	14	14	0.196	0.889	1	0.253	0.896	1
MP3 playback	1	4	10601	8.643	1.268	1	17	8.177	2
sample-rate conv.	4	6	612	102	1.040	2	266	1.246	2
satellite rec.	9	23	4515	-	480	3	-	450	3

without checking the recursion depth, observing that in line with our conjecture in Section 5.4.3 the algorithm terminates in all cases. We furthermore execute *Divide&Conquer* on the entire parameter space without parametric state space exploration. We did not entirely test the *HSDFG Method* of Section 5.3 because it is too slow for practical use. The reason is that the *HSDFG Method* works on the HSDFGs, and even though we have implemented the fastest cycle enumeration algorithm [35], the algorithm takes generally too long. It only worked for the MP3 decoder for which it only took few a milliseconds to compute the DCMS.

The results of the experiments are shown in Table 5.1 in two different columns. For each experiment, for each graph, the time for both the state-space method (st) and divide-and-conquer (dc) in seconds, as well as the number of expressions in the DCMS (#e) are shown. In all cases, only very few dominating mcmes (up to 3) have been found, which is a good indication for the simplicity of the resulting throughput expression.

The number of parametric execution times (#pa), the number of actors (#act) and the sum of their repetition vector entries (rep, which is also the number of actors in the equivalent HSDFG) of each graph is shown. Since the number of cycles in the equivalent HSDFG directly corresponds to the number of different cycle mean expressions, the sum of repetition vector entries is an important indication for the expected run-time, besides the actor and parameter counts.

The two methods compared in Table 5.1 are fast in most cases with only the satellite receiver taking substantially more than a few seconds. The divide-and-conquer method is fast in all the cases. It is also less sensitive to the ranges of parameters than the state-space method. However, its execution time does scale up exponentially with an increasing number of parameters. On the other hand, typically, in practical applications only a few parameters are needed since the number of actors with varying execution times is limited and the variations can be captured by the same underlying parameters.

The state-space method works very fast for applications like the H.263 decoder, the H.263 encoder, the MP3 playback and the MP3 decoder, which have a few actors with large execution times. In a few cases, it is faster than divide and conquer. On the other hand, it performs poorly on graphs whose actors have approximately equal execution times. For example, for the satellite receiver, the algorithm took more than a few hours.

Summarizing, the results show that a design-time parametric throughput analysis is feasible, with a purely divide-and-conquer approach over the entire parameter space performing best. Considering the results in, for example, the context of a run-time resource- or quality management application as proposed e.g. in [54], it is clear that the processing time and memory usage of a throughput calculation for concrete values of the execution time parameters, consisting of an evaluation of the maximum value of the obtained dominating mcmes, are negligible compared to the processing time and memory usage of the typical streaming application. They are in general also small compared to the processing time and memory usage of a traditional throughput calculation, which is typically too expensive to perform at run-time.

5.7 Parametric Throughput Analysis of Arbitrary SDFGs

So far in this chapter, all the throughput calculation techniques explained above work on consistent and strongly connected parametric SDFGs. In this section, we extend the throughput calculation techniques to arbitrary consistent parametric SDFGs. Definition 4.2, in Section 4.5, defines the throughput of an arbitrary SDFG as the normalized throughput of its slowest actor. Proposition 4.3 states that the normalized throughput of all actors in a consistent strongly connected SDFG are equal. Furthermore, the normalized throughput of an actor in a consistent and strongly connected parametric SDFG for every point in the parameter space can be obtained by the value of an mcme in the graph's DCMS which has the maximum value in that point. So, the throughput of a parametric SDFG can be found as follows.

First, we find the DCMS of every maximal strongly connected component of the graph separately. By unifying these sets, we obtain a set which gives the inverse throughput of the arbitrary graph for every point in the parameter space. However, during the unification, some of the mcmes may get dominated by others. The DMCS of an arbitrary parametric SDFG can be obtained by removing the redundant mcmes of this set using the techniques explained in Section 5.3.

5.8 Related Work

The only directly related work we are aware of is [52] which is only applicable on the weighted directed graphs explained in the previous chapter. This work has been proposed in Max-Plus algebra notations and it studies the variations of mcmes if all the actor execution times change with the same value, which means it only allows one parameter in actor execution times.

The main idea of having throughput regions in the *Divide&Conquer* algorithm has been inspired by the regions from timed automata [2] in which, similar to throughput regions, each region shares the same property.

5.9 Summary

We have extended throughput analysis of SDFGs to parametric SDFGs so that actors can have parameters as their execution times. The throughput of such graphs is a function of the parameters. Evaluating these functions is in general faster than concrete throughput analysis methods. We adapted existing methods for computing throughput to parametric SDFGs and proposed a new, faster, algorithm.

Chapter 6

Latency

6.1 Overview

The main goal of using synchronous data flow graphs is to provide predictable performance. Throughput as one of the most prominent performance metrics is discussed in the previous chapters. Other performance indicators are storage requirements and latency. Buffer minimization for SDFGs has also been studied [24, 64, 70], but latency has until recently only been studied for HSDFGs [39, 44, 60]. Latency is important in interactive applications such as video conferencing, telephony and games, where latency beyond a certain bound becomes annoying to the users. It is in principle possible to compute latency metrics for an SDFG via a conversion to a homogeneous SDFG. However, as mentioned before, this conversion might lead to an exponential increase in the number of nodes in the graph, which makes it prohibitively expensive in some cases.

In this chapter, we present a technique to compute the minimal achievable latency between the executions of any two actors in an SDFG. We also present an execution scheme that defines a class of static order schedules, i.e., the order of actor firings is determined a priori, which gives a minimal latency. Since this scheme may negatively affect throughput, we also propose a heuristic to minimize latency under a throughput constraint. We prove that simultaneously optimal throughput and latency is not achievable in all cases. We evaluate our schemes in a single-processor context and in a multi-processor context with sufficiently many resources to maximally exploit parallelism, for various buffering schemes. In the multi-processor context, we compare our execution schemes with self-timed execution. In many cases, substantial gains in latency are possible. It further turns out that for all real models and for most synthetic cases minimal latency and maximal throughput can be achieved simultaneously. For example, we calculated the latency and the throughput of the self-timed execution of an MP3 decoder modeled as an SDFG, taken from [61] and already used in Chapters 4 and 5, assuming the availability of enough resources for achieving maximum parallelism. Our calculation shows that the latency between actors requantization and Subband inversion for self-timed execution is as large as 43ms. We can achieve the same throughput of a frame per 10ms while reducing the latency to 27ms just by changing the order of actor firings.

Section 6.2 defines a notion of latency for SDFGs, generalizing a definition of latency for homogeneous SDFGs [60]. Section 6.3 introduces an execution scheme that minimizes latency, while Section 6.4 presents latency-optimal static order scheduling policies for single-processor systems and for a multi-processor context with sufficient resources to exploit maximal parallelism. Section 6.5 gives our technique for minimizing latency under a throughput constraint. It also disproves the existence of a general scheduling scheme for achieving simultaneous optimal throughput and latency in all cases by providing a counter example. In Section 6.6, we experimentally evaluate our techniques. Section 6.7 discusses related work and finally Section 6.8 summarizes. This chapter is based on publication [30].

6.2 Latency

This section formally defines a notion of latency for timed SDFGs. Generally speaking, latency is the time delay between the moment that a stimulus occurs and the moment that its effect begins or ends. In timed SDFGs, stimuli are actor firings and their effects are the consumptions of produced tokens by some other actors. In the remainder of this chapter we limit ourselves to consistent, live and strongly connected SDFGs. The latency concepts developed in the chapter can be extended to non-live and non-strongly connected graphs, but the definitions and reasoning becomes much more tedious. Furthermore, as argued before, non-live SDFGs are of little interest in the multimedia domain, and many SDFGs in that domain are strongly connected either inherently or due to buffering constraints (See [40, 64]). To define latency, first, we need to define the following.

Definition 6.1. (Corresponding Firing) *Let $a_1, a_2, \ldots, a_k \in A$ be actors of a timed SDFG (A, C, E) on a directed path $a_1, a_2, \ldots, a_k$ connecting a_1 to a_k. We say that the j_1-th firing of a_1 corresponds to the j_k-th firing of a_k iff there exist $j_i \in \mathbb{N}$ such that j_2 is the first firing of a_2 which consumes at least one token produced by the j_1-th firing of a_1, j_3 is the first firing of a_3 which uses at least one token produced by the j_2-th firing of a_2, and so on. We denote the firing of a_k corresponding to the j_1-th firing of a_1 by $cf(a_1, j_1, a_k)$.*

Note that in general the time that tokens need to travel from some source actor to some destination actor may differ in different firings of the source actor. In an HSDFG, where all production and consumption rates are one, there is a one-to-one correspondence between actor firings of some source and some destination. Because of differing firing rates, this correspondence does not exist, in general, between actors in an SDFG. In order to arrive at a proper definition of

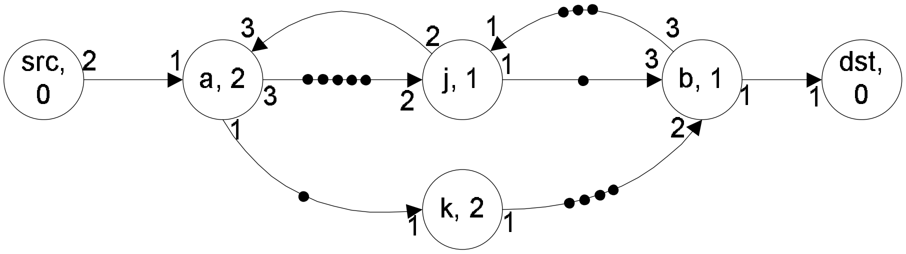


Figure 6.1: An example latency SDFG.

latency for SDFGs, we add an explicit source actor to the source of our latency measurement and a destination actor to the intended destination, each of which fires by construction exactly once in every iteration of the graph. If an SDFG already has meaningful input and output actors with repetition vector entries of one, these actors can function as source and destination and no actors need to be added.

Definition 6.2. (Latency Graph) *Let $a, b \in A$ be two actors of a timed SDFG (A, C, E) with repetition vector q, and let $src, dst \notin A$ be two new actors. We define the latency graph for actors a and b as $G_{L(a,b)} = (A_L, C_L, E_L)$, where $A_L = A \cup \{src, dst\}$, $C_L = C \cup \{c_{src}, c_{dst}\}$ where c_{src} is a channel from src to a with production and consumption rates of $q(a)$ and 1, and c_{dst} is a channel from b to dst with production and consumption rates 1 and $q(b)$ and $E_L = E \cup \{(src, 0), (dst, 0)\}$.*

The latency between two actors is defined through the latency of different firings of actors *src* and *dst* in their latency graph. Note that *src* and *dst* have execution time 0 and unlimited auto-concurrency, so that their addition does not influence the timing behavior of the graph, as further clarified below.

In this chapter we use the SDFG example depicted in Figure 6.1 as our running example. It has six actors src, dst, a, b, j, k and its repetition vector q is $\{(src, 1), (a, 2), (j, 3), (k, 2), (b, 1), (dst, 1)\}$. Observe that the example of Figure 6.1 shows in fact the latency graph for actors a and b of the SDFG obtained when omitting the *src* and *dst* actors. As *src* does not have any input channel it can fire as often as needed; therefore it puts no restriction on the firings of a. Also, as channels are unbounded, the firing of actor b is not restricted by actor dst. Furthermore, because both actors *src* and *dst* have execution times zero, and do not impose any restrictions on the firings of the other actors, any execution of the latency graph is an execution of the original graph too when *src* and *dst* are omitted from that execution.

The following proposition shows that there is a one-to-one correspondence between *src* and *dst* firings (where *dst* may have some initial firings without corresponding *src* firing).

Proposition 6.1. *Let* $G_{L(a,b)}$ *be some latency graph. There is some* $\delta \in I\!N_0$ *such that the* k*-th firing of source actor src for arbitrary* $k \in I\!N$*, corresponds to the* $(k+\delta)$*-th firing of dst, i.e.,* $cf(src, k, dst) = k + \delta$ *for all* $k \in I\!N$*.*

Proof. According to the definition of the repetition vector and of an iteration, we know that if a graph executes a complete iteration i.e., each actor a fires as many times as $q(a)$, then the channel state does not change. Therefore, firing a complete iteration does not change the relation of corresponding firings of any two actors in the graph. Because *src* and *dst* each have entry 1 in the repetition vector, the corresponding firing of the k-th firing of *src* is always firing $k+\delta$ for some fixed δ, if δ is the *dst* firing corresponding to the first firing of actor *src*. □

In practice, we are mostly interested in the latency of actors which are considered the input and the output of the system, and these actors often have a repetition vector entry of one already. Furthermore, usually, only executions of complete iterations of graphs are meaningful. Therefore, in case actors have repetition vector entries different from one, we do not look at all firings of those actors. Instead, via the addition of the *src* and *dst* actors, the latency is defined (below) on the groups of firings of each actor that contain as many firings of the actors as their repetition vector entries.

Executions of SDFGs and consequently their latencies directly depend on the platform they are mapped onto. In other words, due to resource constraints, such as for example a limited number of processing units, some executions might not be feasible. The set of feasible executions is denoted FE. Later in this chapter, feasible executions are defined assuming two types of platforms. The first platform is assuming a single processing element in which only one actor can fire at a time and excludes any concurrent firings of actors. The second platform is assumed to have sufficiently many processing elements in order to exploit all potential parallelism of the applications, i.e., each actor can essentially start its firing as soon as all of its input data is available.

Definition 6.3. (LATENCY) *Let* $a, b \in A$ *be two actors of a timed SDFG* (A, C, E) *with latency graph* $G_{L(a,b)}$*. The* k*-th latency of* a *and* b *for an execution* σ *is defined as the time delay between the* k*-th firing of src and its corresponding firing of dst in* σ*, and it is denoted by* $L_k^\sigma(a,b)$*:*

$$L_k^\sigma(a,b) = F^\sigma_{dst, cf(src,k,dst)} - F^\sigma_{src,k}.$$

The latency *of actors* a *and* b *in execution* σ*,* $L^\sigma(a,b)$*, is defined as the maximum* k*-th latency of* a *and* b *for all firings of* a*:*

$$L^\sigma(a,b) = \max_{k \in I\!N} L_k^\sigma(a,b).$$

The minimal latency *of actors* a *and* b*,* $L^{min}(a,b)$*, is defined as the minimum over all* feasible *executions in FE:*

$$L^{min}(a,b) = \min_{\sigma \in FE} L^\sigma(a,b).$$

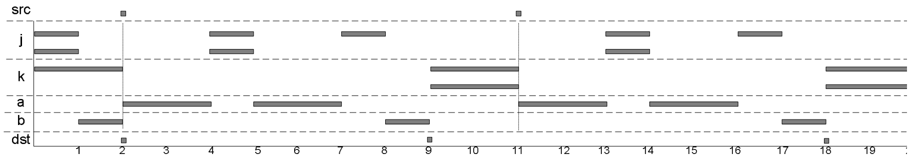


Figure 6.2: A timed execution of the example SDFG.

When assuming that *src* fires immediately before an a firing and *dst* as soon as it gets enabled, this definition implies that latency is measured from the start time of a firing of actor a (or group of firings), intuitively corresponding to the consumption of some input, to the finishing time of the corresponding (group of) firing(s) of actor b, intuitively corresponding to the production of output directly related to the consumed input. The current definition is consistent with and generalizes the definition of latency given for HSDFGs in [60].

Figure 6.2 shows a scheduling trace of the running example. The horizontal axis represents the progress of time. Each row is dedicated to the firing sequences of an actor. To actors with simultaneous firings more rows are dedicated. For example, actors j and k have two rows each. Each actor firing is represented by a box which starts at the time where the firing starts and lasts as long as the execution time of that actor. As the execution times of actors *src* and *dst* are zeros, they are shown by very small boxes. All the executions shown in this chapter are periodic, and the periodic part which repeats indefinitely is specified between two vertical lines. The latency between actors a and b for the execution of Figure 6.2 of the running example equals 7, being the total delay between the firings of actors *src* and *dst* in any period. It will turn out below that the execution shown minimizes the latency between firings of actors *src* and *dst* (and hence between firings of a and b).

6.3 Minimum Latency Executions

In this section, we introduce an execution scheme to determine the minimal possible latency.

The idea of having an execution in which the latency between actors *src* and *dst* is minimal, is by minimizing the number of actor firings between each firing of *src* and its corresponding firing of *dst*. In other words, by allowing only the necessary set of actor firings between each firing of *src* and corresponding *dst* firing we get a class of executions with minimum latency between *src* and *dst*. The following definition characterizes this class of executions in more detail.

Definition 6.4. (Minimum Latency Execution) *Let $G_{L(a,b)}$ be the latency graph of a strongly connected timed SDFG $G = (A, C, E)$ with actors a and b. A feasible execution consisting of the repetition of the following four phases is called* *a minimum latency execution.*

Phase 1 *Execute actors except src until src is the only enabled actor and no other actor firings are ongoing. (Note that src is always enabled because it does not have any inputs.)*

Phase 2 *Fire src once.*

Phase 3 *Execute, without any unnecessary delays, the minimum set of required actor firings for enabling dst for one firing.*

Phase 4 *Fire dst once.*

Let P_n *with* $n \in \mathbb{N}$ *the part of the execution trace which represents the n-th execution of the four phases.*

Figure 6.2 shows a minimum latency execution of the running example. In Phases 1 and 3, execution is self-timed. Phase 1 finishes after a finite number of actor firings because we assumed that the original SDFG is strongly connected, i.e., all the actors in the graph are causally dependent on each other. Therefore, if actor a stops firing, only a finite number of firings of other actors can occur. Note that an SDFG may exhibit more executions that realize minimum latency than those defined in Definition 6.4. However, the defined executions are guaranteed to have minimum latency (proved below).

Proposition 6.2. *Let* $G = (A, C, E)$ *be a live timed SDFG with* $G_{L(a,b)}$ *the latency graph for actors* a *and* b *in* A*. Any minimum latency execution of* $G_{L(a,b)}$ *has the following properties.*

1. P_n *equals one iteration for all* $n > 1$ *and the state reached after Phase 1 is the same for all* $n \geq 1$.
2. *The n-th firing of src and its corresponding firing* $cf(src, n, dst)$ *of dst occur in the same* P_n.
3. *The set of actor firings between any firing of src and its corresponding firing of dst is the smallest possible set among all executions.*

Proof. Part 1: After Phase 1, no actor is enabled, except *src*. Strong connectedness of G implies that Phase 1 terminates. The repetition vector entry is one for *src*, and by construction a firing of actor *src* enables actor a for $q(a)$ firings, when q is the repetition vector. This number of firings of a enables all successors of a for as many firings as their repetition vector entries, and so on. In fact, there is such an execution trace for all live SDFGs [41]. Since there is only one firing of *src*, no actor can also be fired more often than its repetition vector entry. Some part of these firings happen in Phase 1 and the rest happen in Phases 3 and 4. While going from the end of Phase 1, through Phases 2, 3, 4 and 1 again, the

numbers of firings correspond exactly to the repetition vector and, hence, by the balance equations (Definition 2.8), the state reached must be the same.
Part 2: During the execution of Phase 1 in P_1, *dst* fires as often as it can without using any tokens from the first firing of *src*. Hence, the first firing of *dst* after this phase depends on a token from the first firing of *src* after this phase, which is the first firing of *src* ever. *src* fires in Phase 2 of P_1 and *dst* in Phase 4 of P_1. In each of the following P_n, based on Part 1 of the proof and the fact that both *src* and *dst* have repetition vector entry one, both *src* and *dst* fire once in Phases 2 and 4 respectively and because of Proposition 6.1, these firings are each pairs of corresponding firings.
Part 3: By definition of Phase 1, we know that none of the firings of Phase 3 can fire in Phase 1, since all of them depend on the firing of *src*. Besides, Phase 3 only fires the minimum set of actor firings needed for enabling *dst* after the firing of *src*. Therefore, the set of actor firings in Phase 3 is the minimum set of firings possible between the corresponding *src* and *dst*. Hence, for any other execution of the SDFG, the firings between the corresponding *src* and *dst* firings contain at least those firings of Phase 3. □

Proposition 6.2 shows that the set of firings in between the designated *src* and *dst* actors is minimal in any minimum latency execution (part 3), that a minimum latency execution is periodic, in a sense that the states reach after the execution of phase 1 follow a periodic behavior (part 1), and that the pairs of corresponding *src* and *dst* firings that determine the latency always occur in one period (part 2). The precise duration of the firings between *src* and *dst* firings depends on the particular execution. The set of allowed executions may be constrained by the available platform; a single-processor platform, for example, does not allow concurrent execution. If Phase 3 firings are executed within platform constraints without unnecessary delays, the following result follows in a straightforward way from Proposition 6.2.

Theorem 6.1. *Let σ be any minimum latency execution of a latency graph $G_{L(a,b)}$ taken from the set of feasible executions FE. Then, we have*

$$L^{\sigma}(a,b) = L^{min}(a,b).$$

Proof. Part 3 of Proposition 6.2 states that the set of firings in between a *src* firing and its corresponding *dst* firing is the smallest possible. Thus, execution of this necessary set, without any unnecessary delay, leads to an execution with a minimum possible latency. Note that executing a set of actors without any unnecessary delay means that each actor in the set starts its firing as soon as it is enabled and there is a free processor available in the platform on which the graph is executed. In the other words, they are executions in which all actors fire as soon as they have their required data and resources available. □

Observe that Proposition 6.2 proves that a minimum latency execution has a periodic phase consisting of one iteration of the SDFG. In general, executions,

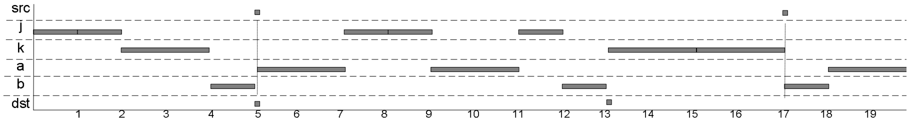


Figure 6.3: A single-processor minimum latency static order schedule of the example SDFG.

such as for example self-timed execution, that optimize throughput, might have a periodic phase consisting of multiple iterations [29]. In the example execution of Figure 6.2, the throughput of (output) actor b is 1/9. Below, it becomes clear that this is not optimal.

6.4 Static Order Scheduling Policies

In general, scheduling an application involves assigning actors to processing elements, ordering the execution of each actor on each processing element and finally determining the firing times for each actor such that data and control dependencies are met. In this chapter, we only look into two types of platform. In both of our platforms, the single-processor and sufficiently-many-processor actor assignments are trivial. Also the last two cases of scheduling in both of these steps are combined by specifying the order of actor firings.

A well-known schedule type in which the firing orders of all actors are determined at the compile time is called the *static order schedule*. The minimum latency execution given in Definition 6.4 results in a static order schedule as it determines the order of firings of all actors in the execution. In the following, we explain the minimum latency execution for each of the two discussed platforms.

6.4.1 Single-Processor Scheduling

In the previous section, we have seen that any minimum latency execution leads to a minimum latency between the designated pair of actors. To create a static order schedule for a single processor, it only remains to order the various executions in Phases 1 and 3 of the scheme. If only considering latency, this order can be arbitrary, as long as it satisfies the data and control dependencies specified by the channels in the SDFG. One could decide to try to optimize other constraints such as code size, using for example single appearance scheduling techniques [8, 67]. With respect to throughput, it can be observed that the order in which the individual actors are scheduled in any feasible schedule of a consistent SDFG on a single processor does not impact the average throughput of the application as long as there are no idle periods. Therefore, any minimum latency execution combines minimum latency with the maximal throughput that can be obtained on a single processor.

Figure 6.3 shows a single-processor static order schedule for our running example that adheres to the minimum latency execution scheme. The latency between actors a and b is 8 and the throughput of b is 1/12. Both latency and throughput are optimal for a single processor.

6.4.2 Scheduling with Maximal Parallelism

An interesting case in a multi-processor context, is the case that sufficiently many resources are available to maximally exploit parallelism, or in other words, a case with unlimited processing resources so that any enabled actor can always make progress and feasibility of executions is only determined by the data dependencies specified by the graph structure. As mentioned, this allows to determine the minimum achievable latency constrained only by the dependencies in the SDFG. The result can be used as a feasibility check for the application latency in a (multi-processor) design trajectory.

Observe that the crucial point in the 4-phase minimum latency execution scheme is that the actor firings of Phase 1 cannot interfere with the firings in Phase 3. In a single-processor context, this simply means that these two phases have to be executed completely separately. However, in a context with sufficient resources, the two phases can be allowed to execute concurrently, in a self-timed manner, because firings of Phase 1 that are executed concurrently with firings of Phase 3 do not interfere with those Phase 3 firings. Furthermore, self-timed execution minimizes the execution time of the critical path of the actor firings in Phase 3. Since also the firing of *dst* (Phase 4) can be integrated into this self-timed execution scheme, these observations lead to the following execution scheme.

Definition 6.5. (Minimum Latency Execution Scheme with Unlimited Resources)

Phase 1 *Execute actors of the latency graph except src in a self-timed manner until src is the only enabled actor and there are no ongoing firings.*

Phase 2 *Fire src once, and repeat.*

This scheme suggests a concrete multi-processor static order schedule that simply schedules the actor firings in the two phases of this minimum latency execution scheme iteratively in a self-timed manner. Note that the first execution of Phase 1 might be different from the other executions of Phase 1, so that the resulting static order schedule still has a transient part and a periodically repeated part. Figure 6.4 shows a latency-optimal static order schedule adhering to this scheme. The latency between actors a and b is 7, which is of course the same latency as in the execution of Figure 6.2 which was already optimal given the dependencies inherent in the SDFG. The advantage of the new execution scheme shows in the improved throughput. The throughput of actor b in the execution of Figure 6.4 is 1/7, whereas it is 1/9 in the execution of Figure 6.2.

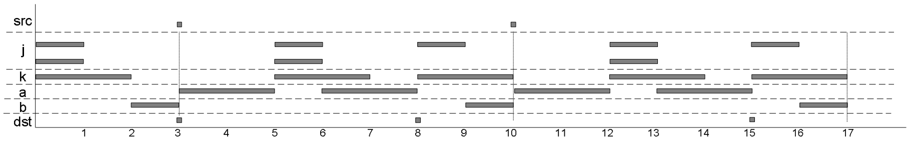


Figure 6.4: A minimum latency static order schedule using the optimized execution scheme for unlimited resources.

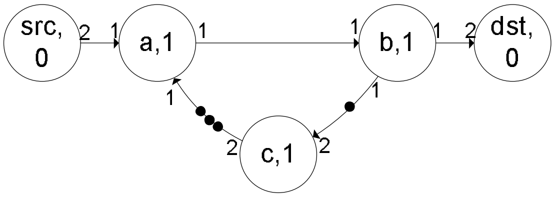


Figure 6.5: A counter example G for simultaneously optimizing throughput and latency.

It is interesting to observe that the minimal achievable latency given some unspecified amount of processing resources is always between the minimal latency with maximal parallelism and the minimum latency for a single processor.

6.5 Throughput Constraints

A multimedia application is often subject to multiple performance constraints such as latency, throughput and memory usage. So far, we have seen several scheduling policies for obtaining minimum latency. The single-processor policy achieves also the maximum throughput since it fully utilizes the only processing unit.

The maximum parallelism policy, as the other policies, disallows overlap between multiple iterations of the SDFG. This potentially influences throughput negatively. By allowing simultaneous firings of the source actor in Phase 1 of Definition 6.5, multiple iterations of the SDFG execution can be scheduled in parallel, which may lead to a higher throughput [41, Sec. 3.4]. However, this might have a negative effect on latency. In fact, we have the following proposition.

Proposition 6.3. *Given an arbitrary SDFG G; assume sufficiently many resources are available, i.e., the feasibility of executions is only determined by the data dependencies between actors. G does not necessarily have an execution that simultaneously minimizes latency and maximizes throughput.*

Figure 6.5 shows an example SDFG G for which it is not possible to simultaneously optimize latency and throughput. The minimal latency that can be

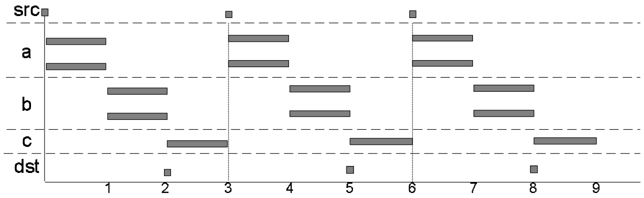


Figure 6.6: Minimal latency execution of G.

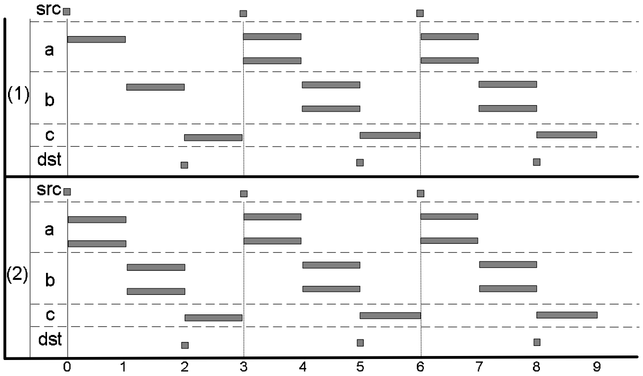


Figure 6.7: Maximal throughput execution of G.

obtained for G is 2. The minimal latency execution obtained via Definition 6.5 is shown in Figure 6.6.

Figure 6.7 shows the self-timed execution of this example (split into two parts, as explained later), with the exception that actor *src* fires only when actor a needs tokens for firing. The firings of *src* in Figure 6.7 are scheduled in such a way that they do not constrain throughput, so the execution in Figure 6.7 achieves maximal throughput. For example, the throughput of actor b is 4/3 firings per time unit. We see that the latency of the execution is 5 (due to the *src* and *dst* firings in part (1) of the execution).

The self-timed execution of G can be divided into two parts. Suppose we color the first token on channel c-a and the token on b-c blue and the second and third token on channel c-a red. This coloring implies that firings of actor c always consume and produce tokens of one color. In Figure 6.7, (1) and (2) correspond to actor firings involving blue and red tokens on channels a-b, b-c, and c-a respectively.

The minimal latency execution of Figure 6.6 follows the schedule of part (2) in Figure 6.7, i.e., all tokens are processed according to the red scheme. Throughput of b in the execution of Figure 6.6 is 2/3 firings per time unit. This is the maximum that can be achieved without executing multiple iterations of G concurrently.

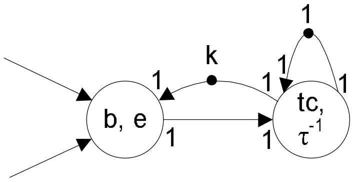


Figure 6.8: Modeling a throughput constraint.

(Note that an iteration of G consists of one firing of *src*, *dst*, and *c*, and two firings of *a* and *b*.) However, executing multiple iterations concurrently implies that tokens are necessarily processed according to the blue scheme, part (1), of Figure 6.7 (or an even slower scheme). This implies that increasing throughput necessarily leads to a higher latency, proving Proposition 6.3.

A consequence of Proposition 6.3 is that it is interesting to explore the throughput and latency trade-off under the maximal parallelism assumption. In the remainder of this section, we propose a heuristic execution scheme that attempts to minimize latency under a given throughput constraint. That is, the algorithm tries to schedule the SDFG in such a way that the throughput constraint is met, while latency is minimized. If the SDFG is inherently too slow to meet the throughput constraint, the algorithm returns a schedule with maximum throughput and a minimized latency.

An important observation is that a throughput constraint can be modeled in an SDFG (See Figure 6.8). Assume we want to impose a throughput constraint of τ firings per time unit on a designated actor b. This can be achieved by adding a fresh actor *tc* to the SDFG with a self-loop containing one token to avoid simultaneous firings of *tc* and with an execution time of τ^{-1}. By adding two channels between b and *tc* as shown in the figure, *tc on the long run* prohibits b to fire more often than τ times per time unit. We can calculate the number of initial tokens on the channel from *tc* to b, denoted by k in the figure, such that the graph achieves the maximum throughput that can be obtained by the graph without k-b channel, i.e., such that the cycle through b and *tc* does not restrict the throughput. In fact, k is the buffer size for the channel connecting b to *tc*, and the minimum k can be calculated by the method proposed in [64, 66]. Even by choosing the right k the rest of the SDFG might slow down b more than *tc*, so the realized throughput for b could be lower than τ. The number of tokens in the *tc*-b channel determines how much short-term deviation in b-s throughput is allowed. This jitter may influence the minimal achievable latency from any source actor to b.

A throughput constraint can be added to the sink actor of a pair of actors for which latency needs to be minimized. This results in the throughput-constrained latency graph.

Definition 6.6. (THROUGHPUT-CONSTRAINED LATENCY GRAPH) *Let* $G_{L(a,b)} =$

(A_L, C_L, E_L) be the latency graph of some SDFG $G = (A, C, E)$ with actors $a, b \in A$ and with repetition vector q. Let τ be a throughput constraint on actor b, and let $tc \notin A$ be a new actor. We define the τ-constrained latency graph for actors a and b as $G_{L(a,b,\tau)} = (A_\tau, C_\tau, E_\tau)$, where $A_\tau = A_L \cup \{tc\}$, $C_\tau = C_L \cup \{c_0, c_1, c_2\}$. c_0 connects b to tc, c_1 connects tc to b and c_2 is a self-loop of tc. All production and consumption rates of these channels are also equal to one. $E_\tau = E_L \cup \{(tc, \tau^{-1})\}$. The initial channel state γ_0 for the new channels c_0, c_1, c_2 is defined as follows: $\gamma_0(c_0) = 0$, $\gamma_0(c_2) = 1$, and $\gamma_0(c_2) = k$ with k the minimal buffer size of channel c_0 needed to achieve maximal throughput of the graph without channel c_1, as computed by the technique of [64, 66].

Given a throughput-constrained latency graph, the goal of minimizing latency under the throughput constraint reduces to minimizing latency while maintaining maximal throughput of the throughput-constrained SDFG. As mentioned, maximal throughput can be achieved via self-timed execution. The algorithm presented below essentially performs a self-timed execution, except that the firings of the designated actor *src* are delayed. The idea is that latency is minimized by scheduling the firing of *src* precisely the minimum achievable latency number of time units before the *dst* firing times in self-timed execution. The algorithm does not change the average number of firings over time of any actor in the graph, although it may delay some firings over time. In other words, the maximal throughput of entirely self-timed execution is maintained, but dependencies in the graph may cause the *dst* actor to fire at a different moment in time in the schedule produced by the algorithm when compared to the self-timed execution. Consequently, the latency need not be equal to the minimal achievable latency.

```
Algorithm optimizeThroughputLatency ($G_{L(a,b,\tau)}$)
Input: A $\tau$-constrained latency graph $G_{L(a,b,\tau)}$ of a strongly connected SDFG $G$.
Output: "A schedule with maximal throughput (under constraint $\tau$) and (close
    to) minimal latency"
1.  Calculate $L^{min}(a, b)$ from the execution defined in Definition 6.5.
2.  Execute $G_{L(a,b,\tau)}$ in self-timed manner, and store the time of all the firings
    of actor dst.
3.  Execute $G_{L(a,b,\tau)}$ as follows
        - Fire all actors but src as soon as they are enabled.
        - Fire src (which is always enabled) if the time is $L^{min}(a, b)$ earlier than
          the time stored in Line 2 for the corresponding dst firing.
    return The schedule obtained from the execution of Line 3.
```

Theorem 6.2. *The schedule returned by algorithm optimizeThroughputLatency achieves maximal throughput under the given constraint τ.*

Proof. Due to consistency and strong connectedness of G, the throughput of all actors in G is in any execution proportionally related through their repetition vector entries as shown in Chapter 4. By construction of the throughput-constrained

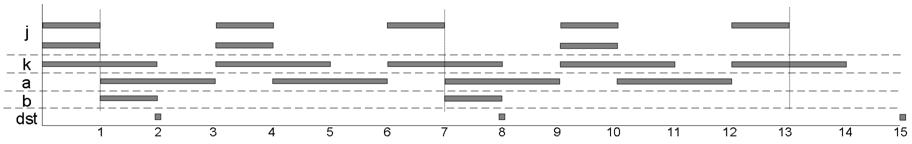


Figure 6.9: Self-timed execution.

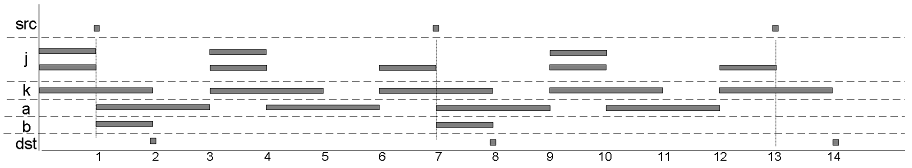


Figure 6.10: The execution result of *OptimizeThroughputLatency*.

latency graph, also the throughput of the *tc* and *dst* actors is in any execution of $G_{L(a,b,\tau)}$ proportionally related to the other actor throughputs through their repetition vector entries. (The throughput of *src* is unbounded because it has no input channels.) In the output schedule of *optimizeThroughputLatency*, actor *src* has by construction exactly the same throughput as actor *dst* has in the self-timed schedule, which is maximal for *dst*. Given the above observations and the fact that both *src* and *dst* have repetition vector entry one, this implies that *src* and *dst* have the same throughput in the output schedule that is maximal for *dst*. Hence also b has maximal throughput in the output schedule. ☐

Figure 6.10 illustrates algorithm *optimizeThroughputLatency* for the running example. The aim is to achieve maximal throughput. In this case, it is not necessary to explicitly model a throughput constraint in the graph. Figure 6.9 shows the self-timed execution of the latency graph, ignoring actor *src* (which in principle can fire infinitely often at time 0). This execution is known to provide the maximal throughput for b, which is $1/6$. Actor *dst* fires at times $2 + 6n$ for every $n \in \mathbb{N}_0$. The first of these firings does not need a firing of *src* and can therefore be ignored for latency purposes. Figure 6.10 shows the output of *optimizeThroughputLatency*. Actor *src* is scheduled at times $1 + 6n$, i.e., 7 time units (the minimum latency) before every *dst* firing in the self-timed execution except the first one. The result is a schedule that follows self-timed execution, with the *src* actor appropriately inserted. It achieves the minimal achievable latency of 7 and the maximal throughput of $1/6$. For each *src* firing, the latency spans the duration till the second subsequent *dst* firing, i.e., the latency exceeds the length of one period.

Table 6.1: Results: synthetic, single-processor.

	Min Lat	Arbitrary Order
Strongly Conn. Graphs	4.40	9.65
Min Buffer for non-zero Throughput	1.36	1.83
Min Buffer for Max Throughput	1.31	2.10

6.6 Experimental Results

In this section, we evaluate our scheduling schemes. In case of the single processor scheme, static order schedules with an arbitrary order of the concurrently enabled actors are used as a reference point. In the maximal parallelism scenario, the latency and throughput of the schemes of Definition 6.5 and of algorithm *optimizeThroughputLatency* are compared with those of the self-timed execution. Arbitrary single processor static order schedules can have a very poor latency, and we were not aware of any scheduling scheme resulting in an execution with optimum latency. Therefore, for each SDFG in the experiment, the generated static order schedules were constrained allowing only a single iteration of the SDFG in the periodic part of the schedule, and 100 different static orders were tested, choosing the best result.

We created a benchmark containing six real DSP and multimedia models and three sets of 300 synthetic SDFGs, generated using the SDF^3 tool [65]. We restricted ourselves to strongly connected graphs. These graphs are either strongly connected by construction or they have become strongly connected in the process of modeling the buffer sizes. The first set is composed of random strongly connected SDFGs. The second set contains graphs in which the dedicated storage capacity for channels is set to the minimum allowing non-zero throughput (computed via techniques from [64]). The third set contains SDFGs in which the buffer sizes for channels are set to the minimum which is enough to obtain the maximal achievable throughput [64]. All experiments were performed on a P4 PC running at 3.4Ghz.

Table 6.1 shows results for optimal latency single-processor schedules and the best of 100 randomly generated static order schedules. The latency entries are averaged over the entire set of models and normalized with respect to minimal achievable latency (without the single-processor constraint). Minimum latency execution improves latency between 26% and 54%. Recall that throughput is the same for both techniques.

Table 6.2 shows, for the synthetic graphs and for the maximal parallelism scheme, the latency, throughput of minimum latency execution (Definition 6.5), self-timed execution and latency optimized maximal throughput execution (Algorithm *optimizeThroughputLatency*). All entries show the average numbers taken over all 300 graphs of each set. The entries for latency are normalized with respect to the results of the minimum latency schedule and the throughput entries are

Table 6.2: Results: synthetic, maximal parallelism.

	Latency	Throughput
Strongly Connected Graphs		
Min latency	1	0.68
Self-timed	1.43	1
optimizeLatencyThroughput	1.12	1
Minimum Buffers and Throughput		
Min latency	1	0.90
Self-timed	1.54	1
optimizeLatencyThroughput	1.10	1
Maximal Throughput		
Min latency	1	0.78
Self-timed	1.31	1
optimizeLatencyThroughput	1.01	1
optimizeLatencyThroughput min latency-max throughput: 858/900 (95.3%)		

normalized with respect to the throughput achieved in the self-timed execution (i.e., the maximal achievable throughput). The self-timed schedule has a 31-54% on average higher latency than the minimum latency execution depending on the benchmark. In other words, minimum latency execution gives a significant latency reduction (24-35%) compared to self-timed execution. The price to be paid is a decrease in throughput of 10-32%. The latency optimized maximal throughput execution reduces the latency with 22-29% with respect to self-timed execution, while guaranteeing maximal throughput. The achieved latency is close to the minimally achievable latency (within 10% on average). In over 95% of the graphs the result combines minimal achievable latency with maximal achievable throughput. The average execution time of the SDF^3 tool to compute the schedule for a single SDFG for any of the scheduling algorithms is a few milliseconds.

We also experimented with SDFG models of actual DSP and multimedia applications. DSP domain applications are a modem and a sample-rate converter from [8], and a channel equalizer, and a satellite receiver from [58]. For the multimedia domain, we used an MP3 and an H.263 decoder from [64]. Table 6.3 shows the results, giving both average and worst-case values for latency and throughput.

The single-processor experiments show only a small latency improvement in one case. Due to the limited parallelism in the graphs, 100 randomly generated static order schedules was in five out of six cases sufficient to achieve optimal results.

Under the maximum parallelism scheme, we considered the application models both with minimal buffers for non-zero throughput and minimal buffers for maximal throughput. The average latency improvement of minimum latency execution with respect to self-timed execution is 10%, at a throughput loss of 33-44% on average. The satellite receiver, the modem, and the H.263 decoder do not show

Table 6.3: Results: DSP and multimedia benchmark.

	Lat (avg/worst)	Thr (avg/worst)
Single-processor results		
Min latency	1/1	1/1
Random-order	1/1.03	1/1
Minimum Buffers and Throughput		
Min latency	1/1	0.67/0.34
Self-timed	1.11/1.36	1/1
optimizeLatencyThroughput	1/1	1/1
Maximal Throughput		
Min latency	1/1	0.56/0.32
Self-timed	1.11/1.59	1/1
optimizeLatencyThroughput	1/1	1/1
optimizeLatencyThroughput min latency-max throughput: 6/6 (100%)		

any improvement. The channel equalizer (26%, minimal buffers) and the MP3 decoder (37%, maximal throughput) show the largest latency improvements (but also the largest throughput loss). However, applying algorithm *optimizeThroughputLatency* to achieve optimal latency for maximal throughput, achieves maximal throughput and minimal latency simultaneously in all cases. Execution times are all in the order of milliseconds and confirming the feasibility of the proposed techniques.

To test the hypothesis expressed in the introduction that latency optimization via a conversion to homogeneous SDFGs is often infeasible, we applied our techniques also to the HSDFG equivalents of our DSP and multimedia models. In two cases (satellite receiver, H.263 decoder), self-timed execution of Phase 2 of minimum latency execution (Definition 6.4), which in essence for HSDFGs is a critical path analysis taking into account parallel and pipelined execution that any potential HSDFG-based latency optimization technique has to perform (because all rates are one, only the longest path between source and destination actors determines the latency), takes several hours. This indeed renders HSDFG-based techniques prohibitively expensive in these cases.

6.7 Related Work

Latency have been defined for HSDFGs in different works [39, 44, 60]. In these references, latency is defined as the time difference between the first firing of a source actor and the corresponding firing of a destination actor. This notion of latency is mostly useful for a class of scheduling for HSDFGs, that are periodic from the beginning of the execution. In this class of schedules, known as *static periodic schedules* [56], only the first time difference between source and destination actors is important. However, the time distance between source and corresponding destination firings can vary a lot during an arbitrary execution of an SDFG graph. Therefore, we have extended the definition proposed by above mentioned references to the maximum of the time difference of the firings of a source and destination over all source and destination firings, grouping together part of a single graph iteration, as explained in Definition 6.3. This definition is more in line with the requirements of streaming multimedia applications. A more recent work [47] has defined latency as the time difference between any two firings of two actors. This means that data dependencies are not explicitly taken into account in this definition, and that the firings belonging to a single iteration of an SDFG are not considered in combination.

6.8 Summary

In this chapter, we have presented a technique to compute the minimum latency that can be achieved between firings of a designated pair of actors of some SDFG. The technique is based on an execution scheme that guarantees this minimum latency. We presented static-order schedules for single-processor platforms, and for a multi-processor context with sufficient resources to maximally exploit the available parallelism in an SDFG. The latter can be used as a feasibility check for application latency in any multi-processor design trajectory. The experimental evaluation shows that the latency computations and the underlying execution schemes are efficient. Compared to traditional scheduling techniques and execution schemes, substantial reductions in latency can be obtained, sometimes at the price of other performance metrics such as throughput. We showed that it is not always possible to simultaneously optimize latency and throughput. Therefore, we also presented a heuristic for optimizing latency under a throughput constraint. The heuristic gives optimal results for both latency and throughput simultaneously for all our real DSP and multimedia models, and for over 95% of our synthetic models. Interesting options for future work are the exploration of the entire latency-throughput trade-off space, analysis and optimization of latency between multiple pairs of source and destination actors, scheduling schemes for multiprocessor systems without assuming maximal parallelism, and extensions taking into account other dimensions like buffer sizes and code size.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Demands for more complex functionality from multimedia systems is increasing constantly. At the same time, the price of these products should not increase and the time-to-market is expected to decrease. This implies the design of more complex multimedia applications, which, following technology and design trends, need to run on multiprocessor systems. To achieve shorter time-to-market and reliable products, predictable design is proposed as a promising way to facilitate the design and verification process by only focusing on the subset of design solutions which are guaranteed to have desirable specifications.

Predictable design cannot be realized without the support of models which are amenable to formal analysis. A very useful means to model streaming multimedia applications are Synchronous Data Flow Graphs (SDFGs). SDFGs are interesting because they provide a very good combination of expressivity and analysis potential.

An SDFG is a graph with actors as vertices and channels as edges. Actors represent basic parts of an application which need to be executed. Channels represent data dependencies between actors. SDFGs have traditionally been used for modeling Digital Signal Processing (DSP) applications which are structurally very similar to streaming multimedia applications. Some analysis techniques have been developed for analyzing different performance metrics of DSP applications modeled in SDFGs. However, these techniques do not scale well for SDFGs with larger actor execution times and data production and consumption rates, typically required to model streaming multimedia applications.

Streaming multimedia applications inherently work on virtually infinitely long streams of data which needs to be processed. Therefore, SDF models with complete or partial deadlock are not of interest. At the same time, as these models are to be implemented in practice, the memory needed for implementing these models should also be finite. Chapter 3 introduces necessary and sufficient conditions for checking SDFGs for liveness and boundedness, i.e., for checking whether an SDFG can be executed indefinitely using only a finite amount of memory.

Throughput is a prominent performance metric, which has already been studied in the literature of SDFGs. All of the studied approaches require a conversion of the considered SDFG to a subclass of SDFGs, namely Homogeneous SDFGs (HSDFGs). This conversion often results in a dramatic (exponential) increase in the size of the graph, making throughput analysis prohibitively expensive in computation time or even infeasible in some occasions. Chapter 4 proposes a method based on state-space exploration, which works directly on SDFGs avoiding the conversion to HSDFGs. Despite the theoretical worst-case complexity, which is exponential in the size of the graph, the method proves to be very fast in our experimental results, mainly because, typically, only a very limited number of states need to be actually stored and kept in memory during the processing.

Even though throughput analysis proposed in Chapter 4 is fast, for applications such as run-time reconfiguration of a system, sufficient computation and memory resources may not be available to perform the analysis on an embedded system. For such applications, we propose parametric throughput analysis techniques in Chapter 5, where actor execution times can be linear expressions of some given parameters. In this way, throughput of an SDFG is expressed in the form of a simple function of these parameters. Evaluation of this function is computationally much cheaper than the recalculation of throughput from scratch. Furthermore, the parametric throughput analysis can be used for optimization purposes like performance bottle-neck analysis, where we are interested in analysis of throughput variations under varying actor execution times. Experimental results show that parametric throughput analysis is feasible.

Although throughput is a salient performance metric, not all timing aspects of applications can be analyzed using only throughput. For example, for interactive applications like telephony, video conferencing and gaming, besides throughput, latency plays an important role as well. In Chapter 6, we extend the existing definition of latency for HSDFGs to SDFGs. Furthermore, we propose an algorithm for determining schedules which result in executions with provably minimum latency between any two given actors. We show that it is not in general possible to simultaneously minimize latency and maximize throughput. We provide a heuristic algorithm for minimizing latency under a throughput constraint.

7.2 Open Problems and Future Research

The analysis techniques presented in this thesis provide methods to calculate performance metrics, throughput and latency in particular, as well as their optimization. An interesting future research direction is the investigation of the simultaneous optimization of these performance metrics together with non-timing-related metrics such as buffer sizes or code size. In general, such analyses will lead to trade-off spaces between the various metrics. Also, we did not consider architecture properties in our performance analyses, which is an interesting direction to look into for future research. Finally, parametric analysis has only been developed for throughput with parametric actor execution times. Other aspects could be parameterized, and other metrics could be considered in future work.

In the following, several concrete issues are mentioned which are interesting to investigate.

- In Chapter 6, analysis techniques are proposed, assuming two platforms, namely, single processor platforms and multiprocessors with sufficiently many processors to exploit all the parallelism in a graph. Our techniques can be extended for general multiprocessor platforms that do not necessarily allow for the simultaneous execution of all enabled actors.

- In general, certain aspects of the target platform of SDFG applications can be modeled in the graph structure of SDFGs. However, not all properties can be modeled only by using SDFGs. Resource sharing and various types of schedulers are for example aspects that cannot always be captured in SDFGs. Our analysis techniques can be extended to take such architecture properties into account, for example in line with the extension of the throughput analysis for TDMA and static-order scheduling techniques already proposed in [62, 61].

- Section 6.5 provides a heuristic algorithm for optimizing latency under a given throughput constraint. The study of the whole throughput and latency trade-off space is a very interesting open issue.

- The trade-off between throughput and channel buffer sizes is studied in [64, 66]. Channel buffer sizes were not considered in this thesis. Since the techniques of [64, 66] are based on the same state-space-based analysis underlying the techniques in this thesis, adding the buffer-sizes dimension to the latency and throughput trade-off space seems possible.

- In this thesis, we assumed unbounded buffers for channels. If needed, channel capacities were modeled using backward channels. This model implies the use of separate storage space for each channel. Assuming shared storage space for all the channels may be more memory efficient. Analysis of the minimum required shared storage for deadlock-free execution of SDFGs has already been explored in [24]. Finding a minimum amount of shared storage for all channels while achieving maximum throughput or minimum latency is an interesting direction to achieve efficiency of memory use.

- Chapter 5 provides parametric throughput analysis for SDFGs which is an interesting starting point for analyzing the behavior of an application under changes due to the inherent dynamism. However, we only assumed parametric expressions for actor execution times, whereas different aspects of SDFGs, like rates and structure of the graphs can be parameterized. Extensions in this direction would provide parametric analysis techniques for computational models more general than SDFGs such as cyclo-static dataflow graphs (CSDF, [9]), parameterized synchronous dataflow (PSDF, [7]), variable-rate dataflow (VRDF, [72]), scenario-aware dataflow (SADF, [69]) or reactive process networks (RPN, [23]).

The above list of research topics emphasizes the further development of analysis techniques for dataflow models, for which this thesis provides a starting point. Besides the further development of analysis techniques, also their application into predictable design flows for embedded multimedia systems, such as proposed and developed in [61], and in predictable run-time resource and quality management, in line with the techniques proposed in [54], are of interest.

Bibliography

[1] M. Adé, R. Lauwereins, and J.A. Peperstraete. Data minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *34th Design Automation Conference, DAC 97, Proceedings*, pages 64–69. ACM, 1997.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity* *`http://www-rocq.inria.fr/metalau/cohen/SED/book-online.html`*. Wiley, 2001.

[4] S.K. Berberian. *Linear Algebra*. Oxford University Press, 1992.

[5] M. Berkelaar, K. Eikland, and P. Notebaert. lpsolve. C library, `http://www.geocities.com/lpsolve/`.

[6] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[7] B. Bhattacharya and S.S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.

[8] S. Bhattacharyya, P. Murthy, and E.A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, 1999.

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

[10] F.D.J. Bowden. A brief survey and synthesis of the roles of time in Petri nets. *Mathematical and Computer Modelling*, 31(10):55–68, 2000.

[11] J. Campos, G. Chiola, and M. Silva. Ergodicity and throughput bounds for Petri nets with unique consistent firing count vector. *IEEE Transactions on Software Engineering*, 17(2):117–125, 1991.

[12] D.Y. Chao, M. Zhou, and D.T. Wang. Multiple weighted marked graphs. *Preprints of 12th IFAC World Congress, Sydney, Australia*, 4:259–263, July 1993.

[13] J. Cochet-Terrasson, G. Cohen, G. Gaubert, and J.-P. Quadrat. Numerical computations of spectral elements in max-plus algebra. In *International Conference on System Structure and Control, Proceedings*, pages 667–674. Elsevier, 1998.

[14] F. Commoner, A.W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(5):511–523, October 1971.

[15] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, 1991.

[16] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):385–418, October 2004.

[17] A. Dasdan and R.K. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.

[18] A. Dasdan, S.S. Irani, and R.K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *36th Design Automation Conference, DAC 99, Proceedings*, pages 37–42. ACM, 1999.

[19] H. Edelsbrunner, L.J. Guibas, and M. Sharir. The upper envelope of piecewise linear functions: Algorithms and applications. *Discrete & Computational Geometry*, 4:311–336, 1989.

[20] J. Esparza. Decidability and complexity of Petri net problems - an introduction. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer-Verlag, 1998.

[21] K. Fukuda. cddlib. C++ library, Swiss Federal Institute of Technology, `http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html`.

[22] S. Garg and D. Marculescu. System-level process variation driven throughput analysis for single and multiple voltage-frequency island designs. In *Conference on Design Automation and Test in Europe, DATE 07, Proceedings*, pages 403–408. IEEE, 2007.

[23] M.C.W. Geilen and T. Basten. Reactive process networks. In *4th International Conference on Embedded Software, EMSOFT 04, Proceedings*, pages 137–146. ACM, 2004.

[24] M.C.W. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model-checking. In *42nd Design Automation Conference, DAC 05, Proceedings*, pages 819–824. ACM, 2005.

[25] A.H. Ghamarian, M.C.W. Geilen, T. Basten, and S. Stuijk. Parametric throughput analysis of synchronous data flow graphs. Technical report ESR-2007-08, TU Eindhoven, http://www.es.ele.tue.nl/esreports/, 2007.

[26] A.H. Ghamarian, M.C.W. Geilen, T. Basten, and S. Stuijk. Parametric throughput analysis of synchronous data flow graphs. In *Conference on Design Automation and Test in Europe, DATE 08, Proceedings*, pages 116–121. IEEE, 2008.

[27] A.H. Ghamarian, M.C.W. Geilen, T. Basten, B. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. Technical report ESR-2006-04, TU Eindhoven, http://www.es.ele.tue.nl/esreports/, 2006.

[28] A.H. Ghamarian, M.C.W. Geilen, T. Basten, B.D. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *6th International Conference on Formal Methods in Computer Aided Design, FMCAD 06, Proceedings*, pages 68–75. IEEE, 2006.

[29] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M.J.G. Bekooij, B.D. Theelen, and M.R. Mousavi. Throughput analysis of synchronous data flow graphs. In *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 25–36. IEEE, 2006.

[30] A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen, and B.D. Theelen. Latency minimization for synchronous data flow graphs. In *10th Euromicro Conference on Digital System Design, DSD 07, Proceedings*, pages 189–196. IEEE, 2007.

[31] A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen, and B.D. Theelen. Latency minimization for synchronous data flow graphs. Technical report ESR-2007-04, TU Eindhoven, http://www.es.ele.tue.nl/esreports/, 2007.

[32] S.V. Gheorghita, T. Basten, and H. Corporaal. Application scenarios in streaming-oriented embedded system design. In *International Symposium on System-on-Chip, SoC 06, Proceedings*, pages 175–178. IEEE, 2006.

[33] R. Govindarajan and G.R. Gao. Rate-optimal schedule for multi-rate DSP computations. *Journal of VLSI signal processing*, 9:211–235, 1995.

[34] A. Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation.* Morgan Kaufmann, 2003.

[35] D.B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[36] N.D. Jones, L.H. Landweber, and Y.E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4(3):277–299, 1977.

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

[38] R.M. Karp and R.E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, November 1966.

[39] S.Y. Kung. *VLSI Arrays Processors.* Prentice Hall, 1988.

[40] S.Y. Kung, P.S. Lewis, and S.C. Lo. Performance analysis and optimization of VLSI dataflow arrays. *Journal of Parallel Distributed Computing*, 4(6):592–618, 1987.

[41] E.A. Lee. *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors.* PhD thesis, University of California, Berkeley, June 1986.

[42] E.A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

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

[44] V. Madisetti. *VLSI Digital Singal Processors.* IEEE Press, 1995.

[45] P.M. Merlin. *A Study of Recoverability of Processes.* PhD thesis, Department of Information and Computer Science, University of California at Irvine, 1975.

[46] mmcycle. `http://elib.zib.de/pub/Packages/mathprog/netopt/mmc-info`.

[47] O.M. Moreira and M.J.G. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, vol. 2007, Article ID 83710, 14 pages, 2007.

[48] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[49] H. Oh and S. Ha. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI Signal Processing*, 37(1):41–51, 2004.

[50] T.M. Parks. *Bounded Scheduling for Process Networks*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1995.

[51] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee. A hierarchical multiprocessor scheduling system for DSP applications. In *Conference on Signals, Systems and Computers, Proceedings*, pages 122–126. IEEE, 1995.

[52] J. Plavka. l-parametric eigenproblem in max-algebra. *Discrete Applied Mathematics*, 150(1):16–28, 2005.

[53] 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 *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 03, Proceedings*, pages 63–72. ACM, 2003.

[54] P. Poplavko, T. Basten, and J. van Meerbergen. Execution-time prediction for dynamic streaming applications with task-level parallelism. In *10th Euromicro Conference on Digital System Design, DSD 07, Proceedings*, pages 228–235. IEEE, 2007.

[55] L. Popova-Zeugmann. On liveness and boundedness in time Petri nets. In *Proceedings of the Workshop on Concurrency, Specification and Programming (CS&P'95)*, pages 136–145, 1995.

[56] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, October 1968.

[57] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed constraints. *IEEE Transactions on Circuits and Systems*, 28:196–202, 1981.

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

[59] J. Sifakis. Use of Petri nets for performance evaluation. In *Measuring, modelling and evaluating computer systems Proceedings*, pages 75–93. Elsevier Science, 1977.

[60] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc, New York, NY, USA, 2000.

[61] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors.* Ph.D. dissertation, Eindhoven University of Technology, 2007.

[62] S. Stuijk, T. Basten, M.C.W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *44th Design Automation Conference, DAC 07, Proceedings*, pages 777–782. ACM, 2007.

[63] S. Stuijk, T. Basten, B. Mesman, and M.C.W. Geilen. Predictable embedding of large data structures in multiprocessor networks-on-chip. In *8th Euromicro Conference on Digital System Design, DSD 05, Proceedings*, pages 388–395. IEEE, 2005.

[64] S. Stuijk, M.C.W. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *43th Design Automation Conference, DAC 06, Proceedings*, pages 899–904. ACM, 2006.

[65] S. Stuijk, M.C.W. Geilen, and T. Basten. SDF^3: SDF For Free. In *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 276–278. IEEE, 2006.

[66] S. Stuijk, M.C.W. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 2008.

[67] W. Sung, J. Kim, and S. Ha. Memory efficient software synthesis from dataflow graphs. In *International symposium on System Synthesis, ISSS'98, Proceedings*, pages 137–144. IEEE, 1998.

[68] E. Teruel, P. Chrzastowski, J. M. Colom, and M. Silva. On weighted T-systems. In Jensen, K., editor, *13th Internationald Conference on Application and Theory of Petri Nets 1992, Sheffield, UK*, volume 616 of Lecture Notes in Computer Science, pages 348–367. Springer-Verlag, 1992.

[69] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *4th International Conference on Formal Methods and Models for Co-Design, MEMOCODE 06, Proceedings*, pages 185–194. IEEE, 2006.

[70] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *4th International Conference on Hardware-Software Codesign and System Synthesis, CODES+ISSS 06, Proceedings*, pages 10–15. ACM, 2006.

[71] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *44th Design Automation Conference, DAC 07, Proceedings*, pages 658–663, New York, NY, USA, 2007. ACM.

[72] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Computation of buffer capacities for throughput constrained and data dependent inter-task communication. In *Conference on Design Automation and Test in Europe, DATE 08, Proceedings*, pages 640–645. IEEE, 2008.

[73] N.E. Young, R.E. Tarjan, and J.B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, 1991.

Curriculum Vitae

Amir Hossein Ghamarian was born on 2nd of October in Tehran, Iran. In 1996, he graduated from Peyvan highschool in Mathematics and Physics. He received his Bachelor in Computer Engineering (software) and Masters in Computer science from Sharif University of Technology, Tehran, Iran, in 2000, and 2002 respectively. Since February 2003, he has been a Ph.D. student within the electronic systems group at the Electrical Engineering Department of Technische Universiteit Eindhoven (TU/e) in Eindhoven, The Netherlands. His research was funded by NWO (The Dutch Organization for Scientific Research) within project PROMES: Programming multi-processor embedded multi-media systems. His research has led amongst others to several publications and this thesis. Amir is currently a post-doctoral researcher, continuing his research in the electronic systems group at the Electrical Engineering Department of the Technische Universiteit Eindhoven.

List of Publications

First author

- A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M.J.G. Bekooij, B.D. Theelen, and M.R. Mousavi. Throughput analysis of synchronous data flow graphs. In *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 25–36. IEEE, 2006.

- A.H. Ghamarian, M.C.W. Geilen, T. Basten, B.D. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *6th International Conference on Formal Methods in Computer Aided Design, FMCAD 06, Proceedings*, pages 68–75. IEEE, 2006.

- A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen, and B.D. Theelen. Latency minimization for synchronous data flow graphs. In *10th Euromicro Conference on Digital System Design, DSD 07, Proceedings*, pages 189–196. IEEE, 2007.

- A.H. Ghamarian, M.C.W. Geilen, T. Basten, and S. Stuijk. Parametric throughput analysis of synchronous data flow graphs. In *Conference on Design Automation and Test in Europe, DATE 08, Proceedings*, pages 116–121. IEEE, 2008.

Co-author

- S. Stuijk, T. Basten, M.C.W. Geilen, A.H. Ghamarian and B.D. Theelen. Resource-efficient routing and scheduling of time-constrained network-on-chip communication. In *9th Euromicro Conference on Digital System Design, DSD 06, Proceedings*, pages 45–52. IEEE, 2006.

- S. Stuijk, T. Basten, M.C.W. Geilen, A.H. Ghamarian, and B.D. Theelen. Resource-efficient routing and scheduling of time-constrained streaming communication on networks-on-chip. In *Journal of Systems Architecture*, 54(3-4):411-426, Elsevier, March-April 2008. (Special issue with selected best papers of DSD 2006.)