Abstractions for Aperiodic Multiprocessor Scheduling

of Real-Time Stream Processing Applications

Joost P.H.M. Hausmans

Members of the graduation committee:

Prof. dr. ir.	M. J. G. Bekooij	University of Twente (promotor)
Prof. dr. ir.	G. J. M. Smit	University of Twente
Prof. dr.	J. C. van de Pol	University of Twente
Dr. ir.	M. C. W. Geilen	Eindhoven University of Technology
Prof. dr.	S. Chakraborty	Technische Universität München
Prof. dr.	L. Thiele	ETH Zürich
Prof. dr.	P. M. G. Apers	University of Twente (chairman and secretary)

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science, Computer Architecture for Embedded Systems (CAES) group.

This work was carried out at NXP Semiconductors in a project of NXP Semiconductors Research.

CTIT Ph.D. Thesis Series No. 15-351
Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE Enschede, The Netherlands

This research has been conducted within the Netherlands Streaming (NEST) project (project number 10346). This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs.

Copyright © 2015 Joost P.H.M. Hausmans, Eindhoven, The Netherlands. This work is licensed under the Creative Commons Attribution 4.0 International License.
<http://creativecommons.org/licenses/by/4.0/>.

This thesis was typeset using $\LaTeX$, Ipe, TikZ and Kile.
This thesis was printed by De Budelse, The Netherlands.

ISBN 978-90-365-3853-4
ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. 15-351)
DOI 10.3990/1.9789036538534

ABSTRACTIONS FOR APERIODIC MULTIPROCESSOR SCHEDULING

OF REAL-TIME STREAM PROCESSING APPLICATIONS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 24 april 2015 om 16.45 uur

door

Jozef Paulus Hermanus Marie Hausmans

geboren op 6 april 1987
te Echt

Dit proefschrift is goedgekeurd door:

Prof. dr. ir. M.J. G. Bekooij (promotor)

Abstract

Embedded multiprocessor systems are often used in the domain of real-time stream processing applications to keep up with increasing power and performance requirements. Examples of such real-time stream processing applications are digital radio baseband processing and WLAN transceivers.

These stream processing applications often have a dynamic character. For example the execution times and execution rates of the tasks of the stream processing applications vary and can even be data dependent. To cope with this dynamic behavior, the tasks are executed on the multiprocessor system in a data-driven fashion on run-time schedule resources.

Another important aspect of real-time stream processing applications are their strict performance constraints. A periodic source or sink imposes a throughput constraint and also latency constraints are common. For stream processing applications, violating these constraints typically leads to a major reduction of the quality of service of the applications.

To prevent such violations of the temporal constraints, analysis methods are used. These analysis methods ease the processes of dimensioning, programming and optimizing the multiprocessor systems within these temporal constraints. Analysis methods rely on accurate abstractions of the analyzed applications. However, current abstractions have a limited accuracy and applicability and do therefore not always suffice.

We focus in this thesis on dataflow analysis techniques. Dataflow analysis is the standard analysis technique for stream processing applications. However, the dataflow abstractions have shortcomings which limit their applicability and relevance.

To use abstractions on the temporal behavior of applications, a formal abstraction/refinement theory is required. Existing theories assume that the temporal behavior of applications is orthogonal to their functional behavior. However, this orthogonality does not always hold. We introduce a new abstraction/refinement

theory in which also the functional behavior is coupled which makes explicit under what conditions refinement is allowed.

Data-driven multiprocessor systems have the benefit that they do not have to be dimensioned for the isolated worst-case situation that can occur. Consecutive executions of tasks can compensate each other which can be used to improve the temporal behavior of applications. However, dataflow analysis techniques only consider so called worst-case execution times of tasks which means that they ignore variation of executions times when the temporal behavior of the application is analyzed. A solution is proposed in this thesis to take information about varying execution times into account to improve the accuracy of dataflow analysis techniques.

Next to that, dataflow analysis techniques focus on pipeline parallelism only, to exploit the parallelism of the multiprocessor platform. The combination with data parallelism is however beneficial but requires different dataflow modeling techniques. A technique is proposed to model data parallelism in dataflow models without replicating dataflow actors. This allows to determine the required amount of data parallelism using existing dataflow analysis techniques. Furthermore, using this new modeling technique, the trade-off between pipeline parallelism, data parallelism and required buffer sizes can be made.

Furthermore, one of the biggest shortcomings in the applicability of dataflow analysis techniques is the limited class of supported run-time schedulers. Current analysis methods only support run-time schedulers for which it is possible to determine worst-case response times of tasks independent of the enablings of tasks. We extend the scope of dataflow analysis techniques by presenting a temporal analysis flow which combines dataflow analysis techniques with a broader class of run-time schedulers including for example static priority preemptive run-time schedulers. The presented analysis techniques do support the combination of cyclic data dependencies in conjunction with cyclic resource dependencies.

The last topic that is addressed in this thesis concerns the compositionality of temporal analysis methods. The temporal behavior of dataflow models is not compositional in general. In fact, a compositional temporal analysis model that supports arbitrary cyclic dependencies between tasks is lacking. We present a new temporal analysis model which has powerful properties with respect to the hierarchical composition and incremental design of real-time stream processing applications.

In this thesis we will present abstractions for multiprocessor systems in which the tasks have aperiodic schedules. These aperiodic schedules can capture the dynamic behavior of the real-time stream processing applications. We present accurate abstractions based on dataflow analysis techniques which can be used for a large class of multiprocessor systems. Compared to state of the art, we broaden the scope of dataflow analysis techniques, improve their accuracy and provide a new higher level of abstraction.

Samenvatting

Ingebedde multiprocessor systemen worden vaak gebruikt in het domein van real-time stream processing applicaties om tegemoet te komen aan hun toenemende prestatie-eisen en hun toenemende energieverbruik. Voorbeelden van zulke real-time stream processing applicaties zijn digitale radio basisband processors en WLAN zendontvangers.

Deze stream processing applicaties hebben vaak een dynamisch karakter. De uitvoeringstijden en de uitvoersnelheden van de taken van deze stream processing applicaties variëren bijvoorbeeld en kunnen zelfs data afhankelijk zijn. Om te kunnen omgaan met dit dynamische gedrag worden de taken op het multiprocessor systeem, datagedreven uitgevoerd op run-time geregelde systeemelementen.

Andere belangrijke aspecten van real-time stream processing applicaties zijn hun strikte prestatie eisen. Een periodieke source of sink legt een eis op aan de verwerkingssnelheid en ook latency eisen komen voor. Voor stream processing applicaties leidt het schenden van deze eisen typisch gezien tot een aanzienlijke vermindering van de kwaliteit van de applicatie.

Om zulke schendingen van de temporele eisen te voorkomen worden analysemethoden gebruikt. Deze analysemethoden vergemakkelijken het proces van de dimensionering, het programmeren en het optimaliseren van multiprocessor systemen binnen de temporele eisen. Analysemethoden vertrouwen op accurate abstracties van de geanalyseerde applicaties. De huidige abstracties hebben echter een beperkte nauwkeurigheid en toepasbaarheid en voldoen daarom niet altijd.

We focussen in deze thesis op dataflow analysetechnieken. Dataflow analyse is de standaard analysetechniek voor stream processing applicaties. Deze dataflow abstracties hebben echter hun tekortkomingen waardoor hun toepasbaarheid en relevantie beperkt wordt.

Om abstracties op het temporele gedrag van applicaties te kunnen gebruiken is een formele abstractie/verfijningstheorie nodig. Huidige theoriën nemen aan

dat het temporele gedrag van applicaties orthogonaal is aan hun functionele gedrag. Deze orthogonaliteit gaat echter niet altijd op. We introduceren daarom een nieuwe abstractie/verfijningstheorie waarin het functionele gedrag gekoppeld is waardoor het expliciet wordt onder welke omstandigheden een verfijning toegestaan is.

Data gedreven multiprocessor systemen hebben het voordeel dat ze niet gedimensioneerd hoeven te worden voor de slechts mogelijke situatie die in isolatie kan optreden. Opeenvolgende uitvoeringen van taken kunnen elkaar namelijk compenseren. Dit gegeven kan gebruikt worden om het temporele gedrag van applicaties te verbeteren. Echter de huidige dataflow analysetechnieken beschouwen alleen maar de slechts mogelijke uitvoeringstijden van taken wat betekent dat ze variatie in deze uitvoeringstijden negeren wanneer het temporele gedrag van applicaties wordt geanalyseerd. Een oplossing hiervoor wordt in deze thesis voorgesteld waarbij informatie over variërende uitvoeringstijden wordt meegenomen om de nauwkeurigheid van dataflow analysetechnieken te verbeteren.

Dataflow analysetechnieken focussen daarnaast alleen maar op pipeline parallelisme om het parallelisme van een multiprocessor platform te gebruiken. De combinatie met data parallelisme is echter lucratief maar vereist andere dataflow modelleringstechnieken. Een techniek wordt voorgesteld om data parallelisme te modelleren in dataflow modellen zonder dat dataflow actoren gerepliceerd hoeven te worden. Hierdoor kan de benodigde hoeveelheid data parallelisme bepaald worden met bestaande dataflow analysetechnieken. Daarnaast kunnen we met deze nieuwe modelleertechniek de afweging maken tussen pipeline parallelisme, data parallelisme en de benodigde buffer groottes.

Daarnaast is één van de grootste tekortkomingen van de toepasbaarheid van dataflow analysetechnieken de beperkte klasse van ondersteunde run-time schedulers. Huidige analysemethoden ondersteunen alleen maar run-time schedulers waarvoor het mogelijk is om de slechts mogelijke reactietijd van taken te bepalen onafhankelijk van hoe vaak ander taken uitvoeren. We breiden het toepassingsgebied van dataflow analysetechnieken uit door een temporele analyse flow te presenteren die dataflow analysetechnieken combineert met een bredere klasse van run-time schedulers. De gepresenteerde analysetechnieken ondersteunen de combinatie van cyclische data afhankelijkheden in combinatie met cyclische resource afhankelijkheden.

Het laatste onderwerp dat besproken wordt in deze thesis heeft betrekking op de compositionaliteit van temporele analysemethoden. Het temporele gedrag van dataflow modellen is niet compositioneel in het algemene geval. Een compositioneel temporeel analyse model dat ondersteuning heeft voor willekeurige cyclische afhankelijkheden tussen taken mist zelfs helemaal. We presenteren een nieuwe temporeel analyse model dat krachtige eigenschappen heeft met betrekking tot hiërarchische compositie alsmede het incrementeel ontwerpen van real-time stream processing applicaties.

In deze thesis presenteren we abstracties voor multiprocessor systemen waarin de taken aperiodieke schedules hebben. In deze aperiodieke schedules is het mo-

gelijk om het dynamische gedrag van de real-time stream processing applicaties vast te leggen. We presenteren accurate abstracties gebaseerd op dataflow analysetechnieken welke gebruikt kunnen worden voor een brede klasse van multiprocessor systemen. In vergelijking met state-of-the-art verbreden we het toepassingsgebied van dataflow analysetechnieken, verbeteren we hun nauwkeurigheid en we introduceren een nieuw abstractieniveau bovenop dataflow analysemethoden.

Dankwoord

Dit is dan echt het laatste stuk van deze thesis dat ik schrijf. Na vier jaar promoveren zit het er op. De totstandkoming van deze thesis is een heel avontuur geweest, een avontuur dat eigenlijk ruim vijf jaar geleden begon. Bij het zoeken naar een geschikt afstudeerproject voor mijn studie Embedded Systems aan de TU/e kwam ik via Benny en Andreas terecht bij NXP Research. Hier stond Marco klaar om mij met een interessante en uitdagende afstudeeropdracht enthousiast te maken voor het onderzoeksvak. Toen ik begon aan dit afstudeerproject had ik nooit gedacht dat de afgelopen vijf jaar zo zouden verlopen. Een interessante en leerzame tijd heeft uiteindelijk geleid tot het 'boekje' dat nu in uw handen ligt.

Natuurlijk heb ik dit niet alleen kunnen doen en graag zou ik dan ook iedereen willen bedanken die heeft bijgedragen aan dit resultaat. Een aantal van deze mensen wil ik hieronder nog persoonlijk bedanken.

Er is één persoon die het meeste heeft bijgedragen aan mijn promotie en dat is Marco. Ik denk dat er maar weinig promovendi zijn die begeleid worden op een manier zoals Marco het doet. Als promotor en dagelijkse begeleider hebben wij veelvuldig contact gehad. Het begon met een introductie in dataflow analysetechnieken en heeft geleid tot diep inhoudelijke discussies over allerhande abstracties. Marco is altijd nauw betrokken geweest bij al mijn werk en zijn kennis van zowel de praktische zaken als de academische state-of-the-art van meerdere vakgebieden hebben telkens geleid tot nieuwe interessante onderzoeksrichtingen.

Ook wil ik graag alle andere personen uit de CAES vakgroep bedanken. In het bijzonder Gerard die mij de kans heeft geboden om bij de vakgroep CAES te kunnen promoveren. Ondanks dat ik vanwege mijn standplaats in Eindhoven mij niet heel vaak heb laten zien in Enschede, heb ik mij altijd welkom gevoeld bij CAES. Verder heb ik veel gehad aan de L^AT_EX-template van Jochem en zijn voorgangers. Van deze template heb ik grote delen kunnen gebruiken om het maken van deze thesis te stroomlijnen. Daarnaast bedank ik Thelma, Nicole en Marlous die altijd klaar stonden om ondersteuning te bieden wanneer dat nodig was.

Natuurlijk zijn er naast Marco meerdere mensen inhoudelijk betrokken geweest bij mijn thesis. Als eerste Stefan, waar ik sinds het begin van mijn studie aan de TU/e mee samenwerk. Deze samenwerking hebben we tijdens onze promotietrajecten met succes voort weten te zetten. De vele koffiepauzes, kritische blikken op elkaars werk en verhelderende discussies hebben allemaal bijgedragen aan het tot stand komen van deze thesis. Ten tweede kan ik natuurlijk Maarten niet vergeten. Maarten heeft vanaf de andere kant van de oceaan zijn kennis en ervaring veelvuldig gebruikt om te zorgen voor nieuwe inzichten en verbeteringen van mijn werk. Verder heb ik veel gehad aan de interessante werkoverleggen die ik dagelijks met Tjerk *und* Philip hield tijdens onze koffiepauzes. Daarnaast leverden Sunil, Koen en Peter als afstudeerders een bijdrage. Hun afstudeerprojecten hebben voor de nodige discussies gezorgd en vele nieuwe inzichten opgeleverd. Als laatste van de direct betrokkenen wil ik graag de collega's bij NXP Research bedanken. Eerst de collega's in de Distributed System Architectures groep en na de groepswisseling ook de collega's van de Signal Processing groep.

Naast de hulp uit de werksfeer heb ik veel steun gekregen van vrienden en familie. Vanwege hun achtergrond zal ik sommige van hen in het Limburgs bedanken.

Om te beginne mien aojers vöör de steun die ze mich altied höbbe gebaoje. Ze höbbe mich gestimuleerdj om zoa good mäögelik mien bès te doon en ze höbbe elke keus die ich höb gemaaktj onvöörwaardelik gesteundj. Zónger hun vertroewe haj ich 't neet kènne doon.

Ouch zónger mien breurs, vrunj en femilie haw ich noats gekèndj. De vöole vekansies, fieëskes, oetstepkes, optraejes, etc. höbbe altied vöör de nuuedige aafleijing gezörgdj. Ondanks det ich de lètste maondje geregeldj aafwezig woor of vreug nao hoes ging, zeen zie mich altied blieve sjteune. De gemiste uurkes waere de kómmende tied zónger twiefel met alle plezeer weer ingehaoldj. In 't bezunjer wil ich mien twieë paranimfe Bert en Thijs alvas hertelik bedanke vöör hun steun.

Als laatste wil ik heel graag Iris bedanken. Ondanks de drukke tijd van het afgelopen jaar en de soms onmogelijke werktijden ben je mij blijven steunen en heb je mij telkens weer aangespoord om nog even door te gaan. Heel erg bedankt voor jouw geduld en steun.

Joost Hausmans
Eindhoven, april 2015

Contents

1	Introduction	1
1.1	Embedded Multiprocessor Systems	2
1.2	Analysis Techniques for Data-Driven Multiprocessor Systems . .	5
1.3	Problem Statement	10
1.4	Contributions	11
1.5	Outline	11
2	Dataflow Analysis	13
2.1	Notation	14
2.2	Task Graphs	15
2.3	Dataflow Models and Their Properties	16
2.4	Timed Actors	23
2.5	Dataflow Modeling	37
2.6	Dataflow Analysis	52
2.7	Summary	57

3	Unified Dataflow Model for Data and Pipeline Parallelism	59
3.1	Related Work	61
3.2	Basic Idea	62
3.3	Implementation	65
3.4	Modeling Data Parallelism	70
3.5	Determining Data and Pipeline Parallelism	79
3.6	Case-Study	82
3.7	Conclusion	83
4	Two Parameter Workload Characterization	85
4.1	Related Work	86
4.2	Basic Idea	88
4.3	Dataflow Analysis Using the (σ, ρ) Workload Characterization	93
4.4	Cyclic Temporal Behavior	103
4.5	Case Study	105
4.6	Conclusion	110
5	Dataflow Analysis for Non-Starvation-Free Schedulers	113
5.1	Related Work	114
5.2	Analysis Flow	117
5.3	Application Characteristics	118
5.4	Response Times	119
5.5	Temporal Analysis	123
5.6	Linearized Response Time	136
5.7	Buffer Sizing	140
5.8	Evaluation	144
5.9	Case-Study	147
5.10	Conclusion	153

6	Compositional Temporal Analysis Model	155
6.1	Basic Idea	156
6.2	Component model	161
6.3	Composition	162
6.4	Practical applications of the CTA model	168
6.5	Related work	180
6.6	Conclusion	181
7	Conclusion	183
7.1	Summary	184
7.2	Contributions	186
7.3	Possible Future Directions	187
	List of Acronyms	189
	List of Symbols	191
	Bibliography	193
	List of Publications	205

Introduction

Embedded multiprocessor systems are often used in the domain of real-time stream processing applications to keep up with increasing power and performance requirements. Examples of these applications are digital radio baseband processing and WLAN transceivers.

Execution times of the tasks and processing rates of the tasks vary and can even be data dependent. The systems on which these applications execute have to support this dynamic behavior. This is ensured by executing the tasks of the stream processing applications in a data-driven fashion on run-time scheduled resources. Applications are implemented as task graphs, with the tasks of the task graphs communicating via buffers.

The nature of real-time stream processing applications implies the existence of performance constraints. A periodic source or sink imposes a throughput constraint and also latency constraints are common. For stream processing applications, violating these constraints typically leads to a major reduction of the quality of the applications.

To ensure that the temporal behavior of the applications meets the temporal constraints, analysis methods are required. Such analysis methods ease the processes of dimensioning, programming and optimizing the multiprocessor systems within these temporal constraints. Crucial for such methods is the existence of abstractions. By applying successive abstractions one can show that the implementation adheres to the temporal constraints. However, current abstractions have a limited accuracy and applicability and thus do not always suffice. In this thesis we will present abstractions for such multiprocessor systems in which the tasks have aperiodic schedules. These aperiodic schedules can capture the dynamic behavior of the real-time stream processing applications. We present accurate abstractions based on dataflow analysis techniques which can be used for a large number of multiprocessor systems. Compared to state of the art we broaden the scope of dataflow analysis techniques, improve its accuracy and pro-

vide a new higher level of abstraction which has powerful properties with respect to compositionality of applications.

The outline of this chapter is as follows. In Section 1.1 we will discuss the different types of multiprocessor systems in the context of real-time stream processing applications and their advantages and disadvantages. This section also motivates the focus of our system setup. Section 1.2 presents analysis techniques for the type of multiprocessor system that is considered in this thesis. It discusses the differences between existing analysis approaches and their shortcomings. The problems of current dataflow analysis techniques are formulated more precisely in Section 1.3 and Section 1.4 discusses the contributions of this thesis. Section 1.5 concludes this chapter with the outline of this thesis.

1.1 EMBEDDED MULTIPROCESSOR SYSTEMS

Different types of embedded multiprocessor systems for real-time stream processing applications exist. These multiprocessor systems each have their own properties with corresponding advantages and disadvantages. In this section we give an overview of different types of multiprocessor systems. We first distinguish periodic and aperiodic multiprocessor systems and then refine further.

1.1.1 PERIODIC MULTIPROCESSOR SYSTEMS

In periodic multiprocessor systems, each task is executed strictly periodically. The tasks are initiated by a strictly periodic clock signal. Such an initiation of tasks is usually referred to as time-triggered execution of tasks.

Such a time-triggered approach is well-established and extensive research is performed on the architectures, schedulability analysis and programming models of such periodic systems. In particular, extensive research has been conducted on the time-triggered architecture [Kop11, KB03] and the schedulability of strict-periodic tasks executed on run-time scheduled resources has also received a lot of attention [But11, SAA⁺04].

Next to the architecture and schedulability of periodic multiprocessor systems, methods exist to program time-triggered systems. Mainly, time-triggered architectures are programmed by using synchronous languages such as Lustre [HCRP91], Esterel [BS91, BG92] and Signal [GGBM91]. These languages make use of the so called synchronous hypothesis which states that each action/-task is atomic and can be seen as instantaneous. With this hypothesis, the parallel composition of synchronous programs is deterministic because no choice has to be made with respect to the interleaving of the composed programs. Another important aspect of the synchronous languages is that they can test for the absence of events while remaining functionally deterministic. This is generally not the case for data-driven approaches.

The synchronous hypothesis relies on the Worst-Case Execution Time (WCET) of a task. This is the maximum processing time that a task requires on a resource

to finish its execution. The functional behavior of synchronous programs is only defined when all tasks are finished before their next time-trigger. This is ensured by using the WCETs of tasks. Synchronous programs can thus not handle situations in which the WCET of a task is not accurate and potentially optimistic. However, it is possible that overload situations are detected locally by checking for the absence of events.

Result of the constraint that each task needs to be finished before their next trigger is that adding buffering between tasks does not help to improve the temporal behavior of the application. In data-driven approaches, adding buffering allows subsequent executions of tasks to compensate for their execution times. On the one hand this can be used to improve the temporal behavior of applications and on the other to increase the robustness to overload situations.

As a result, periodic multiprocessor systems can not exploit information about varying execution times. Furthermore, response times of tasks larger than the period of the assigned clock can only be supported by splitting up the task. And next to that, support for aperiodic (data-dependent) execution rates of tasks is not possible.

1.1.2 APERIODIC MULTIPROCESSOR SYSTEMS

Aperiodic multiprocessor systems are the systems in which the tasks do not necessarily execute strictly periodic. Run-time schedulers are used to schedule task executions. Main advantages of these aperiodic multiprocessor systems are that there is support for starting and stopping of applications, variation in execution times of tasks can be exploited and aperiodic execution rates of tasks are allowed.

We distinguish two classes of aperiodic multiprocessor systems. First, we discuss strict-periodic scheduling approaches in which tasks can execute aperiodically but are released strict periodically. Second, the data-driven approach is presented which is the most generic approach that is discussed in this thesis.

Strict-Periodic Scheduling

The first class of aperiodic multiprocessor systems which we discuss are the approaches which use strict-periodic scheduling techniques [BT13, BTV12, BS11]. Strict-periodic schedules are used to determine parameters of tasks and on run-time, tasks are periodically released which ensures that run-time schedulers can be employed to schedule the tasks with the determined parameters. Tasks can execute aperiodically but task releases are delayed such that they are not released before the next period. The applied periodic task model enables to use classical real-time schedulability analysis techniques such as [But11, SAA⁺04] to give performance guarantees.

The strict-periodic schedules are also called static-periodic schedules and are upper bounds on the self-timed schedule of an application. In such self-timed schedule, each task starts its execution as soon as it is enabled. When

strict-periodic schedules are used, executions of tasks are always delayed until their next period. Strict-periodic schedules are rate-optimal for Homogeneous Synchronous Dataflow (HSDF) models [MB07]. However, for more expressive dataflow models such as the Synchronous Dataflow (SDF) and the Cyclo-Static Dataflow (CSDF) model, the strict-periodic schedules are not optimal. In [BS11] the CSDF model is used to describe the application and determine scheduler settings, in [BTV12, BT13] the more general affine dataflow models are used.

In [BT13, BTV12, BS11] the periodic releases of tasks are used to determine the settings of a Partitioned Earliest Deadline First (PEDF) run-time scheduler. Given the strict-periodic schedules, the release moments and deadlines of tasks are computed such that the precedence constraints of the tasks are satisfied. Given these release times and deadlines, the PEDF scheduler is used to schedule the tasks on run-time and ensure that the deadlines are met.

The main advantages of these approaches is that they can rely on traditional real-time schedulability analysis theory. Furthermore, compared to periodic multiprocessor systems, the use of run-time schedulers allows the starting and stopping of applications. Next to that, because local deadlines are known, mistakes with respect to the load hypothesis (WCET of tasks) can be detected.

However, the use of the periodic task releases prevents to exploit dynamic behavior within an application. A shorter execution of a task cannot compensate for a longer previous task execution. Furthermore, aperiodic and data-dependent execution rates of tasks are not supported. Next to that, the periodic task releases in combination with strictly periodic schedules can not handle response times of tasks that are larger than the periods of the tasks.

Data-driven Scheduling

The second class of aperiodic multiprocessor systems which we distinguish are the data-driven scheduling approaches. On this class of multiprocessor systems will lie the focus in this thesis.

In data-driven multiprocessor systems, tasks are triggered by the availability of sufficient data. Run-time schedulers are often used on shared resources to schedule the tasks. Typically, the interfaces of data-driven applications are periodic.

Data-driven multiprocessor systems such as [BMP⁺04] are used in the context of stream processing applications such as software-defined radio applications. The data-driven execution allows subsequent executions of tasks to compensate for their execution times. Despite this dynamic behavior of tasks, providing guarantees on the temporal behavior of the application is still possible.

Data-driven systems form a good match with firm real-time applications. An overload situation will not immediately lead to a functional misbehave of the application. However, a drawback of this property is that local deadlines of tasks are not available or are very relaxed. Typically only global deadlines are used.

As a result, critical overload situations are detected later than in the previous discussed classes of multiprocessor systems.

Next to variation in execution times of tasks, data-driven multiprocessor systems have support for aperiodic (data-dependent) execution rates of tasks [WBS08a, WBS10]. Also response times of tasks are allowed to be larger than the period of the tasks. The main difficulty of this type of multiprocessor system is the analysis of applications. The dynamic behavior of applications prevents the use of simple periodic task models and corresponding analysis techniques. In the next section we discuss temporal analysis techniques for data-driven multiprocessor systems and their shortcomings, in more detail.

1.2 ANALYSIS TECHNIQUES FOR DATA-DRIVEN MULTIPROCESSOR SYSTEMS

The abstractions that are presented in this thesis are all related to the analysis techniques for data-driven multiprocessor systems. In this section we discuss existing analysis techniques for real-time stream processing applications and point out existing shortcomings of them.

1.2.1 RUN-TIME SCHEDULING

Figure 1.1 illustrates a generic analysis flow as used by compositional temporal analysis methods for data-driven multiprocessor systems with run-time schedulers [PWT⁺07, JPTY08, SRIE08, HGWB14a]. The analysis is based on fixed-point computation. The local analysis is often based on fixed-point computation [TBW94] and next to that, the local analysis can change how tasks interfere each other and updated interferences do change the local analysis. A sufficient condition for convergence of the flow is that the interference monotonically increases every iteration of the flow. The loops of the flow can be seen as a function. When this function is monotonic, the least fixed-point of the flow can be found by starting at an initial underestimation of the interference and then determining the new interference iteratively. This is a result of Kleene's fixed-point theorem [DP02, JPTY08, SDI⁺08].

If the function of the flow is linear or constant, iteration is not needed to determine the fixed-point of the flow. A distinction can be made between types of schedulers to exploit this analysis property. In [WBS09] three different types of schedulers are differentiated based on the information that is required to compute worst-case response times of tasks. The three classes of schedulers are illustrated in Figure 1.2. The broader the class of schedulers, the more knowledge is required to find conservative response times.

The characteristics of tasks that are distinguished in [WBS09] are as follows:

1. (worst-case) execution times of all interfering tasks
2. execution rate of all interfering tasks

Figure 1.1: Overview of the generic analysis flow of analysis methods for data-driven multiprocessor systems.

Budget Scheduling $\subset$ Starvation-Free $\subset$ Non-Starvation-Free

Figure 1.2: Classification of run-time schedulers

The execution time of a task is the required amount of processing time on the processor. The execution rate of a task is the number of task executions in a certain time interval.

The three classes of schedulers that are considered are non-starvation-free schedulers (or deterministic schedulers), starvation-free schedulers (or latency-rate schedulers [SV98]) and budget schedulers.

The budget schedulers are the smallest class of considered schedulers and requires the least characteristics of tasks to be known. They are a subclass of a-periodic schedulers [But11] for which it is possible to determine conservative response times without having to know the execution times or execution rates of the interfering tasks. Examples of these schedulers are Time-Division Multiplexing (TDM), Priority-Based Budget Scheduler (PBS) [SBW09] and the polling server [SSL89]. Neither characteristic number 1 (execution times) nor characteristic 2 (execution rates) of the interfering tasks is required to be known to compute a conservative response time of a task.

The second class of schedulers are the starvation-free or latency-rate schedulers [SV98]. This class includes for example the round-robin scheduler. The class contains the schedulers for which it is possible to compute the worst-case response time of tasks without having to know the execution rates of interfering tasks. Only the execution times of the interfering tasks have to be known (characteristic number 1). Note that the budget schedulers are a subset of the starvation-free schedulers.

Non-starvation-free or deterministic schedulers are the broadest class of schedulers that is considered in this thesis. Schedulers that belong to this class are for example Static Priority Preemptive (SPP) and Static Priority Non-Preemptive (SPNP) schedulers. When non-starvation-free schedulers are used, conservative response times of tasks can only be found when both the execution times (characteristic 1) as well as the execution rates (characteristic 2) of tasks are known. The budget schedulers as well as the starvation-free schedulers are a subset of the non-starvation-free schedulers.

For budget-schedulers as well as starvation-free schedulers it is possible to determine an upper bound on the interference that does not change in the iterations of the analysis flow from Figure 1.1. For non-starvation-free schedulers this is not possible and fixed-point iteration is needed to determine the interference and response times of tasks.

We distinguish three compositional temporal analysis techniques for real-time stream processing applications that are executed on data-driven multiprocessor systems, Modular Performance Analysis (MPA) with Real-Time Calculus (MPA-RTC) [TCN00, CKT03, WTVL06], Symbolic Timing Analysis for Systems (SymTA/S) [HHJ⁺05] and dataflow analysis techniques. This family of related analysis approaches originates from the Network Calculus [Cru91a, Cru91b, BT01] which introduced concepts to reason about for example buffer sizes and end-to-end latencies of data flowing through network connections. MPA-RTC, SymTA/S and dataflow analysis apply similar techniques to task graphs. A more detailed comparison of the different analysis methods is presented in Chapter 5.

MPA with Real-Time Calculus

MPA-RTC [TCN00, CKT03, WTVL06] is a framework for the performance analysis of embedded systems. It has its roots in the Network Calculus and is based on event arrival curves and service curves. The event arrival curves define bounds on the amounts of events that arrive in each interval of time and the service curves define bounds on the available service in each interval in time. Local analysis techniques based on convolution are used to compute the output arrival and service curves of each component. A broad range of schedulers is supported including schedulers from the non-starvation-free class. Supported schedulers are among others, SPP, SPNP, TDM and Round Robin. Fixed-point iteration is used to resolve dependencies between resource dependencies [JPTY08].

Symbolic Timing Analysis for Systems

The SymTA/S approach [HHJ⁺05] combines results on standard event models [RRE03] with existing work on response times [Leh90, TBW94]. Standard event models are used to model the traffic between components and the traditional real-time analysis techniques are used for the local analysis of components. Existing work in the domain of traditional real-time schedulability analysis can be reused and therefore SymTA/S has support for schedulers from all considered classes of run-time schedulers such as SPP, SPNP, TDM, Earliest Deadline First (EDF) and Round Robin.

The used standard event models describe the traffic of events between components with simple parameters such as period, jitter and minimum distance. The local analysis results are used to determine the output event models of each component. These standard event models are in the time interval domain and fixed-point iteration is required to resolve the dependencies between the event models [SDI⁺08].

Dataflow Analysis

Dataflow models are often used to intuitively model the temporal behavior of real-time stream processing applications executed on multiprocessor systems [LP95, SB00].

The SDF model is the best known dataflow model and is originally introduced as an untimed model [LM87]. In [SB00] the models are extended by annotating the actors with so-called firing durations. These firing durations are used to model the timing behavior of applications. Since then, more expressive dataflow models are developed for the temporal analysis of real-time stream processing applications. Examples of such more expressive dataflow models are CSDF [BELP96], Scenario-Aware Data Flow (SADF) [TGB⁺06] and Variable-Rate Dataflow (VRDF) [WBS08a].

Several analysis methods exist which use the timed extension of dataflow models to verify the temporal constraints of applications [SB00, GGS⁺06, MBBM07,

HWM⁺09, BMKD12]. Next to that, dataflow models can also be used for the optimization of applications given such temporal constraints [FKH⁺08, MBBM08, HBC11].

Temporal analysis of real-time systems of which the tasks are scheduled using run-time schedulers is also possible [WBS07b, WBS09, LMC12]. However, the dataflow analysis techniques rely on the fact that it is possible to define upper bounds on the response times that are independent of the enablings of other tasks. The scope of dataflow analysis methods is thus limited to the class of starvation-free scheduling algorithms [WBS09]. This is in contrast to the MPA-RTC and SymTA/S approaches which do have support for the broader class of non-starvation-free run-time schedulers.

One of the advantages of the dataflow analysis techniques used in this thesis, is the support of arbitrary graph topologies. The SymTA/S and MPA-RTC approaches have difficulties with cyclic data dependencies in applications whereas this is no problem for the dataflow analysis techniques considered in this thesis. Such cyclic dependencies occur also when finite size buffers with blocking write semantics are modeled. Important results in the dataflow analysis domain concerns such finite size buffers with blocking writes. Algorithms based on dataflow analysis techniques have been developed for the computation of the sizes of such buffers which ensure the liveness of a dataflow graph [GGB⁺06, BMKHB13] and which allow sufficient pipelining to meet the throughput of the application [SGB06, WBS07a, HWM⁺08, MBGS10].

Recent work introduced a temporal analysis method based on real-time calculus with arrival curves in the time domain to add correlation between events in different streams [TS09]. Dataflow models are used as the underlying model. However, the only supported dataflow model is the simplest HSDF model and difficulties arise with more expressive dataflow models as is discussed in Chapter 5. Furthermore, algorithms exploiting the cyclic data dependencies, such as buffer sizing algorithms, are not available. A more detailed comparison between the MPA-RTC, SymTA/S and dataflow analysis approaches can be found in Chapter 5.

There are several other shortcomings of dataflow analysis techniques that will be addressed in this thesis next to the limited class of supported run-time schedulers.

In data-driven systems it is possible that consecutive task executions compensate for their execution times. This can be exploited by analysis methods when information about varying execution times is known. For MPA-RTC and SymTA/S, methods are developed to take information about varying execution times into account [MKT04, QHE12]. However, current analysis methods for dataflow models assume only one WCET for tasks [SB00]. Dataflow analysis techniques therefore do not take this important benefit of data-driven systems into account.

Another shortcoming of current dataflow analysis techniques concerns the parallelism that can be analyzed. Existing analysis methods only make use of pipeline

parallelism. They determine buffer sizes which allow the required amount of pipeline parallelism such that the temporal constraints are met. The combination with data parallelism can be beneficial for both the latency as well as the throughput of an application but is not considered in dataflow analysis techniques.

The last shortcoming that is considered in this thesis concerns the scalability of temporal analysis methods for real-time stream processing applications. No intuitive compositional analysis model exists for the temporal analysis of such applications. Such compositional analysis model should support hierarchy and incremental design of the application. In general, the temporal behavior of dataflow models is not compositional and additional structure needs to be exploited to achieve powerful compositionality properties.

1.3 PROBLEM STATEMENT

Despite the frequent use of dataflow models there are a number of shortcomings which limit the scope of dataflow analysis or do not use several promising benefits of dataflow analysis methods. The problem addressed in this thesis is to find abstractions which solve a number of these shortcomings and improve the applicability of dataflow models.

Firstly, a valid timed actor theory has to be defined which can be used as the formal base of the abstractions that will be presented in this thesis. With this timed actor theory it should be made explicit under which conditions refinement is allowed.

The second challenge is to solve the problem that dataflow analysis techniques for data-driven systems are used to exploit pipeline parallelism but ignore the combination with data parallelism.

The next problem that is addressed in this thesis concerns the accuracy of existing dataflow analysis methods. Traditional dataflow analysis methods use one WCET per task. Analysis methods using only such WCETs are not able to exploit all the benefits of data-driven execution of tasks. As already discussed, subsequent task executions can compensate for their execution times in data-driven systems. However, this phenomenon is not taken into account during analysis when only the WCET execution time of a task is used.

The fourth problem of current dataflow analysis techniques is the limited support of run-time schedulers. Currently, only starvation-free schedulers are supported. The challenge is to broaden the scope of dataflow analysis techniques by defining an analysis method for non-starvation-free schedulers.

The last problem that is considered concerns the compositionality of analysis methods. The temporal behavior of dataflow models is not always preserved when actors are composed. This hampers the incremental design of real-time stream processing applications. The last challenge is to define a powerful analysis model, and corresponding analysis methods, which is compositional and has powerful mathematical properties.

1.4 CONTRIBUTIONS

This thesis improves dataflow analysis techniques and broadens the scope of dataflow analysis in a number of directions.

The contributions can be summarized as follows. The chapter in which they are discussed in detail is denoted in brackets.

1. Introduced a new timed actor theory which can be used as the formal base of abstractions for stream processing applications (Chapter 2)
2. Introduced a method to take data parallelism into account and derive the required amount of data parallelism by explicitly defining the trade-off with pipeline parallelism (Chapter 3)
3. Extended dataflow analysis to the use of workload characterizations instead of using only the WCET of tasks (Chapter 4)
4. Extended the scope of dataflow analysis to non-starvation-free schedulers (Chapter 5)
5. Developed a higher level analysis framework on top of dataflow analysis which is compositional (Chapter 6)

1.5 OUTLINE

This thesis is structured as follows. We start with a detailed discussion of the for this thesis relevant dataflow analysis techniques in Chapter 2. This chapter also presents a new timed actor theory which is used as the formal base of the abstractions that are introduced in this thesis. In Chapter 3 we present a new dataflow analysis technique to model data parallelism which enables to determine the required amount of data parallelism and make the trade-off with pipeline parallelism.

Chapter 4 introduces a dataflow analysis technique to exploit more accurate information about the execution times of tasks. In Chapter 5 an analysis method is presented which can be used to analyze real-time applications with non-starvation-free run-time schedulers. The technique presented in this chapter broadens the scope of dataflow analysis techniques to such systems. Chapter 6 then introduces a compositional temporal analysis model which can be used as a higher level abstraction of dataflow models. The conclusion and future work of this thesis is presented in Chapter 7.

Dataflow Analysis

ABSTRACT – In this chapter, we discuss dataflow models and the properties of dataflow models which are important for this thesis. This chapter also introduces a refinement relation which we use to prove temporally conservativeness of our dataflow models. This refinement relation forms the formal base behind the abstractions that are introduced in this thesis.

The goal of the abstraction that are discussed in this thesis, is to give temporal guarantees on applications implemented as task graphs that run on a multiprocessor system with run-time schedulers. The requirements on these task graphs are discussed in this chapter and also the conformance relation between task graph and corresponding dataflow model. Furthermore, we give a short overview on existing dataflow analysis techniques.

Analysis methods are used to provide guarantees on the temporal behavior of applications. Often dataflow models are used in such analysis methods to model the temporal behavior of streaming applications. The abstractions that are introduced in this thesis are also based on dataflow modeling techniques. However, to use these dataflow models, a formal base is required which supports the relation between the application and the corresponding temporal analysis model.

In this chapter we introduce such a formal base in the form of a timed actor theory. Compared to existing timed actor theories we add a coupling of the functional behavior of the different abstraction levels. By formalizing the constraints on this functional behavior, it becomes explicit what the constraints on the temporal analysis model are when this analysis model is used to model the task graph of an application. Based on this insight we present in this chapter a subset of dataflow models for which the functional behavior of an application by construction corresponds to the used dataflow model. For this subset, the dataflow model thus purely models the temporal behavior of the application.

We start this chapter by introducing some basic notation in Section 2.1. This notation is also used throughout the rest of this thesis. Then task graphs are introduced in Section 2.2. These task graphs are used in this thesis to represent applications. To model the temporal behavior of such task graphs, dataflow models are used in this thesis. These dataflow models and their relevant properties are introduced in Section 2.3. This section also presents a conformance relation between task graphs and dataflow models.

Section 2.4 presents the developed timed actor theory. This timed actor theory forms the formal base for the dataflow analysis techniques that are used in this thesis. Timed actor theory as a formal base behind dataflow modeling is discussed in Section 2.5. This section also introduces an important subset of dataflow modeling constructs for which the functional behavior is by construction orthogonal to the temporal behavior. In Section 2.6 we give an overview of the for this thesis relevant dataflow analysis techniques and we conclude this chapter with a summary in Section 2.7.

2.1 NOTATION

In this section we introduce some basic notation which we use throughout this thesis.

We use $\mathbb{N}$ for the set of natural numbers, i.e., the positive integer numbers and zero. For the complete set of integer numbers we use $\mathbb{Z}$. The real numbers are denoted by $\mathbb{R}$. We furthermore use $\mathbb{N}^+$ for the strictly positive natural numbers which excludes 0 and we write $\mathbb{R}^+$ for the strictly positive real numbers. Finally, we use $\mathbb{B}$ for the boolean domain.

Throughout this thesis we use integer arithmetic operations. We write $\lfloor x \rfloor$ for the floor function on real numbers. It gives the largest integer number not greater than x . We also use the ceil function, $\lceil x \rceil$, which returns the smallest integer number greater or equal to x . Next to that, we use the modulo operation, $x \bmod y$, which returns the remainder of the division of x by y . We define $x \bmod y$ such that it is always positive, i.e., $0 \leq x \bmod y < y$.

We denote a set that consists of the values a , b and c with $\{a, b, c\}$. We use the $\in$ operator to test membership of an element in a set. We use the notation $\{x \in D \mid P(x)\}$ to define a set consisting of the values $x \in D$ for which $P(x)$ returns true. We furthermore use $\max(D)$ for the maximum value of a set D and $\min(D)$ for the minimum value of set D . We also use the $\max$ and $\min$ operators on individual elements, $\max(a, b)$ returns the maximum of a and b . The union of sets can be obtained by using the union operator $\cup$ and a subset of a set can be specified using the $\subseteq$ operator. The $\setminus$ operator is used for set difference where $B \setminus A$ gives the items from set B which are not in A .

We also use logic operators in this thesis. We write $\neg$, $\wedge$, $\vee$, $\implies$ and $\equiv$ for logical negation, conjunction, disjunction, implication and equivalence respectively. For the universal quantifier we write $\forall$ and for existential quantification $\exists$. We

use the notation $(\forall_{x \in D} : P(x))$ in which the variables and their domains are specified before the colon and the predicate is specified after the colon. Furthermore, lines in the proofs that contain hints start with the $||$ sign.

2.2 TASK GRAPHS

The abstractions presented in this thesis are enabled by a number of requirements on the task model and the multiprocessor platform. In this thesis we use a simplified task model which is similar to the one presented in [Wig09]. Although it is outside the scope of this thesis, more flexible and advanced task models for which the presented abstractions do hold exist as well as methods which generate the corresponding dataflow models as used in this thesis [Bij11, GHB13, GHB14b].

In this section we present a short overview of the properties of the task model which we use in this thesis.

In the used task model, applications are implemented as task graphs executed on multiprocessor systems. The interfaces, i.e., the tasks that interface with the environment are required to be time-triggered. These tasks sample the environment strictly periodic. Furthermore, the used task model does support task graph topologies with cyclic dependencies.

The semantics of the task model is based on Kahn Process Networks (KPNs) [Kah74, KM77]. The used task graph is functional deterministic because it behaves as a KPN. We require that the output values of a task are completely determined by the input data.

Tasks are executed in parallel and communicate via finite size First-In First-Out (FIFO) buffers. More generic access patterns and communication patterns than is possible with FIFO buffers, can also be supported [BBJS08, BBS11].

We call the locations of a FIFO buffer, containers. The implementation and usage of the FIFO buffer enforces the synchronization between tasks. The synchronization is decoupled from the actual communication by making use of explicit synchronization statements. Tasks explicitly acquire a container from the buffer by calling an acquire primitive and release that container to the same buffer again by calling a release primitive.

Tasks always first acquire the required containers, then perform their computation and the actual reads and writes and finally release the used containers again. We distinguish data containers, which we also call full containers, and space containers which are also called empty. These two types of containers can be seen as two flows of containers, the data flow and the space flow. Although KPNs only support buffers with an infinite size, the separation of the data and space flow of containers allows us to create buffers with a finite size. Containers are moved from the space flow to the data flow and the other way around. The sum of the number of containers in the data flow plus the number of containers in the space flow is constant which thus models the finite size of the FIFO buffer. An example of a simple task is shown in Figure 2.1. It consists of two tasks, τ_0 and τ_1 , which communicate with each other via a buffer.

Figure 2.1: An example of a simple task graph with corresponding implementation of the tasks.

Tasks block on both the acquire of an empty container as well as on the acquire of a full container. A consumer task acquires data containers from the flow of data containers, uses the data and then releases the container as an empty container. This behavior is illustrated in Figure 2.1(c). A producing task first acquires an empty container, writes the corresponding data in it and then releases the filled container on the flow of data containers. An example of such a producing tasks is illustrated in Figure 2.1(b). The part of a task containing the actual reads and writes and the actual computation is non-blocking. Such non-blocking code segments are introduced in [Wig09]. A non-blocking code segment is the part of the program which happens between two acquisition statements. We call this an execution of a task. For more information about non-blocking code segments we refer to [Wig09].

Based on the definition of an execution of a task we also define the execution time of a task. We have that the execution times of the tasks are input to our analysis methods. The execution time of a task is the time that the operations take which happen between two acquisition primitives, i.e., the time required to execute a non-blocking code segment. We define the start of a task execution as the moment at which it is detected that sufficient containers are present in the FIFO buffers. The finish of a task execution is defined as the moment at which the task arrives at the next acquisition primitive. The execution time is then the time between the start and the finish of a task execution. For this execution time we assume that tasks execute in isolation on a resource and without interruptions.

Often the execution time of a task execution is not constant. Typically, only an upper bound is known which we call the WCET of a task execution. Chapter 4 will further elaborate this concept of a WCET of a task and presents an alternative for this WCET.

2.3 DATAFLOW MODELS AND THEIR PROPERTIES

The task graphs of the previous section can in general not be used for the temporal analysis of an application because they do not contain sufficient structure. In this section we introduce dataflow models as an abstraction of task graphs. These dataflow models can be used for the temporal analysis of applications.

We first introduce dataflow models in Section 2.3.1. In Section 2.3.2 we then present the important properties of these dataflow models. Section 2.3.3 finally presents the conformance relation between task graphs and dataflow models.

2.3.1 DATAFLOW MODELS

A dataflow graph consists of dataflow actors which communicate tokens over queues. A dataflow actor transfers tokens by firing [LP95]. Each firing, tokens are transferred from input queues of the actor to output queues. A set of firing rules specifies how an actor fires. Such a firing rule specifies the number of tokens that are required to be present in input queues such that the actor can fire. A firing rule also specifies the amount of tokens that are produced in outgoing queues in a firing. We say that a firing consumes tokens from input queues and produces tokens to output queues.

Traditionally, dataflow models are untimed [LM87]. In a firing, tokens are consumed and produced in an atomic action. We extend dataflow models with time as in [SB00]. The dataflow actors are annotated with a firing duration and the token consumption is separated from the token production in each firing. The atomic consumption and production of tokens is separated in one atomic consumption action and an atomic production action.

We have that an actor firing is enabled when in the input queues the amount of tokens is available that is specified by the firing rule. After this enabling, the actor can start its firing and at the beginning of the firing, the specified number of tokens are consumed from the input queues. The amount of time after which the actor firing finishes after the start of the firing is specified by the firing duration. The specified number of tokens are produced in the output queues in an atomic fashion at this finish of the firing.

Different dataflow models have been developed. The difference between these dataflow models is the firing rule. A dataflow model prescribes a firing rule with a specific structure. The less restrictive this firing rule structure is, the more expressive the dataflow model. In this section we discuss three of these dataflow models in more detail. We present them in order of increasing expressivity. These three dataflow models are also used in the remainder of this thesis. We also give a small overview of other related dataflow models and present transformations between different dataflow models.

HSDF

A Homogeneous Synchronous Dataflow (HSDF) graph $G = (V, E, \delta, \rho)$, is a directed graph that consists of a finite set of actors V and a finite set of directed queues E between those actors. Actors communicate by producing and consuming tokens over these edges. An edge $e_{ij} \in E$, directed from actor v_i to actor v_j , represents an unbounded token queue and contains initially $\delta(e_{ij})$ tokens, with $\delta : E \rightarrow \mathbb{N}$. We use δ_{ij} as a shorthand notation for $\delta(e_{ij})$. An HSDF actor is enabled when on each of its incoming queues at least one token is available. At

the start of a firing, one token is consumed from each of these incoming queues. Each firing of actor v_i finishes ρ_i after the start and at this finish, one token is produced in each outgoing queue. We have $\rho : V \rightarrow \mathcal{T}$ with $\mathcal{T}$ a time domain with an ordering $\leq$ such as $\mathbb{N}$ or $\mathbb{R}^+$.

SDF

HSDF models only support homogeneous transfer rates of actors. The Synchronous Dataflow (SDF) model [LM87] extends such HSDF models by modifying the firing rule such that a number of tokens can be specified that is required for a firing. Also the number of produced tokens can be specified. The consumption quanta do not need to be equal to the production quanta. SDF models can thus be used to model rate changes.

An SDF graph $G = (V, E, \delta, \rho, \pi, \gamma)$, is defined similarly as an HSDF graph. It consists of a finite set of actors V and a finite set of directed queues E between these actors. An SDF edge e_{ij} contains initially $\delta(e_{ij})$ tokens. Next to that, a firing finishes ρ time later than its start.

The difference with the HSDF models lies in the firing rule of an actor. Not one token is consumed/produced each firing but consumption and production quanta can be specified.

An actor v_j is enabled to fire when on each input queue e_{ij} at least $\gamma(e_{ij})$ tokens are present, with $\gamma : E \rightarrow \mathbb{N}^+$. At the start of a firing of actor v_j , $\gamma(e_{ij})$ tokens are consumed in one atomic action from all input queues e_{ij} of actor v_j . At the finish, actor v_j atomically produces $\pi(e_{jk})$ tokens on each of its output queue e_{jk} , with $\pi : E \rightarrow \mathbb{N}^+$.

CSDF

The number of tokens that is consumed/produced per firing is constant in the HSDF and SDF model. The Cyclo-Static Dataflow (CSDF) model [BELP96] extends the SDF model with the support of varying production and consumption quanta. These quanta change in a cyclo-static order. A fixed list of quanta is traversed cyclically. One complete iteration of this list is called the cyclo-static period of a CSDF actor.

A CSDF graph $G = (V, E, \delta, \rho, \pi, \gamma, \kappa)$ is defined similar to an SDF graph. The only difference is the addition of phases. The production and consumption quanta change per phase. Also the firing duration can change per phase. The phases have a static order and are repeated cyclically. The number of phases of an SDF actor v_i is specified using the function κ_i , with $\kappa : V \rightarrow \mathbb{N}^+$.

The number of tokens that is consumed by actor v_j from edge e_{ij} in phase k , $0 \leq k < \kappa_j$, is equal to $\gamma(e_{ij}, k)$ with $\gamma : E \times \mathbb{N} \rightarrow \mathbb{N}$. The number of tokens that is produced in queue e_{ij} by phase k of actor v_i is equal to $\pi(e_{ij}, k)$ with $\pi : E \times \mathbb{N} \rightarrow \mathbb{N}$. The firing duration of phase k of actor v_i is equal to $\rho_i(k)$ with $\rho : V \times \mathbb{N} \rightarrow \mathcal{T}$ where $\mathcal{T}$ is a time domain with an ordering $\leq$ such

as $\mathbb{N}$ or $\mathbb{R}^+$. The phases are repeated cyclically and thus, firing n of actor v_j consumes $\gamma(e_{ij}, n \bmod \kappa_j)$ tokens from queue e_{ij} and produces $\pi(e_{jk}, n \bmod \kappa_j)$ tokens in output queue e_{jk} . The firing duration of firing n of actor v_j is equal to $\rho_j(n \bmod \kappa_j)$.

Furthermore, we define the cumulative production and consumption in one cyclo-static period of a CSDF actor. We have $\Pi(e_{ij})$ which defines the cumulative production on edge e_{ij} during one cyclo-static period of actor v_i and $\Gamma(e_{ij})$ the cumulative consumption from edge e_{ij} during one cyclo-static period of actor v_j . We have $\Gamma(e_{ij}) = \sum_{0 \leq k < \kappa_j} \gamma(e_{ij}, k)$ and $\Pi(e_{ij}) = \sum_{0 \leq k < \kappa_i} \pi(e_{ij}, k)$.

Other Dataflow Models

More expressive dataflow models exist but are outside the scope of this thesis. These models support less restrictive firing rules than the HSDF, SDF or CSDF models. They can for example be used to model run-time variation of the production and consumption rates of tokens and/or run-time variation of the firing duration of actors. Examples of such models are the parameterized dataflow model [BB01], the VRDF model [WBS08a, WBS08b] and the variable-Rate Phased Dataflow (VPDF) model [WBS10].

Also dataflow models exist which can model scenarios and conditions, such as the SADF model [TGB⁺06, GS10] and the Boolean Dataflow (BDF) model [Buc93] respectively.

Transformations Between Dataflow Models

Transformations between different types of dataflow models exist. With these transformations, algorithms that are defined for a specific type of dataflow model can be applied to other types of dataflow models. For example, the Maximum Cycle Mean (MCM) method, as discussed in Section 2.6.2, can be used to obtain a measure for the throughput of an HSDF graph. This method can be applied to SDF and CSDF models by first transforming them into an HSDF model.

Exact transformations from SDF and CSDF models into an equivalent HSDF model exist [LM87, SB00, BELP96] and also an exact transformation from a CSDF model to an SDF model is defined [GHKS14a].

The methods which transform an SDF or a CSDF model into an equivalent HSDF use the so-called repetition factor of each SDF or CSDF actor. When the number of times that each actor fires is equal to this repetition factor, then on each edge in the dataflow model the same number of tokens is produced as is consumed. This is detailed in more detail in Section 2.3.2. Each SDF or CSDF actor is modeled using a number of HSDF actors which is equivalent to this repetition factor. The constraints between actor firings that are imposed by edges in the original SDF or CSDF actor are then transformed into edges between the corresponding HSDF actors. Algorithms which perform the transformation of SDF or CSDF models to equivalent HSDF models can be found in [LM87, SB00, BELP96].

The equivalent transformations between different type of dataflow models have one large drawback. The size, in the number of actors, of the equivalent models can be exponential in the size of the original model [JSL95]. Such an exponential blow-up of the size of the dataflow models leads to the conclusion that even algorithms with a polynomial time-complexity obtain a worst-case exponential time-complexity when they are used after these transformations.

Approximation methods are developed which do not suffer from this exponential blow-up of the number of actors in the graph [GHKB13, GHKS14b]. They are based on a linearization of the actor schedule such that smaller but conservative transformations can be defined.

2.3.2 PROPERTIES OF DATAFLOW MODELS

We present in this section some of the important properties of dataflow models. We focus on the properties which we use in this thesis. First we discuss functional determinism of dataflow models. Then the monotonicity property of functional deterministic dataflow graphs is presented, and finally, consistency of dataflow models is presented.

Functional determinism

The abstractions presented in this thesis rely on the fact that the used dataflow models are functionally deterministic. A dataflow model is functionally deterministic when all output values are only determined by input values.

A sufficient condition for a dataflow model to be functionally deterministic is presented in [LP95]. The condition is that each firing of a dataflow actor is functional and that the firing rules are sequential. Firing rules are sequential when there is a pre-defined order in which the firing rules are applied. That a firing of a dataflow actor is functional means that the firing does not have any side-effects and that the output tokens are purely a function of the tokens that are consumed from the input queues in that firing.

These sufficient conditions hold for untimed dataflow models. When time is introduced, also the order of tokens becomes important. We discuss this in more detail in Section 2.5.2.

As shown in [Wig09], functional determinism of dataflow models can also be used to prove functional determinism of task graphs. The conformance relation as presented in Section 2.3.3 links the inputs of tasks to the inputs of dataflow actors and the outputs of tasks to the outputs of dataflow actors. The locations that are released in an execution of a task are then also a function of the previously acquired locations. Furthermore, similar to dataflow actors, tasks have blocking acquire primitives. A task execution can only start when all the required locations can be acquired. This excludes so-called or-activations of tasks which are allowed by methods like [Jer05, HT07]. These or-activations result in functionally non-deterministic behavior which complicates the conformance between dataflow models and task graphs.

Monotonicity

Functionally deterministic dataflow graphs, such as SDF graphs, have a monotonic temporal behavior [WBS09]. We show this also in Section 2.5.3 by using the timed actor theory introduced in this chapter.

We say that the temporal behavior of a dataflow model is monotonic when the token production times can not become worse when a certain property of the dataflow model is increased or decreased. Temporal monotonicity of functional deterministic dataflow graphs has a number of important implications. Increasing the firing duration of one actor in the graph can never lead to an earlier enabling of any of the actors in the graph and also, decreasing the firing durations of one of the actors can never lead to any later enabling.

Similarly, earlier arrival of tokens can never lead to later enablings/finishes. Also increasing the amount of initial tokens in the graph can never lead to a later enabling of any of the actors in the graph and the opposite, decreasing the amount of initial tokens can never lead to any earlier enabling. More details on monotonicity of dataflow models can be found in Section 2.5.3.

Consistency

Another important property of dataflow models is consistency. For an inconsistent SDF model, any finite number of initial tokens will either result in deadlock or an unbounded accumulation of tokens on an edge. Algorithms exist to verify consistency of connected SDF models [Lee91].

In consistent dataflow models, the average rate at which tokens are produced on an edge is equal to the average consumption rate on that edge. Therefore, a repetition vector $\mathbf{q}$ can be determined which contains the relative firing frequencies between the actors. We use q_i for the repetition factor of actor v_i where q_i is the i th component of $\mathbf{q}$. When every actor v_i in a dataflow model fires exactly q_i times, it holds for each edge in the SDF model that the number of tokens produced on an edge is equal to the number of consumed tokens from that edge:

$$\forall_{e_{ij} \in E} : q_i \cdot \Pi(e_{ij}) = q_j \cdot \Gamma(e_{ij}) \quad (2.1)$$

Note that we use the cumulative consumption and production of a cyclo-static period of a CSDF actor. For an SDF actor we have that this cumulative consumption and cumulative production is equal to the consumption and production quanta respectively. The method to determine consistency of an SDF model is for the rest equivalent.

The repetition vector can be determined by using the topology of the dataflow model. The topology of a dataflow model can be described using a topology matrix Ψ of size $|E| \times |V|$ [Lee91, BELP96]. We use the edges on the rows of the matrix and number them with $0 \leq i < |E|$. The columns of the topology matrix

Figure 2.2: Task graph to dataflow model conformance.

correspond to the actors and are numbered with $0 \leq j < |V|$:

$$\Psi_{ij} = \begin{cases} \Pi(e_i) & \text{if } e_i = e_{jk} \text{ and } v_j \neq v_k \\ -\Gamma(e_i) & \text{if } e_i = e_{kj} \text{ and } v_j \neq v_k \\ \Pi(e_i) - \Gamma(e_i) & \text{if } e_i = e_{jj} \\ 0 & \text{otherwise} \end{cases} \quad (2.2)$$

A connected SDF model is said to be consistent if $\text{rank}(\Psi) = |V| - 1$. For consistent SDF models it is possible to find an integer repetition vector $\mathbf{q}$ of length $|V|$ for which $\Psi \mathbf{q} = \mathbf{0}$ holds.

2.3.3 TASK GRAPH, DATAFLOW MODEL CONFORMANCE

Figure 2.2 illustrates the relation between an SDF model and a task graph. This relation is similar to the relation presented in [WBS07b]. Each task τ_i in the task graph corresponds to a dataflow actor v_i in the dataflow model. Next to that, each dataflow actor has a self-edge with one token to denote that its firings may not overlap, i.e., auto-concurrency is excluded.

In this section we present the conformance relation for a task graph with a corresponding SDF model. The conformance of task graphs with other types of dataflow models is similar to the conformance relation of this section. They are all based on the fact that the number of consumed and produced tokens of a firing corresponds to the numbers of locations that are acquired and released, respectively, during the corresponding execution of the task.

As discussed in Section 2.2, a FIFO buffer can be seen as two unbounded buffers which both together form the finite size FIFO buffer. One buffers contains the flow of empty containers and the other contains the flow of the data containers.

The sum of the locations of these two buffers corresponds to the size of the FIFO buffer.

Correspondingly, such a FIFO buffer c_{ij} from task τ_i to τ_j is modeled in the dataflow model using two oppositely directed edges. The first edge, edge e_{ij} , connects actor v_i to v_j . The second edge, edge e_{ji} , is directed from v_j to v_i . The forward edge, e_{ij} , models the flow of data containers and contains initially $\delta(e_{ij})$ tokens which corresponds to the number of initially filled data locations in the FIFO buffer. Similarly, the backwards edge, e_{ji} , models the flow of empty containers and has initially $\delta(e_{ji})$ tokens which corresponds to the initially empty locations in the FIFO buffer. The total size of the FIFO buffer is then equal to $\delta(e_{ij}) + \delta(e_{ji})$.

The number of tokens that are produced and consumed on these edges is equal to the number of released and acquired containers respectively. We have that the number of locations that are filled with data by the producing task of a FIFO buffer, is equal to the number of tokens that are consumed from the edge modeling the space of the buffer and is also equal to the number of tokens that are produced on the edge modeling the flow of data containers, i.e., $\pi(e_{ij}) = \gamma(e_{ji}) = \psi(c_{ij})$. We have a similar property for the consuming task of a FIFO buffer: $\pi(e_{ji}) = \gamma(e_{ij}) = \chi(c_{ij})$. Note that we use $\psi(c_{ij})$ for the amount of locations that are produced in FIFO buffer c_{ij} by task τ_i in each execution. The number of containers that task τ_j consumes for buffer c_{ij} in each execution is equal to $\chi(c_{ij})$.

It is possible that not a fixed size but only a maximum size is specified for the FIFO buffer from task τ_i to τ_j is. The number of tokens on e_{ji} is then chosen to be equal to this maximum buffer size. As is shown in Section 2.5.3, adding more tokens reduces the constraints imposed on the dataflow model and thus using the maximum number of tokens allows the maximum scheduling freedom. Buffer sizing methods can be used to determine smaller buffer sizes and corresponding numbers of tokens that are still sufficient to meet the temporal constraints of an application. Such buffer sizing methods are illustrated in Section 2.6.2.

2.4 TIMED ACTORS

In this section we introduce the theory on which we base the abstractions that are introduced in this thesis. The concepts and the notation are inspired by [Wig09, GTW11, TLHL11].

The theory of [TLHL11] is applicable for synchronous systems whereas the approaches from [Wig09] and [GTW11] are applicable for timed actors, similar to what we use in this thesis. The refinement relation from [Wig09] is incomplete and only considers one individual execution/firing of a task/actor. No knowledge about previous executions/firings is included which means that certain abstractions which require knowledge about previous executions cannot be applied which we will show in Section 2.6.2. Also the step from actor refinement to graph refinement is incomplete. Next to that, [Wig09] suffers from the same

problem as [GTW11]; the relation with the functional behavior of the different abstraction levels is lacking.

Especially, the approach of [GTW11] sees the functional behavior of the actors as orthogonal and even introduce functional non-determinism in their refinement relation. However, the correlation of events in the different abstraction levels is not always possible when the functional behavior is non-deterministic. Nonetheless, the goal of the timed actor theories is to find conservative estimates of the temporal behavior of an application. When the events, which include a notion of value, in different abstraction levels can not be correlated exactly, then it is unclear why it is allowed to correlate the temporal behavior of these abstraction levels. Furthermore, the exclusion of non-deterministic functional behavior allows us to introduce a less complicated refinement relation.

We present in this section an alternative refinement relation which also couples the functional behavior of different abstractions levels. Based on this new refinement relation we present a subset of dataflow modeling constructs for which the functional behavior is coupled by definition and does therefore not need to be carried through the complete thesis. This simplified refinement relation also enables to use semantic models of dataflow graphs which only consider their temporal behavior. Semantic models such as max-plus algebra [BCOQ92] and constraint satisfaction formulations are only applicable when the temporal behavior does not influence the functional behavior which is the case for the presented subset of dataflow models.

The refinement relation that is introduced in this section does not allow functional non-determinism. However, an important application field of the non-determinism of the refinement relation in [GTW11] is supported by our approach by using multiple hierarchical abstractions as we will show in Section 2.6.1. By using the different levels of abstraction carefully, limited non-determinism of the temporal behavior can be supported and analyzed conservatively. For example, tasks of which the actual execution time is unknown or unpredictable can still be analyzed by using the fact that the task is functionally deterministic and that there is an upper bound on the execution time. By using the different hierarchical abstraction levels, a simple analysis model can be obtained which can be used to conservatively analyze the temporal behavior of such tasks.

The introduced timed actor theory acts as a formalism behind the dataflow analysis techniques that are used in this thesis. Note that dataflow actors are not the same as timed actors and that the connections between timed actors are not equivalent to queues/edges between dataflow actors. Section 2.5 illustrates this difference in more detail and presents the timed actor theory as a formalism behind dataflow models. In Section 2.5 we use the introduced timed actor theory to prove our key result that for a specific class of dataflow subgraphs a simpler refinement relation can be defined. For this class, the functional behavior is orthogonal to the temporal behavior and it is defined in such a way that it allows to reason in so-called external enabling events of these subgraphs.

In Section 2.4.1 we first introduce event sequences as the unit of computation of our formalization of timed actors. We then apply these event sequences in Section 2.4.2 to formalizes the timed actors and the composition of these actors.

2.4.1 EVENT SEQUENCES

We formalize the interface of actors using event sequences. In accordance with the tagged signal model of [LSV98], we define an event as a tuple of a time stamp and a value. The time stamp of event i of an event sequence x is denoted by $x(i)$. The value of event i of event sequence x is denoted as $\vartheta_x(i)$. An event sequence is ordered in increasing order of the time stamps of the different events. Time stamps are chosen in a totally ordered, continuous time domain $\mathcal{T}$ with an ordering $\leq$. We add $-\infty$ and ∞ to the time domain to denote the minimal and maximal elements respectively such that $\forall_{t \in \mathcal{T}} : -\infty < t < \infty$. We use $\mathcal{T}^\infty$ to denote $\mathcal{T} \cup \{-\infty, \infty\}$ and we have that $(\mathcal{T}^\infty, \leq)$ forms a complete partial order (CPO). The values of events are chosen in a domain $\mathcal{O}$. We furthermore have $\emptyset \in \mathcal{O}$ which is used for values that are undefined.

Each event sequences starts with event 0 and the length of event sequence a is equal to $|a|$. We define each event sequence for the complete domain $\mathbb{Z}$ by defining that $a(i) = -\infty$ and $\vartheta_a(i) = \emptyset$ for $i < 0$. Similarly, for $i \geq |a|$ we define $a(i) = \infty$ and $\vartheta_a(i) = \emptyset$.

Definition 2.1: (Event sequence)

An event sequence $x : \mathbb{Z} \rightarrow \mathcal{T}^\infty \times \mathcal{O}$ is a sequence of events. An event is a tuple of a time stamp and a value. An event sequence is weakly monotone in the time stamps of events, i.e., events in an event sequence x are ordered such that $x(i) \leq x(j)$ for $i, j \in \mathbb{Z}$ and $i \leq j$.

Definition 2.2: (Length of an event sequence)

The length of an event sequence x is defined as the number of events for which the time stamp is defined. When the time stamp of all events is defined then $|x| = \infty$:

$$|x| = \min (\{i \in \mathbb{N} \mid x(i) = \infty\} \cup \{\infty\})$$

We also introduce the notion of a subsequence. The subsequence $x\langle j \rangle$ is the sequence that consists of the events up to and including event j of an event sequence x . Such a subsequence is on itself also a valid event sequence. The properties of event sequences therefore also hold for subsequences.

Definition 2.3: (Subsequences)

The subsequence $x\langle j \rangle$ is the sequence that consists of the events of sequence x up to

and including event j :

$$x\langle j\rangle(i) = \begin{cases} x(i) & \text{if } i \leq j \\ \infty & \text{if } i > j \end{cases} \quad (2.3)$$

$$\vartheta_x\langle j\rangle(i) = \begin{cases} \vartheta_x(i) & \text{if } i \leq j \\ \emptyset & \text{if } i > j \end{cases} \quad (2.4)$$

The length of such a subsequence is then:

$$|x\langle j\rangle| = \min(|x|, j + 1) \quad (2.5)$$

We also introduce periodic event sequences which are used in Chapter 6. In such a periodic event sequence, the distance between events is constant. The time stamps of a periodic event sequence can be specified using a tuple (o, ℓ) in which o is the offset of the event sequence and ℓ is the distance between successive events.

Definition 2.4: (Periodic event sequence)

A periodic event sequence x_p of which the time stamps are specified by (o, ℓ) is an event sequence in which the distance between each event is equal to ℓ . We have for a periodic event sequence:

$$\forall_{i \in \mathbb{N}} : x_p(i) = o + i \cdot \ell$$

An event sequence refines an other event sequence when the values of the events in both event sequences are equal and when the time stamps of the events are not later. We present two equivalent definitions of event sequence refinement. The first definition uses knowledge about all the events in the event sequences and the second definition uses the structure of the event sequences to simplify the refinement relation.

Definition 2.5: (Event sequence refinement)

Event sequence x refines event sequence y , $x \sqsubseteq y$, when:

$$x \sqsubseteq y \equiv \forall_{i \in \mathbb{Z}} : x(i) \leq y(i) \wedge \vartheta_x(i) = \vartheta_y(i) \quad (2.6)$$

Equivalently we can use:

$$x \sqsubseteq y \equiv \forall_{i \in \mathbb{N}, i < \max(|x|, |y|)} : x(i) \leq y(i) \wedge \vartheta_x(i) = \vartheta_y(i) \quad (2.7)$$

Lemma 2.1 (Event sequence refinement forms a partial order). *Refinement on event sequences forms a partial order, i.e., event sequence refinement is reflexive ($x \sqsubseteq x$), antisymmetric ($x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$) and it is transitive ($x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$).*

Proof. Straightforward. $\square$

Event sequence refinement forms a CPO with a minimal element in which the time stamps of each event in a event sequence are equal to $-\infty$ and a maximal element with the time stamps of the events equal to ∞ .

Note that for subsequences $x\langle n \rangle$ and $y\langle n \rangle$ we can simplify the definition of event sequence refinement. Based on the refinement relation of Equation 2.7 and because the length of both $x\langle n \rangle$ and $y\langle n \rangle$ is maximally equal to $n + 1$, we have that:

$$x\langle n \rangle \sqsubseteq y\langle n \rangle \equiv \forall_{i \in \mathbb{N}, i \leq n} : x(i) \leq y(i) \wedge \vartheta_x(i) = \vartheta_y(i)$$

Based on this relation we have the following property which is used in a number of induction proofs which are presented in this thesis.

$$x\langle n - 1 \rangle \sqsubseteq y\langle n - 1 \rangle \wedge x(n) \leq y(n) \wedge \vartheta_x(n) = \vartheta_y(n) \implies x\langle n \rangle \sqsubseteq y\langle n \rangle$$

2.4.2 ACTORS

An actor $v = (I, O, R)$ maps event sequences on the input ports I to event sequences on the output ports O using the relation $R \subseteq (I \times \mathbb{N}) \times (O \times \mathbb{N})$. This relation specifies the dependencies between events in the event sequences on the input ports and events on the output ports. When such a dependency exists, also a distance in time between the corresponding events is specified as well as a mapping of values between the events. We use the notation $(x(i), y(j)) \in R$ for the relation between event i in event sequence x and event j in event sequence y . To simplify the notation we choose the name of an event sequence to be equal to the name of the port. We write $x \in I$ for an arbitrary event sequence on a port in I . Furthermore, when we write $x \in I$ and $x' \in I'$ in one expression, it means that x and x' are event sequences on the same port in I respectively I' . This is only possible when $I = I'$.

We use $\rightsquigarrow: (I \times \mathbb{N}) \times (O \times \mathbb{N}) \rightarrow \mathbb{B}$ to denote the existence of dependencies between event sequences. We have that $x_i \rightsquigarrow y_j$ is equivalent to $((x(i), y(j)) \in R$ and it denotes that there is an ordering between event j in the event sequence on output y and event i in the event sequence on input x . Event j on output y can only be generated after event i on input x is available. The relation between the time stamps of the corresponding events as well as the mapping of values between the events does also belong to R but is left implicit.

Event sequences are ordered in time. Event j in event sequence y does never happen later than event k in event sequence y when $k \geq j$. When event j in event sequence y depends on event i in event sequence x then also event k in event sequence y depends on the availability of event i in event sequence x . This is because of the fact that if event j in event sequence y can not be generated before event i in event sequence x is available then also event $k \geq j$ in event sequence y can not be generated. We thus have: $\forall_{k \in \mathbb{N}, k \geq j} : x_i \rightsquigarrow y_j \implies x_i \rightsquigarrow y_k$.

We require that timed actors are reproducible and continuous.

Definition 2.6: (Timed actor reproducible)

An actor $v = (I, O, R)$ is reproducible when the same output sequences are generated when equal input sequences are used: $\forall_{x_0, x_1 \in I, y_0, y_1 \in O : x_0 = x_1 \implies y_0 = y_1}$ with x_0 and x_1 corresponding to the same port in I and y_0 corresponding to the same port in O as y_1 . Furthermore, we have that y_0 are the output sequences corresponding to the input sequences x_0 and y_1 the output sequences corresponding to the input sequences x_1 .

Timed actors that are reproducible do not allow choice. The output event sequences are fixed given input event sequences. However, the actual output event sequences do not need to be known.

We use chains of refinements of event sequences on input and output ports. A chain is an infinite series of refinements $a_0 \sqsubseteq a_1 \sqsubseteq \dots \sqsubseteq a_n \dots$ with a_n a set of event sequences on the ports. We use the notation $\{a_n \mid n \in \mathbb{N}\}$ for a chain of refinements of event sequences on the set of input ports I . A chain of refinements of event sequences on the set of output ports O is denoted as $\{b_n \mid n \in \mathbb{N}\}$. Furthermore, we write $\sqcup a_n$ to denote the least upperbound of $\{a_n \mid n \in \mathbb{N}\}$.

Definition 2.7: (Timed actor continuity)

Consider chains $\{a_n \mid n \in \mathbb{N}\}$ and corresponding chains $\{b_n \mid n \in \mathbb{N}\}$ of a timed actor v . Actor v can be seen as a function that transforms a set of input event sequences a_n into a set of output event sequences b_n . We use the function $\mathcal{F}_v$ to describe this transformation for an actor v . We thus have $\mathcal{F}_v(a_n) = b_n$.

Timed actor v is continuous when for every pair of chains $\{a_n \mid n \in \mathbb{N}\}$ and $\{b_n \mid n \in \mathbb{N}\}$ we have that $\sqcup \mathcal{F}_v(a_n) = \mathcal{F}_v(\sqcup a_n)$ or alternatively $\sqcup b_n = \mathcal{F}_v(\sqcup a_n)$.

Note that a continuous timed actor is always monotone. This means that the function of the timed actor preserves the refinement order, i.e., later input event sequences will lead to later output event sequences.

Definition 2.8: (Timed actor monotonicity)

We have for a monotone timed actor $v = (I, O, R)$ the following:

$\forall_{x_0, x_1 \in I, y_0, y_1 \in O : x_0 \sqsubseteq x_1 \implies y_0 \sqsubseteq y_1}$ with x_0 and x_1 corresponding to the same port in I and y_0 corresponding to the same port in O as y_1 . Furthermore, we have that y_0 are the output sequences corresponding to the input sequences x_0 and y_1 the output sequences corresponding to the input sequences x_1 .

Timed actors refine when the input and output ports are equal, the dependencies between events are equivalent and when output event sequences are refined given that the input event sequences are also refined. Recall that event sequences refine when the time stamps are not later and when the values of the events are equivalent.

Definition 2.9: (Timed actor Refinement)

A timed actor $v = (I, O, R)$ refines actor $v' = (I', O', R')$ when the ports are equal, the dependencies between ports correspond and when the output event sequences of v refine the output sequences of v' given that the input event sequences refine each other. We use x and y for event sequences on the ports of v and x' and y' for event sequences on the ports of v' . We have that x belongs to the same port as x' and y belongs to the same port as y' . Actor v refines actor v' when:

$$\begin{aligned}
 v \sqsubseteq v' &\equiv I = I' \wedge O = O' \wedge \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : \\
 (1) \quad &x_i \rightsquigarrow y_j \implies x'_i \rightsquigarrow y'_j \wedge \\
 (2) \quad &x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsubseteq x' \langle i \rangle \implies y \langle j \rangle \sqsubseteq y' \langle j \rangle \quad (2.8)
 \end{aligned}$$

Lemma 2.2 (Timed actor refinement forms a partial order). *Refinement on timed actors forms a partial order, i.e., time actor refinement is reflexive, antisymmetric and transitive.*

Proof. We prove this by using the properties of refinement of event sequences. We assume actors $v = (I, O, R)$, $v' = (I', O', R')$ and $v'' = (I'', O'', R'')$. Because the actors form refinement relations, we have that the ports of the actors correspond. To prove that actor refinement is reflexive, antisymmetric and transitive, we only have to prove the respective properties for the relations between event sequences and that the dependencies between ports are equivalent.

Reflexive: $v \sqsubseteq v$

Property (1) of timed actor refinement holds by definition because the actors are equal.

For property (2) of timed actor refinement we have to prove:

$$\forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsubseteq x' \langle i \rangle \implies y \langle j \rangle \sqsubseteq y' \langle j \rangle$$

This follows from monotonicity of timed actors which tells us that later input event sequences will lead to later output event sequences. Given the property $x \langle i \rangle \sqsubseteq x' \langle i \rangle$ it thus also follows that $y \langle j \rangle \sqsubseteq y' \langle j \rangle$ holds.

Antisymmetric: $v \sqsubseteq v' \wedge v' \sqsubseteq v \implies v = v'$

Given $v \sqsubseteq v'$ we have that the ports of v and v' correspond. We only have to prove that the relation between input and output ports is equal.

We first prove that the dependencies of v and v' correspond:

$$\forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : x_i \rightsquigarrow y_j \iff x'_i \rightsquigarrow y'_j \quad (2.9)$$

This holds given the properties (1) of both the refinements $v \sqsubseteq v'$ and $v' \sqsubseteq v$.

Second we prove that given $v \sqsubseteq v'$ and $v' \sqsubseteq v$ we have that when the same input sequences are offered for both v and v' , also the output sequences are equal.

We have to prove:

$$v \sqsubseteq v' \wedge v' \sqsubseteq v \implies \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : x_i \rightsquigarrow y_j \wedge x \langle i \rangle = x' \langle i \rangle \implies y \langle j \rangle = y' \langle j \rangle \quad (2.10)$$

Because the ports of v and v' correspond, also the domains of the universal quantifiers in Equation 2.10 and the universal quantifiers that are used internally in $v \sqsubseteq v'$ and $v' \sqsubseteq v$ are equal. We can thus rewrite Equation 2.10 to:

$$\forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : \begin{aligned} &v \sqsubseteq v' \wedge v' \sqsubseteq v \implies (x_i \rightsquigarrow y_j \wedge x \langle i \rangle = x' \langle i \rangle \implies y \langle j \rangle = y' \langle j \rangle) \end{aligned} \quad (2.11)$$

We have that $(a \implies (b \implies c)) \equiv ((a \wedge b) \implies c)$. Applying this on Equation 2.11 learns us that we can prove actor equality by proving the right side of the right implication of Equation 2.11 when assuming:

$$\forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : v \sqsubseteq v' \wedge v' \sqsubseteq v \wedge x_i \rightsquigarrow y_j \wedge x \langle i \rangle = x' \langle i \rangle$$

To simplify the notation we do not explicitly state the refinement information of v and v' in each step but instead we carry the refinement information implicitly along. We furthermore use antisymmetry of event sequences:

$$\begin{aligned} &\forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : \\ &\quad x_i \rightsquigarrow y_j \wedge x \langle i \rangle = x' \langle i \rangle \implies x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsubseteq x' \langle i \rangle \wedge x' \langle i \rangle \sqsubseteq x \langle i \rangle \\ || \text{ From } v \sqsubseteq v' \text{ we know } &x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsubseteq x' \langle i \rangle \implies y \langle j \rangle \sqsubseteq y' \langle j \rangle : \\ &\implies y \langle j \rangle \sqsubseteq y' \langle j \rangle \wedge x' \langle i \rangle \sqsubseteq x \langle i \rangle \\ || \text{ We have } x_i \rightsquigarrow y_j \iff &x'_i \rightsquigarrow y'_j \text{ and from } v' \sqsubseteq v \text{ we know } x'_i \rightsquigarrow y'_j \wedge x' \langle i \rangle \sqsubseteq \\ &x \langle i \rangle \implies y' \langle j \rangle \sqsubseteq y \langle j \rangle : \\ &\implies y \langle j \rangle \sqsubseteq y' \langle j \rangle \wedge y' \langle j \rangle \sqsubseteq y \langle j \rangle \\ || \text{ Using antisymmetry of event sequences we have:} \\ &\implies y \langle j \rangle = y' \langle j \rangle \end{aligned}$$

This proves antisymmetry of timed actor refinement.

Transitive: $v \sqsubseteq v' \wedge v' \sqsubseteq v'' \implies v \sqsubseteq v''$

We have to prove that:

$$\begin{aligned} v \sqsubseteq v' \wedge v' \sqsubseteq v'' &\implies \forall_{x \in I, y \in O} : \forall_{x'' \in I'', y'' \in O''} : \forall_{i, j \in \mathbb{N}} : \\ (1) \quad &x_i \rightsquigarrow y_j \implies x''_i \rightsquigarrow y''_j \\ (2) \quad &x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsubseteq x'' \langle i \rangle \implies y \langle j \rangle \sqsubseteq y'' \langle j \rangle \end{aligned} \quad (2.12)$$

Property (1) follows from the properties (1) of $v \sqsubseteq v'$ and $v' \sqsubseteq v''$ and the transitivity of the logical implication.

Property (2) can be proven as follows. Similarly as for antisymmetry, the ports of v , v' and v'' correspond and thus the domains of the different universal quantifiers correspond. By using $(a \implies (b \implies c)) \equiv ((a \wedge b) \implies c)$ we can then prove:

$$\begin{aligned} \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{x'' \in I'', y'' \in O''} : \forall_{i, j \in \mathbb{N}} : \\ v \sqsubseteq v' \wedge v' \sqsubseteq v'' \wedge x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsubseteq x'' \langle i \rangle \implies y \langle j \rangle \sqsubseteq y'' \langle j \rangle \end{aligned} \quad (2.13)$$

We choose an event sequence x' which corresponds to the same port as event sequences x and x'' and is according to the refinement order between x and x'' . We choose x' such that $\forall_{i \in \mathbb{N}} : x \langle i \rangle \sqsubseteq x' \langle i \rangle \sqsubseteq x'' \langle i \rangle$. Then by using transitivity of event sequences and because the domains of the universal quantifiers match we can apply the refinements between v , v' and v'' , directly. Again, we do not explicitly note these refinements in the following derivation to simplify the notation:

$$\begin{aligned} \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{x'' \in I'', y'' \in O''} : \forall_{i, j \in \mathbb{N}} : \\ x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsubseteq x' \langle i \rangle \wedge x' \langle i \rangle \sqsubseteq x'' \langle i \rangle \end{aligned}$$

|| With $v \sqsubseteq v'$ we have:

$$\implies x'_i \rightsquigarrow y'_j \wedge y \langle j \rangle \sqsubseteq y' \langle j \rangle \wedge x' \langle i \rangle \sqsubseteq x'' \langle i \rangle$$

|| With $v' \sqsubseteq v''$ we have:

$$\implies y \langle j \rangle \sqsubseteq y' \langle j \rangle \wedge y' \langle j \rangle \sqsubseteq y'' \langle j \rangle$$

|| Using transitivity of event sequences we have:

$$\implies y \langle j \rangle \sqsubseteq y'' \langle j \rangle$$

This proves transitivity of timed actor refinement. $\square$

Timed actor refinement forms a CPO with a minimal element in which the timed actor always generates output sequences in which the time stamps of the events are equal to $-\infty$ and a maximal element in which the timed actor always generates output sequences in which the time stamps are equal to ∞ .

Composition

We now define parallel, sequential and feedback composition of actors. We then prove that actor refinement is preserved over these types of composition. This lifts actor refinement to graph refinement.

The simplest type of composition is parallel composition of actors. Two actors are composed in parallel by combining the sets of ports. These sets of ports must be disjoint. The relations between the ports are maintained and no dependencies are added between ports of the two different actors. The parallel composition of two timed actors is illustrated in Figure 2.3 and is defined as follows.

Figure 2.3: The parallel composition of two timed actors v_p and v_q into $v_p || v_q$.

Definition 2.10: (Parallel composition of timed actors)

The parallel composition of actors $v_p = (I_p, O_p, R_p)$ and $v_q = (I_q, O_q, R_q)$ with disjoint input and output ports is equivalent to:

$$v_p || v_q = (I_p \cup I_q, O_p \cup O_q, R_p \cup R_q)$$

Timed actors can also be composed sequentially. A set of connections is added from the outputs of one actor to the inputs of the other actors. The event sequences on these added connections become equal. The sequential composition of two actors is illustrated in Figure 2.4. The sequential composition of timed actors is defined as follows:

Definition 2.11: (Sequential composition of timed actors)

Let actors $v_p = (I_p, O_p, R_p)$ and $v_q = (I_q, O_q, R_q)$ be two actors with disjoint input and output ports. Actors v_p and v_q are composed sequentially by adding connections specified by $\Theta \subseteq O_p \times I_q$. The sequential composition $v_p \ominus v_q = (I_\Theta, O_\Theta, R_\Theta)$ is then defined as:

$$I_\Theta = I_p \cup \{c \in I_q \mid \neg \exists b \in O_p : (b, c) \in \Theta\}$$

$$O_\Theta = O_p \cup O_q$$

$$R_\Theta = R_p \cup R_q \cup R_{pq} \quad \text{with}$$

$$R_{pq} = \{(a(i), d(k)) \in (I_p \times \mathbb{N}) \times (O_q \times \mathbb{N}) \mid \exists b \in O_p, c \in I_q : (b, c) \in \Theta \wedge \exists j \in \mathbb{N} : (a(i), b(j)) \in R_p \wedge (c(j), d(k)) \in R_q \wedge b(j) = c(j)\}$$

The relation of $v_p \ominus v_q$ is chosen such that the event sequences on the ports of the added connections become equal. We have $\forall (b, c) \in \Theta, i \in \mathbb{N} : b(i) = c(i)$.

When a connection from an output of an actor back to an input of itself is added, feedback composition is applied. The event sequences of the connected ports of such a composition become equal. When also an internal connection was present between the ports, a cyclic dependency is created. Feedback composition is illustrated in Figure 2.5 and is defined as follows:

Figure 2.4: The sequential composition of two timed actors v_p and v_q into timed actor $v_p \ominus v_q$. A connection is added from output port b of v_p and input port c of v_q .

Figure 2.5: The feedback composition of timed actors v into timed actor $v_{(p=q)}$. A connection is added between output port q to input port p .

Definition 2.12: (Feedback composition of timed actors)

The feedback composition of an actor $v = (I, O, R)$ is the actor in which a connection is added from port $q \in O$ to port $p \in I$:

$$v_{(p=q)} = (I \setminus \{p\}, O, R_{(p=q)}) \quad \text{with} \\ R_{(p=q)} = \{(a(i), b(j)) \mid (a(i), b(j)) \in R \wedge \forall i \in \mathbb{N} : p(i) = q(i)\}$$

When feedback composition forms a cyclic dependency, then the internal connection of this cyclic dependency must be causal otherwise events must be available before they are generated which means that the actor cannot make any progress.

Definition 2.13: (Causal dependency)

A dependency from port p to port q is causal when:

$$\forall i, j \in \mathbb{N} : p_i \rightsquigarrow q_j \implies j > i$$

Next to the discussed composition operations, a hiding operator is introduced. This hiding operator can be used to remove ports from the external interface.

Definition 2.14: (Hiding)

Consider a timed actor $v = (I, O, R)$. Hiding port $q \in O$ from actor v results in actor:

$$v_{\setminus q} = (I, O \setminus \{q\}, R)$$

Note that the definition of sequential composition, feedback composition and hiding is not completely formally sound. Despite the fact that ports are removed from the external interface, internally they are still used to form the correct relations between the ports. It is possible to solve this issue by, for example, adding internal ports to the definition of a timed actor. For simplicity reasons we have chosen not to do this. However, when the internal ports are made explicit, the ideas and the proofs that are presented in this chapter would still hold.

Properties of Compositions

We now prove that parallel, sequential and feedback composition preserve actor refinement. We show that when two compositions have the same structure and the actors of the compositions refine each other, that then also the compositions refine. This lifts actor refinement to the graph level. By proving the following lemmas, we know that when a timed actor in a graph is replaced by a refined actor, then the complete graph is being refined.

Note that because a timed actor refines itself, i.e., $v \sqsubseteq v$, we do not have to distinguish which of the actors of a composition is refined. We can assume that each of the involved actors is refined.

Lemma 2.3 (Actor refinement is preserved over parallel composition). *The parallel composition of actors v_p and v_q preserves actor refinement. Given actors v'_p and v'_q with $v_p \sqsubseteq v'_p$ and $v_q \sqsubseteq v'_q$ we have:*

$$v_p || v_q \sqsubseteq v'_p || v'_q$$

Proof. Straightforward. □

Lemma 2.4 (Actor refinement is preserved over sequential composition). *The sequential composition of actors $v_p = (I_p, O_p, R_p)$ and $v_q = (I_q, O_q, R_q)$ by adding connections specified by $\ominus$, preserves actor refinement. Given actors $v'_p = (I'_p, O'_p, R'_p)$ and $v'_q = (I'_q, O'_q, R'_q)$ with $v_p \sqsubseteq v'_p$ and $v_q \sqsubseteq v'_q$ we have:*

$$v_p \ominus v_q \sqsubseteq v'_p \ominus v'_q$$

Proof. The ports of v_p and v'_p are equal as well as the ports of v_q and v'_q . The ports of $v_p \ominus v_q$ are thus also equal to $v'_p \ominus v'_q$. Given the actor refinements $v_p \sqsubseteq v'_p$ and $v_q \sqsubseteq v'_q$ we have that the sets R_p and R_q from the definition of the sequential composition $v_p \ominus v_q$ correspond with the relations R'_p and R'_q of $v'_p \ominus v'_q$. For the event sequence relations of set R_{pq} we have the following.

When $(a(i), d(k)) \in R_{pq}$ then there is a $(b, c) \in \Theta$ and $j \in \mathbb{N}$ such that $b \in O_p \wedge (a(i), b(j)) \in R_p$ and $c \in I_q \wedge (c(j), d(k)) \in R_q$. These dependencies also exist in R'_p and R'_q respectively because $v_p \sqsubseteq v'_p$ and $v_q \sqsubseteq v'_q$. We make use of the fact that $\forall_{(b,c) \in \Theta, i \in \mathbb{N}} : b(i) = c(i) \wedge b'(i) = c'(i)$. We have:

$$\begin{aligned}
 & \forall_{(a(i), d(k)) \in R_{pq}} : a_i \rightsquigarrow d_k \wedge a\langle i \rangle \sqsubseteq a'\langle i \rangle \implies a_i \rightsquigarrow b_j \wedge c_j \rightsquigarrow d_k \wedge a\langle i \rangle \sqsubseteq a'\langle i \rangle \\
 || \text{ With } v_p \sqsubseteq v'_p \text{ we have:} & \implies a'_i \rightsquigarrow b'_j \wedge c_j \rightsquigarrow d_k \wedge b\langle j \rangle \sqsubseteq b'\langle j \rangle \\
 || \text{ Using } b = c \text{ and } b' = c': & \implies a'_i \rightsquigarrow b'_j \wedge c_j \rightsquigarrow d_k \wedge c\langle j \rangle \sqsubseteq c'\langle j \rangle \\
 || \text{ With } v_q \sqsubseteq v'_q \text{ we have:} & \implies a'_i \rightsquigarrow b'_j \wedge c'_j \rightsquigarrow d'_k \wedge d\langle k \rangle \sqsubseteq d'\langle k \rangle \\
 & \implies a'_i \rightsquigarrow d'_k \wedge d\langle k \rangle \sqsubseteq d'\langle k \rangle
 \end{aligned}$$

□

Lemma 2.5 (Actor refinement is preserved over feedback composition). *Feedback composition of actor $v_{(p=q)}$ preserves actor refinement. We have actors $v = (I, O, R)$ and actor $v' = (I', O', R')$ with $v \sqsubseteq v'$. We use ports $p' \in I'$ and $q' \in O'$ which correspond to ports $p \in I$ and $q \in O$ respectively, we have:*

$$v_{(p=q)} \sqsubseteq v'_{(p'=q')}$$

Proof. The difficult part is when the feedback composition forms a cyclic dependency. When no cyclic dependency is formed, the situation is similar to sequential composition and is left out of the proof.

When the feedback composition forms a cyclic dependency then there exists an internal dependency from port p to port q and together with the added connection from port q to port p , a cyclic dependency is formed. The internal dependency from port p to port q must be causal and therefore:

$$\forall_{i,j \in \mathbb{N}} : p_i \rightsquigarrow q_j \implies j > i \wedge p'_i \rightsquigarrow q'_j \implies j > i$$

Because input ports p and p' are being connected and will be hidden, the refinement relations using port p and port p' should be still feasible. This is the case when the refinement $p \sqsubseteq p'$ always holds given that the event sequences on the remaining ports do refine. It should hold that: $\forall_{x \in I, x \neq p} : x \sqsubseteq x' \implies p \sqsubseteq p'$.

We can prove this by induction on the length of the subsequences of p and p' and by using the properties of refinement:

$$\text{Base: } \forall_{x \in I, x \neq p} : x \sqsubseteq x' \implies p\langle 0 \rangle \sqsubseteq p'\langle 0 \rangle$$

We have that the event sequence on port p is equal to the event sequence on port q . Furthermore, we know that the internal connection from port p to port q is causal which means that $q(0)$ and $q'(0)$ do not depend on $p(0)$ and $p'(0)$ respectively. The events $q(0)$ and $q'(0)$ only depend on the event sequences on

other input ports of the actors. With $v \sqsubseteq v'$ and given that $\forall_{x \in I, x \neq p} : x \sqsubseteq x'$ we have $q\langle 0 \rangle \sqsubseteq q'\langle 0 \rangle$ and therefore $p\langle 0 \rangle \sqsubseteq p'\langle 0 \rangle$.

Step: $\forall_{x \in I, x \neq p} : x \sqsubseteq x' \implies p\langle i \rangle \sqsubseteq p'\langle i \rangle \implies$

$$\forall_{x \in I, x \neq p} : x \sqsubseteq x' \implies p\langle i+1 \rangle \sqsubseteq p'\langle i+1 \rangle$$

By using $((a \implies b) \implies (a \implies c)) \equiv (a \implies (b \implies c)) \equiv ((a \wedge b) \implies c)$ we know that it is equivalent to prove: $\forall_{x \in I, x \neq p} : x \sqsubseteq x' \wedge p\langle i \rangle \sqsubseteq p'\langle i \rangle \implies p\langle i+1 \rangle \sqsubseteq p'\langle i+1 \rangle$.

Because of $v \sqsubseteq v'$ we have that $\forall_{x \in I, x \neq p} : x \sqsubseteq x' \wedge p_i \rightsquigarrow q_j \wedge p\langle i \rangle \sqsubseteq p'\langle i \rangle \implies q\langle j \rangle \sqsubseteq q'\langle j \rangle$. Given the induction hypothesis, i.e., $\forall_{x \in I, x \neq p} : x \sqsubseteq x' \wedge p\langle i \rangle \sqsubseteq p'\langle i \rangle$, we can thus conclude $q\langle j \rangle \sqsubseteq q'\langle j \rangle$ for a dependency $p_i \rightsquigarrow q_j$. Because of causality we know that $j \geq i+1$ and thus from $q\langle j \rangle \sqsubseteq q'\langle j \rangle$ follows that also $q\langle i+1 \rangle \sqsubseteq q'\langle i+1 \rangle$. We have $p = q$ and $p' = q'$ and thus $p\langle i+1 \rangle \sqsubseteq p'\langle i+1 \rangle$.

Furthermore, because timed actors are continuous also an event sequence p with an infinite length refines event sequence p' . $\square$

Best-Case Temporal Behavior

The actor refinement relation that is introduced in this section can be used to bound the temporal behavior from above. This is useful when an upper bound on the temporal behavior of an application needs to be given. For example, minimum throughput or maximum latency guarantees can be provided.

The presented actor refinement can be modified such that it can also be used to bound the best-case temporal behavior of an application from below. Such a lower bound on the best-case temporal behavior is required when this best-case behavior also influences the bounds that can be given on the temporal behavior of an application. For example the analysis method of Chapter 5 requires such a lower bound on the temporal behavior of an application. The method of Chapter 5 requires that a range can be given in which an event is produced. This range can thus be bounded from below with a best-case refinement relation that we introduce in this section.

The refinement of the best-case behavior of a task is defined similar to the actor refinement relation of Definition 2.9. The only difference is that the direction of the $\sqsubseteq$ operator on the event sequences is flipped. The intuition of a refinement of the best-case temporal behavior is that the timed actor which refines an other actor has creation times of events on the outputs which are not earlier, given that the input events are not earlier.

Definition 2.15: (Actor Refinement Best-Case Behavior)

An actor $v = (I, O, R)$ refines the best case temporal behavior of an actor $v' = (I', O', R')$ when the ports are equal, the dependencies between ports correspond and when the output event sequences of v' refine the output sequences of v given that the input event sequences refine each other. We use x and y for event sequences on the ports of v and x' and y' for event sequences on the ports of v' . We

have that x belongs to the same port as x' and y belongs to the same port as y' . Actor v refines the best-case behavior of actor v' when:

$$\begin{aligned}
 v \sqsubseteq_b v' &\equiv I = I' \wedge O = O' \wedge \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : \\
 (1) \quad &x_i \rightsquigarrow y_j \implies x'_i \rightsquigarrow y'_j \wedge \\
 (2) \quad &x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsupseteq x' \langle i \rangle \implies y \langle j \rangle \sqsupseteq y' \langle j \rangle
 \end{aligned} \tag{2.14}$$

It can be verified that the properties for timed actor refinement also hold for best-case actor refinement. Also, best-case actor refinement is preserved over the different types of compositions.

2.5 DATAFLOW MODELING

The timed actor theory introduced in Section 2.4 can act as a formalism behind dataflow modeling techniques. In this section we discuss these dataflow modeling techniques using the timed actor specification. By presenting this relation between dataflow models and the timed actor theory we enable to use the properties, derived for the timed actor theory, for dataflow modeling techniques. Which means that we for example can prove monotonicity of dataflow graphs and that we can use the fact that actor refinement can be lifted to graph refinement.

In this section we first formalize dataflow actors and dataflow edges using timed actor specifications in Section 2.5.1. Note that dataflow actors are not equivalent to timed actors and also dataflow edges are not equivalent to the connections between timed actors. In fact, we present a representation of a dataflow actor as a timed actor and also a representation of a dataflow edge as a timed actor. In Section 2.5.2 we then discuss functional determinism of timed dataflow models and in Section 2.5.3 we discuss monotonicity of dataflow graphs using functional determinism and the timed actor theory. Finally, we present the most important contribution of this chapter in the form of a subset of dataflow models for which the refinement relation can be significantly simplified in Section 2.5.4. This subset is used in the remainder of this thesis to present the introduced abstractions. By using this subset of dataflow models in combination with the timed actor theory we have that an easy local refinement property suffices to show that the abstractions are conservative.

2.5.1 DATAFLOW MODELING USING TIMED ACTORS

Actors in dataflow models map tokens from incoming edges to tokens on outgoing edges. This mapping is based on a firing rule which specifies when an actor firing can start and how the tokens are mapped from incoming edges to outgoing edges. It both specifies the number of tokens that are consumed from incoming edges and produced on outgoing edges as well as the duration that the firing takes.

This firing duration is an extension on traditional dataflow models [SB00]. In functionally deterministic dataflow models we have that the tokens produced by an actor are only determined by the tokens arriving on incoming edges. The produced tokens are independent of the arrival times of the tokens on the input edges. As shown in [SB00], time can be added to such functionally deterministic dataflow models. The firing of an actor is split in two different atomic actions. The first action consumes the tokens and the second action produces the tokens. The time between these two actions is equal to the firing duration.

Besides actors, dataflow models consist of edges which are used to connect actors. These edges can contain initial tokens. The actor which consumes from an edge can use these initial tokens such that it initially does not have to wait on the producing actor. We can define actors as well as edges using the timed actor theory of Section 2.4.2.

For a dataflow actor we have for each incoming edge an input port and for each outgoing edge an output port. Tokens in the dataflow model correspond to events in the timed actor model. The relation between the ports is defined using the firing rule. Each output port depends on each input port and the quanta of the dataflow actor determines which event of an output event sequence depends on which event on the input event sequence.

We define the enabling of an dataflow actor as the moment at which the number of tokens as specified by the firing rule are available on its incoming edges. The finish time of a firing is then this enabling time plus the firing duration as specified by the firing rule. The firing rule also specifies the number of tokens that are produced on the outgoing edges at the finish of a firing.

We use $\gamma_x(i)$ for the number of tokens that are consumed from input port x at the beginning of firing i and $\pi_y(i)$ for the number of produced tokens on output port y at the end of firing i . The firing duration of firing i is equal to $\rho(i)$. Furthermore, we define $\Gamma_x(i)$ as the cumulative token consumption on port x up to and including firing i , $\Gamma_x(i) = \sum_{k \in \mathbb{N}, k \leq i} \gamma_x(k)$, and we define $\Pi_y(i)$ as the cumulative token production on port y up to and including firing i . We have $\Pi_y(i) = \sum_{k \in \mathbb{N}, k \leq i} \pi_y(k)$.

Firing i is enabled when on each input sequence x , event $\Gamma_x(i) - 1$ is available. We have the enabling time of firing i , $e(i) = \max(\{x(\Gamma_x(i) - 1) \mid x \in I\})$ and when the actor fires self-timed, i.e., the actor starts a firing as soon as it is enabled, the finish of firing i is equal to $f(i) = e(i) + \rho(i)$. The events $\Pi_y(i) - \pi_y(i)$ up to and without $\Pi_y(i)$ are created on output y by firing i . Or equivalent, event j on output y is generated by firing $i = \min(\{k \in \mathbb{N} \mid \Pi_y(k) > j\})$. The time stamp of event j on output y is then equal to $f(i)$ with $i = \min(\{k \in \mathbb{N} \mid \Pi_y(k) > j\})$. The value mapping is left implicit.

Definition 2.16: (Dataflow actor)

A dataflow actor that fires self-timed and has a set of incoming edges I and outgoing

Figure 2.6: An example of an SDF actor for which we define a timed actor.

edges O is modeled using the timed actor $v = (I, O, R)$ with:

$$R = \{(x(i), y(j)) \in (I \times \mathbb{N}) \times (O \times \mathbb{N}) \mid$$

$$i = \Gamma_x \left(\min \left(\{k \in \mathbb{N} \mid \Pi_y(k) > j\} \right) - 1 \right)$$

with

$$y(j) = \max(\{x(i) \mid x \in I \wedge x_i \rightsquigarrow y_j\}) + \rho \left(\min \left(\{k \in \mathbb{N} \mid \Pi_y(k) > j\} \right) \right)$$

Recall that a firing of a dataflow actor is enabled when in each input event sequence the required events are available. The enabling time of a firing is thus equal to the maximum of the corresponding time stamps. The definition of a dataflow actor using timed actors uses this information about enabling times as well as the information about the finish time of a firing. Note that we assumed self-timed scheduling of the dataflow actors. When the schedule of the dataflow actor is different, we can add an additional time to the definition of time stamp $y(j)$. This additional times follows from the schedule. We then add to each time stamp $y(j)$ the time between the enabling of the corresponding firing and the start of that firing.

Example 2.1: (SDF actor)

Consider SDF actor v of Figure 2.6. It consumes each firing γ tokens from the edge denoted by x and produces each firing π tokens on the edge denoted by y . Each firing takes $\rho(i)$ time units. We assume that actor v fires self-timed which means that it fires as soon as it has γ tokens available on edge x . The start of each firing is thus equal to its enabling. Token j is produced on edge y when firing $\lfloor \frac{j}{\pi} \rfloor$ has finished which is $\rho(\lfloor \frac{j}{\pi} \rfloor)$ after its start/enabling. Firing k starts when token $(k + 1) \cdot \gamma - 1$ is available on edge x . Token j of edge y thus depends on token $(\lfloor \frac{j}{\pi} \rfloor + 1) \cdot \gamma - 1$.

The SDF actor can be presented using the following timed actor:

$$v = (\{x\}, \{y\}, R) \quad \text{with}$$

$$R = \left\{ ((x(i), y(j)) \mid j \in \mathbb{N} \wedge i = \left(\left\lfloor \frac{j}{\pi} \right\rfloor + 1 \right) \cdot \gamma - 1) \right\}$$

$$\text{with } y(j) = x(i) + \rho\left(\left\lfloor \frac{j}{\pi} \right\rfloor\right)$$

An edge in a dataflow model is also modeled as a timed actor. It is modeled using one input port and one output port:

Figure 2.7: An example of an SDF actor with a self-edge.

Definition 2.17: (Dataflow edge)

An edge in a dataflow model with δ initial tokens is modeled using the timed actor $v = (I, O, R)$ with:

$$\begin{aligned}
 I &= \{x\} \\
 O &= \{y\} \\
 R &= \{(x(i), y(i + \delta)) \mid i \in \mathbb{N}\}
 \end{aligned}$$

With R such that for the tokens $0 \leq j < \delta$ we have the time stamps $y(j) = -\infty$ and the values $\vartheta_y(j) = \emptyset$ which corresponds with the first δ tokens being available initially. For the tokens $j \geq \delta$ we have $(x(j - \delta), y(j)) \in R$, i.e., $x_{j-\delta} \rightsquigarrow y_j = \text{true}$, the time stamps $y(j) = x(j - \delta)$ and the values $\vartheta_y(j) = \vartheta_x(j - \delta)$.

The definitions of dataflow actors and dataflow edges can of course also be combined. We illustrate this with the following example in which we present the timed actor representation of a SDF actor with a self-edge. The self-edge also forms a cycle from the SDF actor back to itself which is an example of feedback composition.

Example 2.2: (SDF actor with self-edge)

Figure 2.7(a) contains a similar actor as in Figure 2.6, only one input and one output edge is added. In Figure 2.7(b) we connect output c to input b by adding an edge with one initial token. A self-edge with one initial token is thus added. This examples combines, dataflow actors, edges with initial tokens and feedback composition as defined in Section 2.4.2. The actor of Figure 2.7(b) can be obtained by first using sequential composition to add the dataflow edge that contains the initial token to input b and then using feedback composition to connect the other end of the added edge to output c .

The SDF actor that is obtained by adding the self-edge can be presented using the following timed actor. It is derived by combining Definitions 2.16 and 2.17 and

using the definitions of sequential and feedback composition of Section 2.4.2.

$$\begin{aligned}
 v_{(b=c)} &= (I, O, R) \quad \text{with} \\
 I &= \{a\} \\
 O &= \{c, d\} \\
 R &= \{((a(i), c(k)) \mid k \in \mathbb{N} \wedge i = (k+1) \cdot \gamma - 1\} \cup \\
 &\quad \{((b(j), c(k)) \mid j \in \mathbb{N} \wedge j = k\} \cup \\
 &\quad \{((a(i), d(l)) \mid l \in \mathbb{N} \wedge i = \left(\left\lfloor \frac{l}{\pi} \right\rfloor + 1\right) \cdot \gamma - 1\} \cup \\
 &\quad \{((b(j), d(l)) \mid l \in \mathbb{N} \wedge j = \left\lfloor \frac{l}{\pi} \right\rfloor\} \cup \\
 &\quad \{((c(k), b(j)) \mid k \in \mathbb{N} \wedge j = k+1\} \\
 &\text{with the time stamps such that:} \\
 &\quad \forall_{i,j,k \in \mathbb{N}, a_i \rightsquigarrow c_k \wedge b_j \rightsquigarrow c_k : c(k) = \max(a(i), b(j)) + \rho(k) \\
 &\quad \forall_{i,j,l \in \mathbb{N}, a_i \rightsquigarrow d_l \wedge b_j \rightsquigarrow d_l : d(l) = \max(a(i), b(j)) + \rho\left(\left\lfloor \frac{l}{\pi} \right\rfloor\right) \\
 &\quad \forall_{k \in \mathbb{N} : b(k+1) = c(k)
 \end{aligned}$$

Note that for firing i of actor $v_{(b=c)}$, the finish time $f(i)$ is equal to $c(i)$. The enabling time of firing i of actor $v_{(b=c)}$ is equal to $e(i) = \max(a((i+1) \cdot \gamma - 1), b(i))$. Given the self-edge we have that $b(i)$ is equal to $c(i-1) = f(i-1)$ and thus $e(i) = \max(a((i+1) \cdot \gamma - 1), f(i-1))$. We call $a((i+1) \cdot \gamma - 1)$ the external enabling of firing i of actor $v_{(b=c)}$ and $e(i)$ the actual enabling of firing i .

2.5.2 FUNCTIONAL DETERMINISM OF TIMED DATAFLOW MODELS

For untimed dataflow models we have that the dataflow model is functional deterministic when the set of firing rules is sequential and when each firing rule is functional (see Section 2.3.2). Given this functional determinism, existing methods assume that the functional behavior of an application is orthogonal to the temporal behavior [Wig09, GTW11]. However, for timed dataflow models this is not guaranteed. Tokens can overtake in an actor and the order in which tokens are produced by a dataflow actor can in some cases depend on the time at which tokens arrive at the inputs. We discuss this reordering and possible solutions in more detail in Section 2.5.4.

For functional determinism of timed dataflow models we distinguish two cases. Either the values of the tokens are not important and the order in which tokens arrive has no influence on the functional behavior, or the values are important and the values should thus be produced in the correct order, independent of the arrival times of input tokens. As we will discuss in Section 2.5.4, this can be achieved by ensuring that reordering of tokens inside a dataflow actor is excluded.

When it is ensured that the values of tokens are always correct, independent of the arrival times of tokens, then the functional behavior of the dataflow graph is independent from the temporal behavior.

2.5.3 MONOTONICITY

In [WBS09] it is shown that functional deterministic dataflow graphs have a monotonic temporal behavior. However, as discussed in the previous section, this depends on the notion of functional determinism which is in [WBS09] defined for untimed dataflow models. In this section we prove the monotonic temporal behavior of dataflow models by means of our introduced refinement relation.

We say that the temporal behavior of a dataflow model is monotonic when the token production times can not become worse when a certain property of the dataflow model is increased or decreased. We prove that the temporal behavior of a functional deterministic timed dataflow graph can never be worse when the firing durations are decreased, the schedules are improved and when the number of initial tokens are increased. By using the timed actor refinement relation as introduced in this chapter, it becomes explicit which conditions on the functional behavior should hold to be able to exploit the temporal monotonic behavior.

When the values of the tokens do not correspond and are for example re-ordered, then the relation between the events on the different abstraction levels is lost. When this relation is lost, no conclusions can be drawn from refining the time stamps. As discussed in the previous section, in functional deterministic timed dataflow graphs, the functional behavior is independent from the temporal behavior. The dataflow actors that are considered in this section ensure this functional determinism of timed dataflow graphs and are thus in such a way that either the values of the tokens are not relevant or the values of the tokens always correspond.

Because the functional behavior is then independent from the temporal behavior we only have to discuss this temporal behavior. We have that the enabling of a firing only depends on the availability of tokens on incoming connections. The time at which tokens are produced, depends on the time at which the tokens are available, the time between the enabling and the actual start of a firing (the schedule) and on the duration of the firing.

Based on the theory of timed actors, we show that the temporal behavior of a dataflow graph can only improve when the time between enabling and start of a firing of an actor is decreased, the firing duration of an actor is decreased or the number of initial tokens on an edge is increased. We show that in these cases the corresponding actor refines the original actor. As shown in Section 2.4.2, proving this actor refinement is sufficient to prove that the complete dataflow graph is refined. We show thus that the temporal behavior of the dataflow graph behaves monotonically in the schedule of an actor, in the firing duration of an actor and in the number of initial tokens on an edge.

Lemma 2.6 (Timed actor refinement for an improved schedule). *We have two actors, v and v' . The only difference between the two actors is the time between the enabling of a firing and the start of a firing. We use $u(i)$ for the time between the enabling of firing i of actor v and its actual start. The time between the start and*

enabling of firing i of actor v' is equal to $u'(i)$. We have that:

$$\forall_{i \in \mathbb{N}} : u(i) \leq u'(i) \implies v \sqsubseteq v'$$

Proof. The ports and the dependencies between ports are equivalent for v and v' . We thus only have to show:

$$\begin{aligned} \forall_{i \in \mathbb{N}} : u(i) \leq u'(i) &\implies \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{n, m \in \mathbb{N}} : \\ &x_n \rightsquigarrow y_m \wedge x \langle n \rangle \sqsubseteq x' \langle n \rangle \implies y \langle m \rangle \sqsubseteq y' \langle m \rangle \end{aligned}$$

Because the actors correspond we have that the mapping of values correspond thus we only have to show that the time stamps are refined.

Each output token of v and v' is produced in a certain firing and thus depends on input tokens of the actor. When we have $x_n \rightsquigarrow y_m$ then there is a firing i in which token n in event sequence x is consumed and token m in event sequence y is produced. We have thus that the enabling of firing i depends on the availability of token n in event sequence x . Furthermore, each firing before firing i depends on tokens $k \leq n$ in event sequence x .

Given that $x \langle n \rangle \sqsubseteq x' \langle n \rangle$ holds we know that every enabling j up to and including enabling i of actor v is not later than enabling j of actor v' . We use $e(j)$ for the enabling time of enabling j of actor v and $e'(j)$ for the enabling time of enabling j of actor v' . We thus have $\forall_{j \in \mathbb{N}, j \leq i} : e(j) \leq e'(j)$.

We furthermore have that the finish of firing j of actor v is equal to the enabling time plus the time between the start and the enabling plus the firing duration: $f(j) = e(j) + u(j) + \rho(j)$. Similarly, the finish of firing j of v' is equal to: $f'(j) = e'(j) + u'(j) + \rho'(j)$. Because $\forall_{j \in \mathbb{N}, j \leq i} : e(j) \leq e'(j) \wedge \rho(j) = \rho'(j) \wedge u(j) \leq u'(j)$ also $\forall_{j \in \mathbb{N}, j \leq i} : f(j) \leq f'(j)$. Because the tokens up to and including token m in event sequence y are produced by firings not later than firing i we have that $y \langle m \rangle \sqsubseteq y' \langle m \rangle$ holds. $\square$

Lemma 2.7 (Timed actor refinement for a decreased firing duration). *We have two actors, v and v' . The only difference between the two actors is the firing duration of the firings. We use $\rho(i)$ for the firing duration of firing i of actor v and $\rho'(i)$ for the firing duration of firing i of actor v' .*

We have that:

$$\forall_{i \in \mathbb{N}} : \rho(i) \leq \rho'(i) \implies v \sqsubseteq v'$$

Proof. The proof is almost similar to the proof of Lemma 2.6. We have to prove that:

$$\begin{aligned} \forall_{i \in \mathbb{N}} : \rho(i) \leq \rho'(i) &\implies \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{n, m \in \mathbb{N}} : \\ &x_n \rightsquigarrow y_m \wedge x \langle n \rangle \sqsubseteq x' \langle n \rangle \implies y \langle m \rangle \sqsubseteq y' \langle m \rangle \end{aligned}$$

Again we have $f(j) = e(j) + u(j) + \rho(j)$ and $f'(j) = e'(j) + u'(j) + \rho'(j)$. For this proof we have that $\forall_{i \in \mathbb{N}} : \rho(i) \leq \rho'(i)$ from which we conclude that given $\forall_{j \in \mathbb{N}, j \leq i} : e(j) \leq e'(j) \wedge u(j) = u'(j)$ it follows that also $\forall_{j \in \mathbb{N}, j \leq i} : f(j) \leq f'(j)$.

Therefore, following the same reasoning as the proof of Lemma 2.6 we have $x_n \rightsquigarrow y_m \wedge x \langle n \rangle \sqsubseteq x' \langle n \rangle \implies y \langle m \rangle \sqsubseteq y' \langle m \rangle$ and thus $v \sqsubseteq v'$. $\square$

Next to dataflow actor refinement we show that dataflow edges can refine each other. More initial tokens on an edge means that the tokens are available earlier which means that no token is produced later. However, adding initial tokens can change the functional behavior of an application because with a different amount of initial tokens, different tokens are related to firings. We show that timed actor refinement holds for edges given that the values of the tokens on the edge are chosen from a singleton alphabet such that the values are constant. An example of such an edge is an edge in which all values are undefined. When the values on an edge are not constant then adding more initial tokens can relocate the mapping between the values of events which is not allowed for timed actor refinement.

Lemma 2.8 (Timed actor refinement for an increased number of initial tokens on an edge). *We have two actors, v and v' , both corresponding to a dataflow edge. We have $v = (\{x\}, \{y\}, \{(x(i), y(i + \delta)) \mid i \in \mathbb{N}\})$ and $v' = (\{x'\}, \{y'\}, \{(x'(i), y'(i + \delta')) \mid i \in \mathbb{N}\})$ with δ the number of initial tokens on the the edge that corresponds to actor v and δ' the number of tokens on the edge that corresponds to actor v' . Furthermore, the values of the events in both actors v and v' are equal to $\emptyset$: $\forall_{i \in \mathbb{N}} : \vartheta(i) = \vartheta'(i) = \emptyset$. Then we have that:*

$$\delta \geq \delta' \implies v \sqsubseteq v'$$

Proof. The ports of v and v' are equal. We thus only have to show:

$$\begin{aligned} \delta \geq \delta' \implies & \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{n, m \in \mathbb{N}} : \\ & (1) \ x_i \rightsquigarrow y_j \implies x'_i \rightsquigarrow y'_j \wedge \\ & (2) \ x_i \rightsquigarrow y_j \wedge x \langle i \rangle \sqsubseteq x' \langle i \rangle \implies y \langle j \rangle \sqsubseteq y' \langle j \rangle \end{aligned}$$

Because the values of all the events are undefined, we only have to prove that the dependencies of the actors correspond and that the time stamps of the events refine.

We have for actor v that $x_i \rightsquigarrow y_j$ holds when $j = i + \delta$. To prove property (1) of $v \sqsubseteq v'$ we thus have to show that also $x'_i \rightsquigarrow y'_{i+\delta}$ holds for actor v' .

For actor v' we have that $x'_i \rightsquigarrow y'_j$ holds when $j = i + \delta'$. As defined in Section 2.4.1, event sequences are ordered in time. We have that event k is not earlier than event j when $k \geq j$. As defined in Section 2.4.2, we say that events depend on each other when there is an ordering between them. We have $x'_i \rightsquigarrow y'_j$ when event j in event sequence y' can only be generated after event i in event

sequence x' is available. Because of the ordering in time of event sequences we thus have that when event j in an event sequence y' depends on an event i in event sequence x' , then also event k in event sequence y' depends on event i in event sequence x given that $k \geq j$. For $k \geq j$ we have $x'_i \rightsquigarrow y'_j \implies x'_i \rightsquigarrow y'_k$ and thus because $\delta \geq \delta'$ we have $x'_i \rightsquigarrow y'_{i+\delta'} \implies x'_i \rightsquigarrow y'_{i+\delta}$ and thus $x'_i \rightsquigarrow y'_{i+\delta}$ holds for actor v' which proves property (1) of $v \sqsubseteq v'$.

To prove property (2) of $v \sqsubseteq v'$, we have to show that the time stamps of the events of y refine the time stamps of y' given that the input event subsequences refine. For actor v we have the dependency $x_i \rightsquigarrow y_{i+\delta}$ and we thus have to prove $\forall i \in \mathbb{N} : x\langle i \rangle \sqsubseteq x'\langle i \rangle \implies y\langle i + \delta \rangle \sqsubseteq y'\langle i + \delta \rangle$.

For $i \geq \delta$ we have $y(i) = x(i - \delta)$ and $y'(i) = x'(i - \delta')$. Because the event sequences are ordered in time we have that $x'(i - \delta') \geq x'(i - \delta)$ when $\delta \geq \delta'$ and thus $y'(i) \geq x'(i - \delta)$. Therefore, using $y(i + \delta) = x(i)$ and using the result $y'(i + \delta) \geq x'(i)$ we have: $\forall i \in \mathbb{N} : x(i) \leq x'(i) \implies y(i + \delta) \leq y'(i + \delta)$.

For $i < \delta$ it holds that $y(i) = -\infty$ and thus for $i < \delta$ we have $y(i) \leq y'(i)$. Combining this result for $i < \delta$ with the result of the previous paragraph for $i \geq \delta$ gives us: $\forall i \in \mathbb{N} : x\langle i \rangle \sqsubseteq x'\langle i \rangle \implies y\langle i + \delta \rangle \sqsubseteq y'\langle i + \delta \rangle$ and thus $v \sqsubseteq v'$. $\square$

2.5.4 SUBSET WITH SIMPLER REFINEMENT

In this section we define a subset of dataflow modeling constructs for which the refinement relation can be significantly simplified. We present constraints to which the dataflow actors should adhere. With this subset of dataflow modeling constructs we exclude reordering of tokens and make the relation between different abstraction levels easier. We first start with discussing properties of dataflow actors regarding reordering of tokens and then present the simplified refinement relation based on a specific subset of dataflow modeling constructs which among others excludes that tokens internally can overtake each other.

The presented dataflow subset is used as the formal base of the abstractions presented in this thesis. Also several existing abstractions using dataflow models adhere to the constraints of this subset. The in this section presented theory can thus also act as a formal base for existing abstractions such as [WBS07b, WBS09, SBW09, SB09, LMC12].

For untimed functional deterministic dataflow graphs we have that the enabling of a firing only depends on the arrival of tokens on the incoming edges. Section 2.5.2 presents an additional condition for functional determinism of timed dataflow graphs. The functional behavior of an actor is independent from the temporal behavior when either the values of tokens are not important or the order in which these values are produced is independent from the arrival times of tokens.

We have that actors consume and produce tokens in a FIFO manner when tokens do not overtake each other inside the actor. This means that between any two firings of the same actor, the first firing that starts is also the first firing that produces output tokens. When the firing duration is not constant, tokens

might overtake each other in the actor [BCOQ92, PBB⁺03]. The tokens of firing i can be produced earlier than the tokens corresponding to firing $i - 1$ when the firing duration of firing i is shorter than the firing duration of firing $i - 1$. This complicates the relation between the event sequences on the inputs and the event sequences of the outputs. Also functional determinism of timed dataflow graphs is not guaranteed. The order in which values are produced in the external queues can depend on the arrival time of tokens. Next to that, typically tasks do not have such a reordering behavior from input to output because a task execution needs to be finished before the next execution can start.

A subset of dataflow modeling constructs can exclude such reordering of tokens [BCOQ92, PBB⁺03]. When actors have a constant firing duration, tokens can not overtake each other inside an actor. Next to that, when the enabling of firing i is always later than the finish of firing $i - 1$ then naturally overtaking of tokens is also excluded. That the enabling of a firing is not earlier than the previous finish can be achieved with a cyclic dependency from an actor back to itself. The most simple cyclic dependency that ensures this ordering of firings is a self-edge, i.e., an edge back to the same actor, with one initial token and constant consumption and production quanta equal to one. The subset of dataflow actors that is used in this section is such that actors do have a constant firing duration or actors do have a self-edge with one initial token.

Excluding reordering of tokens ensures functional determinism of timed dataflow graphs. Furthermore, it simplifies the refinement relation of timed actors but only in combination with a specific structure of the dataflow graph. We present the simplified refinement relation for groups of actors with this specific structure. We call these groups, actor components which in fact are subgraphs with a specific structure. Next to excluding reordering of tokens and the use of a specific graph layout of an actor component, we use the property of homogeneity of connections.

Definition 2.18: (Homogeneous connection)

A connection between actors is homogeneous when in firing i of the producing actor, the same number of tokens is produced as is consumed in firing i of the consuming actor.

For homogeneous connections we use the property that on a homogeneous connection without initial tokens, the firings of the consuming and producing actor correspond. The number of produced tokens in a firing is equal to the number of consumed tokens in the same firing of the consuming actor. Next to that, without initial tokens we have that firing 0 of the consuming actor is enabled by firing 0 of the producing actor and thus we also have that firing i of the consuming actor is enabled by firing i of the producing actor.

Definition 2.19: (Actor component)

An actor component is a set of dataflow actors in which only one actor consumes

Figure 2.8: Two examples of actor components.

tokens from the outside. We call this actor the *input actor*. Also only one actor produces tokens to the outside of the actor component. This actor is called the *output actor*. These two actors are allowed to be the same actor.

Actor components need to be fully connected and be externally enabled. This means in this context that there is a directed path from the input actor to each of the actors, including the output actor, without initial tokens. This means that the enabling of each actor depends on the enabling of the input actor. Furthermore, the internal connections of an actor component are required to be homogeneous and tokens are not allowed to overtake each other.

Figure 2.8 shows two examples of such actor components. The left actor component consists of CSDF actors and the right actor component consists of SDF actors. Note that the internal connections are homogeneous. Also we require that the firing durations of the input actors of both actor components, are constant. This ensures that tokens are transferred in order. The other actors are connected via cyclic dependencies which ensures that a firing can only start after the previous firing is finished. This means that reordering of tokens is also excluded.

The input actor is the only actor with external incoming connections and the output actor is the only actor with external outgoing connections. We use the external enabling of an actor component as well as the finish of an actor component which are both based on these properties of the input and output actors.

Definition 2.20: (External enabling of an actor component)

An actor component is externally enabled when the input actor can consume sufficient tokens from the external connections.

Definition 2.21: (Finish of an actor component)

The finish of an actor component is equal to the finish of the corresponding firing of the output actor.

To sum up the properties of an actor component we have, (1) one input actor and one output actor, (2) no reordering of tokens, (3) homogeneous internal

connections and (4) a path without tokens from the input actor to each other actor.

Because tokens do not overtake each other (2) and because internal connections are homogeneous (3) we have that for internal actors a produced token i is related to a consumed token i of that actor. Furthermore, there is a path without initial tokens from the input actor (1) to each other actor in the actor component (4). Therefore, each firing of the actors is linked to each other and none of the internal actors is enabled initially, i.e., firing i of an internal actor is not enabled before firing i of the input actor is finished. Because also each internally produced token i is related to a consumed token i from the same actor we have that the enabling of firing i of an internal actor depends exactly on firing i of the input actor. Also for the output actor we have that firing i depends on firing i of the input actor. Therefore, we have that the finish i of an actor component relates to the external enabling i of the actor component. We can thus reason about actor components by using the external enabling of the actor component and the corresponding finish. We can abstract from the actual consumptions and productions of external tokens. These external consumption and production patterns can be checked in isolation.

Based on the relation between the external enabling and the finish of an actor component we can define a simplified refinement relation. We have that when the external enabling patterns correspond and the production patterns are equal than the refinement of actor components can be shown by using only the time stamps of the external enablings and the finishes of the actor component, i.e., the external enabling sequence and the finish sequence. The external enabling patterns correspond when the consumption quanta on the incoming connections from the outside are equivalent and the production patterns correspond when the production quanta on outgoing external connections correspond.

Lemma 2.9 (Enabling-based refinement of actor components). *We have two actor components for which the external enabling conditions of the input actors of two actor components correspond and the production rules of the output actors of two actor components correspond. Furthermore, the actor components are such that the value mapping from tokens on external incoming queues to tokens on external outgoing queues correspond. For these actor components the refinement relation can be simplified as follows.*

We use e and e' for the external enabling sequences of actor components v and v' respectively and f and f' for the finish sequence of v and v' respectively. We have that actor v refines v' when:

$$\forall_{i \in \mathbb{N}} : e\langle i \rangle \sqsubseteq e'\langle i \rangle \implies f\langle i \rangle \sqsubseteq f'\langle i \rangle \quad (2.15)$$

Proof. This can be proven by showing that when Equation 2.15 holds, it implies

that $v \sqsubseteq v'$ holds:

$$\begin{aligned}
 \forall_{i \in \mathbb{N}} : e\langle i \rangle \sqsubseteq e'\langle i \rangle &\implies f\langle i \rangle \sqsubseteq f'\langle i \rangle \implies \\
 \forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : & \\
 (1) \ x_i \rightsquigarrow y_j &\implies x'_i \rightsquigarrow y'_j \wedge \\
 (2) \ x_i \rightsquigarrow y_j \wedge x\langle i \rangle \sqsubseteq x'\langle i \rangle &\implies y\langle j \rangle \sqsubseteq y'\langle j \rangle \quad (2.16)
 \end{aligned}$$

For an external enabling k we have for each $x \in I$ of the input actor a token i which enables firing k . Because the enabling conditions of v and v' match we also have that token i of $x' \in I'$ of the input actor of v' enables firing k . Then there exist a dependency $x_i \rightsquigarrow y_j$ in actor component v which specifies that token j is produced at output port $y \in O$ of the output actor in firing k . Because v' has the same enabling and production rules it follows that then also $x'_i \rightsquigarrow y'_j$ is true and token j is produced at output port $y' \in O'$ in firing k . This proves property (1) of Equation 2.16.

Furthermore, tokens cannot overtake each other internally and thus token j on output port y of actor component v corresponds to token j on output port y' of actor component v' . The value mapping is equivalent for both actor components and thus the value of token j on port y is equivalent to the value of token j on port y' .

Given for each input event sequence a token i which enables firing k we have that given $\forall_{x \in I, y \in O} : \forall_{x' \in I', y' \in O'} : \forall_{i, j \in \mathbb{N}} : x_i \rightsquigarrow y_j \wedge x\langle i \rangle \sqsubseteq x'\langle i \rangle$ it implies that $e(k) \leq e'(k)$. Because $x\langle i \rangle \sqsubseteq x'\langle i \rangle$ we have for every enabling n up to enabling and including enabling k : $e(n) \leq e'(n)$. Therefore, $e\langle k \rangle \sqsubseteq e'\langle k \rangle$ and given Equation 2.15: $f\langle k \rangle \sqsubseteq f'\langle k \rangle$.

Token j , produced at output y , is produced at the end of firing k of actor component v and token j , produced at output y' , is also produced at the moment firing k finishes and thus because of $f\langle k \rangle \sqsubseteq f'\langle k \rangle$ we have $y(j) \leq y'(j)$. Because we have $f\langle k \rangle \sqsubseteq f'\langle k \rangle$ we know that the finish times of firings before k are also refined and thus for every token m up to and including token j we have $y(m) \leq y'(m)$ and thus $y\langle j \rangle \sqsubseteq y'\langle j \rangle$ which proves property (2) of Equation 2.16. $\square$

Similar to Definition 2.15, we can also provide an enabling-based actor refinement for the best-case behavior of dataflow actors. We have that actor v' bounds the best-case temporal behavior of v from below when:

$$\forall_{i \in \mathbb{N}} : e\langle i \rangle \sqsupseteq e'\langle i \rangle \implies f\langle i \rangle \sqsupseteq f'\langle i \rangle \quad (2.17)$$

Next to actor refinement we have refinement of dataflow edges. We distinguish two types of dataflow edges, edges on which the data does has changing values and edges on which the values of data is constant. A typical example of the second type of dataflow edges are edges that are used to model the flow of empty locations in a buffer. The values on these edges are always undefined and thus constant. An example of an edge in which the value of the tokens does change is

an edge that models the flow of full locations of a buffer. The tokens on such an edge correspond to a location in the buffer and contain the actual written data which can of course be different for each location.

As shown in Lemma 2.8 we have for edges with constant values for the data that an edge refines another edge when the number of initial tokens on the edge is higher. For the example of the edge modeling space locations of a buffer this corresponds to having more available space in a buffer. For dataflow edges on which the values of data do change we only have refinement when the number of tokens is exactly equal.

2.5.5 TASK GRAPH, DATAFLOW MODEL SUBSET CONFORMANCE

We can also use the in Section 2.5.4 introduced subset of dataflow models to define a temporally conservative abstraction of task graphs. This section introduces such a conformance relation. The introduced subset enforces functional determinism of timed dataflow graphs and therefore has the property that its functional behavior is independent of the temporal behavior. We can thus discuss the conformance of the functional behavior independent of the conformance of the temporal behavior.

The conformance relation is similar to the relation presented in 2.3.3. The only difference is that a task does correspond to an actor component instead of one dataflow actor. We discuss this relation between a task and an actor component in more detail in this section. We first show that the dependencies and the functional behavior of a task can be modeled correctly using an actor component. Then we present the refinement relation between a task and an actor component. This refinement relation is used in other chapters of this thesis to prove the conservativeness of the presented abstractions.

Given the conformance relation of Section 2.3.3 we have that the consumption and production quanta of a firing of a dataflow actor corresponds to the amount of consumed and produced containers in a task execution. Given the properties of actor components we thus have that the dependencies of the actor component correspond to the dependencies of the task.

The dataflow model is purely used to model the temporal behavior of an application. To prove that the functional behavior of the dataflow model corresponds to the task graph, we only have to show that for each actor component a mapping between the values of input events to the values of output events exists, which corresponds to the functional behavior of the corresponding task.

To define this mapping of the values of events we distinguish two cases. The first case is when the actor component corresponding to the task consists of only one actor, then this actor maps the values from input queues to output queues in the same way as the task and then of course the value mapping corresponds.

When the actor component consists of more than one actor we use the following mapping of the functional behavior. Firing i of the input actor of an actor

component consumes the tokens that correspond to the locations that are acquired in execution i of the task. This input actor then aggregates these tokens into one single token and communicates this aggregated token to the internal actors connected to the input actor. Internal edges of the actor component are homogeneous thus exactly one token per firing is transferred from each input queue to each output queue. The internal actors, except the output actor, thus transfer the token from the input actor to the next actor. This corresponds to a functional behavior that only copies the aggregated token from their input queues to their output queues. The token that finally arrives at the output actor in firing i is then equal to this aggregated token that thus consists of the tokens consumed from the external connections. The output actor receives thus in firing i , the tokens that correspond to the locations that are acquired by execution i of the task. The output actor then finally does the actual mapping from these input tokens to the output tokens in the same way as the corresponding task does. And thus the mapping from input tokens to output tokens of the actor component corresponds to the mapping of acquired locations to released locations of the corresponding task.

We then have that the dependencies match and the value mapping of the actor component and the task corresponds. What remains to show is that the temporal behavior of the task refines the temporal behavior of the actor component. This can be shown using the refinement relation for actor components as presented in Lemma 2.9.

We use e for the external enabling of the task and e' for the external enabling time of the actor component. The finish time of the task is denoted by f and we use f' for the finish time of the actor component. We have that the task refines the actor component when:

$$\forall_{i \in \mathbb{N}} : e\langle i \rangle \sqsubseteq e'\langle i \rangle \implies f\langle i \rangle \sqsubseteq f'\langle i \rangle \quad (2.18)$$

Furthermore, we have that the actor component bounds the best-case temporal behavior of the task from below when:

$$\forall_{i \in \mathbb{N}} : e\langle i \rangle \sqsupseteq e'\langle i \rangle \implies f\langle i \rangle \sqsupseteq f'\langle i \rangle \quad (2.19)$$

Note that the enabling-based refinement relation of Equation 2.18 is different from the refinement relation of [WBS09]. The refinement relation in [WBS09] is in fact too weak to prove the refinement relation between tasks and certain actor components. The refinement relation of [WBS09] is as follows:

$$\forall_{i \in \mathbb{N}} : e(i) \leq e'(i) \implies f(i) \leq f'(i) \quad (2.20)$$

The difference with the enabling-based refinement relation introduced in this chapter is that the refinement relation of Equation 2.20 does not contain any knowledge about executions/firings before execution/firing i when something is proven about execution/firing i . For example, $f(i) \leq f'(i)$ can not be proven

Figure 2.9: An illustration of the hierarchical analysis of an application.

with this refinement relation when $f(i)$ and $f'(i)$ depend on $f(i-1)$ and $f'(i-1)$ respectively.

However, as we will also illustrate in Section 2.6.2, for certain abstractions this information about previous executions/firings is required. With the refinement relation of Equation 2.20, refinement can thus not be proven for such actor components. The refinement relation of Equation 2.18 does contain this information about previous executions/firings and can be used to formally prove the refinement relation of tasks with these actor components.

2.6 DATAFLOW ANALYSIS

This section presents applications of dataflow models for the temporal analysis of applications. We first use the properties derived in this chapter to define an hierarchical analysis approach in Section 2.6.1. This hierarchical analysis approach can also be used to analyze applications of which the actual temporal behavior is not known. In Section 2.6.2 we discuss the for this thesis relevant dataflow analysis techniques.

2.6.1 HIERARCHICAL ANALYSIS

Timed actor refinement is transitive as is shown in Lemma 2.2. Abstractions can thus be stacked on top of each other and a hierarchical analysis of applications is thus supported.

This hierarchy also allows to model and analyze applications of which the temporal behavior is not constant or even non-deterministic, similar to [GTW11]. This is illustrated in Figure 2.9. The goal is to prove that the implementation of an application adheres to the temporal constraints which can be proven by means of the hierarchical chain of abstractions as illustrated in Figure 2.9.

On the right hand side of this chain we have the most abstract level which contains the temporal constraints of the application. On the left we have the most refined level of abstraction. This is the implementation in the form of a task graph. To illustrate the idea of the hierarchical analysis, only one task and corresponding actor component is discussed. The actor component complies with the conformance relation between task graphs and data models as presented in Section 2.5.5.

The illustrated task, τ , has a varying, potentially non-deterministic, temporal behavior. The time execution i of task τ takes is equal to $ET(i)$. Each execution of task τ can take a different amount of time.

First, we add a level of abstraction in which the exact functional and temporal behavior of the task is modeled with a dataflow model. This is obviously only possible in theory because the actual temporal behavior of the task is unknown. However, adding this level of abstractions enables to define further abstractions which can then be based on dataflow modeling techniques.

In this first abstraction level we have that task τ is modeled using actor component v which consists of one dataflow actor, v_0 . This actor has a self-edge with one token and is thus a valid actor component. Actor v_0 models the exact execution time of task τ . We have that $\forall_{i \in \mathbb{N}} : \rho_0(i) = ET(i)$. However, because the actual value of $ET(i)$ is not known, also $\rho_0(i)$ is unknown which means that actor component v can not be analyzed using temporal analysis methods.

The next abstraction level, actor component v' , solves this issue. As an example, two dataflow actors are used in this actor component. We will show in Section 2.6.2 that such an actor component with two actors is used to model the effects on the temporal behavior of a task when it is scheduled using a run-time scheduler that uses budgets. Also in Chapter 4 such an actor component is used to model knowledge about variation in the execution times of a task.

The two actors of actor component v' model together the temporal behavior of actor component v . Following the refinement relation of Equation 2.18, the only constraint we have is that the time between the external enabling and finish of actor component v' needs to be at least equal to the time between the corresponding external enabling and finish of actor component v . An upper bound on the temporal behavior of the task τ or actor component v can thus be used to conservatively model the temporal behavior. This is illustrated in more detail in Section 2.6.2 and Section 3.4.4. The conservative actor component can then be used to verify that the temporal constraints are met.

2.6.2 ANALYSIS TECHNIQUES

Dataflow models are often used for analysis and optimization of the temporal behavior of applications. In this section we give a short overview of the for this thesis relevant dataflow analysis techniques.

Throughput Analysis

Throughput analysis of dataflow models is used to verify the throughput constraint of an application. Such throughput analysis techniques exist for different types of dataflow models.

A measure for the throughput of a self-timed executed HSDF graph can be defined using the so-called MCM of the HSDF graph [Rei68, SB00]. The throughput is equal to $\frac{1}{\mu(G)}$ with $\mu(G)$ the MCM of HSDF graph G . The MCM of an HSDF

graph is equal to the average time between two firings of each actor in the HSDF graph [Rei68, SB00].

The MCM of an HSDF graph is very similar to the Maximum Cycle Ratio (MCR) of weighted graphs. The MCM of an HSDF graph can therefore also be determined by transforming the HSDF graph in such a weighted graph. Depending on the chosen structure of the weighted graph, an MCM or a MCR algorithm [Das04] can be applied on the weighted graph to determine the MCM of the HSDF graph, see [BCOQ92, GGS⁺06].

The MCM of an HSDF graph is equal to the maximum of the cycle means of the cycles in graph G . A cycle mean of a cycle is defined as the sum of firing durations of the actors on the cycle divided by the sum of the numbers of tokens on the edges in the cycle. Only simple cycles have to be considered because the cycle mean of other cycles can not be higher. A simple cycle is a cycle in which each actor is traversed only once.

We define the MCM of an HSDF graph G as follows:

$$\mu(G) = \max_{c \in C(G)} \frac{\sum_{v_i \in V(c)} \rho_i}{\sum_{e \in E(c)} \delta(e)}$$

We use $C(G)$ for the set of simple cycles of G . Furthermore, for a cycle c we use $V(c)$ for the actors on the cycle and $E(c)$ is equal to the set of edges on this cycle c .

Also throughput analysis methods for SDF and CSDF models exist. As discussed in Section 2.3.1, the SDF and CSDF models can be transformed into an equivalent HSDF model. On this HSDF model, the MCM algorithm can be applied to determine the throughput of the SDF or CSDF model. However, the size of this HSDF model is worst-case exponential in the size of the SDF or CSDF model. The run-time of the approach thus grows worst-case exponentially in the size of the graph.

State-space exploration can be used to determine the throughput directly from the SDF model [GGS⁺06]. Self-timed simulation of the dataflow graph is used to compute the complete state-space. This prevents the transformation to an HSDF model. The run-time of the algorithm is on average better, nevertheless, the worst-case time-complexity is non-polynomial.

An alternative to simulating an SDF graph is to determine its throughput by making use of max-plus algebra [GGS⁺06, GKBS12]. The max-plus algebra is used to analyze the self-timed execution of the SDF graph.

By using periodic schedules of the dataflow actors, the time-complexity of determining the throughput of an application can be reduced. Recent work introduces such an approximation method by using k -periodic schedules of the dataflow actors [BMKD12]. When k is equal to 1, strictly periodic schedules are used and the time-complexity algorithm is polynomial. Increasing k improves the accuracy of the throughput estimate but also increases the time-complexity.

Buffer Sizing

The throughput of an application depends on the sizes of the buffers between tasks. Larger buffers allow more pipeline parallelism which can result in a higher throughput of the application. As shown in the previous section, dataflow models can be used to determine the throughput given buffer sizes. Next to that, we show in this section that dataflow models can also be used to determine the required buffer sizes given a throughput.

As discussed in Section 2.3.3, a buffer between two tasks is modeled using two oppositely directed edges in the dataflow model. One edge models the flow of locations in the buffer to which data is written and the other edge models the flow of empty locations. The sum of these locations is constant and is equal to the size of the buffer. The number of initial tokens on the edges correspond to the initially available locations in the buffer. The sum of these initial tokens is thus also equal to the size of the buffer.

When the size of a buffer is not fixed, typically the amount of space locations can be freely chosen up to the maximum size of the buffer. The amount of space locations can be chosen freely because the values of these locations are not important, see Lemma 2.8. On the other hand, the values of the data locations are relevant and the amount of the data locations can thus not be chosen freely.

The space locations are modeled with the tokens on the edge that corresponds to the flow of empty locations of the buffer. The values of the events on this edge have a value equal to $\emptyset$. Given Lemma 2.8, we know that adding more tokens on an edge can never lead to worse temporal behavior of the dataflow model. On the other hand, the temporal behavior can be worse when the number of tokens is decreased.

Given these properties, it follows that the dataflow model corresponding to a task graph can be used to determine the required buffer sizes. We can use temporal analysis methods for dataflow models to determine the amount of initial tokens which allow to meet the throughput constraint. We give a short overview of existing buffer sizing algorithms which use dataflow modeling techniques.

An important result is the buffer sizing method for HSDF models of [NG93]. It provides an algorithm with a polynomial time-complexity to determine the minimum sum of buffer sizes such that the throughput constraint of the application is met. However, the algorithm is shown to be non-optimal [MBGS10]. In fact it is shown in [MBGS10] that this buffer sizing problem for HSDF models is an NP-complete problem. Furthermore, a revisited buffer sizing algorithm is presented to determine the minimum sum of buffer sizes given a throughput constraint.

In [GBS05] a model checking approach is used to determine the minimum sum of buffer sizes which ensure deadlock freedom of an SDF model. Deadlock freedom means in this context that the buffer sizes allow each actor to fire infinitely often.

For SDF as well as CSDF models also methods exist which determine the minimum sum of buffer sizes which guarantee a certain throughput of the dataflow

Figure 2.10: The actor component corresponding to a task that is scheduled using a budget scheduler.

model. State-space exploration is used to solve the trade-off between the minimum sum of buffer sizes and the throughput of the dataflow model [SGB06, SGB08].

The above mentioned methods suffer from a worst-case non-polynomial time-complexity. Approximation methods with a worst-case polynomial time-complexity are developed to determine sufficient but conservative buffer sizes which ensure that a throughput constraint can be met. These methods are based on linearizations of the actor schedules and are developed for different types of dataflow models such as the SDF, the CSDF and the VRDF model [WBS07a, WBS08a, GHKB13].

Modeling Budget Based Run-Time Arbitration

Dataflow models can also be used to conservatively analyze the temporal behavior of tasks that are scheduled using a run-time scheduler. Existing dataflow analysis methods cover the so-called starvation-free schedulers. In Chapter 5 we extend this to a larger class of schedulers.

An important subclass of the starvation-free schedulers are the budget schedulers. A run-time budget scheduler guarantees every task τ_i a minimum time budget S_i in every time interval of length Q_i . This interval is called the replenishment interval. The temporal behavior of tasks that are scheduled using a budget scheduler can be accurately modeled and analyzed using dataflow models [BHM⁺05, WBS07b, WBS09, LMC12].

These dataflow models are a typical example of the conformance relation between tasks and actor components. These actor components consist of two [WBS07b, WBS09] or even more dataflow actors [LMC12] to accurately analyze the temporal behavior of tasks. This illustrates immediately the need for the conformance and refinement relation between tasks and actor components.

As an example we consider the dataflow model of [WBS09]. The used actor component in this work consists of two actors as is illustrated in Figure 2.10. We call such an actor component a latency-rate actor component because the rate is decoupled from the latency. The maximum rate is determined by the actor with the self-edge whereas the latency of the actor component is determined by the two

actors together. It is shown in [WBS09] that the actor component of Figure 2.10 models the temporal behavior of the corresponding task conservatively when $\forall i \in \mathbb{N} : \rho_i \geq ET(i)$ with $ET(i)$ the execution time of execution i of the task.

It can indeed be proven that given that the external enablings are refined then also the finish time of a firing of the actor component is never earlier than the corresponding finish time of the execution of the task. To prove this for a certain finish, knowledge about the previous finish is required as will for example be illustrated in the proof of Lemma 4.1 in Chapter 4. However, as is also discussed in Section 2.5.5, the refinement relation that is used in [WBS09] does not include this information. Temporal conservativeness of the actor component can thus not be proven with the refinement relation of [WBS09]. The refinement relation which is introduced in this chapter does include this information about previous finishes and can be used to prove the conservativeness of the actor component. Such a proof is very similar to the proof of Lemma 4.1.

2.7 SUMMARY

In this chapter, a timed actor theory is presented. This timed actor theory is used as a formal base for dataflow models and we use this timed actor theory as a base for the abstractions that are presented in this thesis. For the timed actor theory it is proven that actor refinement can be lifted to graph refinement.

The presented timed actor theory distinguishes itself from other timed actor theories on the field of functional determinism. In our refinement relation, the functional behavior is taken into account whereas in alternative timed actor theories this functional behavior is seen as orthogonal to the temporal behavior and thus not taken into account.

We have identified difficulties which arise when the functional behavior is not taken into account. For example, when this functional behavior is not considered, it is unknown when and how events of different abstraction levels should be correlated. Another example regards the sizes of buffers in the application. By adding the functional behavior to the refinement relation it becomes explicit that increasing a buffer in the task graph is typically only allowed when initially empty locations are added to the buffer and not when initially full locations are added.

We have also presented a subset of dataflow models for which the presented refinement relation can be simplified. For this subset, the functional behavior is orthogonal to the temporal behavior. It is defined in such a way that it allows to reason in the external enabling events of actor components. The functional behavior of different abstractions levels corresponds when the numbers of consumed and produced tokens/locations correspond in each firing/execution. The presented subset is the key result of this chapter and adds a formal base to several existing abstractions that use timed dataflow models to model the temporal behavior of applications. This subset is furthermore used in the abstractions that are defined in this thesis.

Next to the introduced timed actor theory, we have presented in this chapter the for this thesis important properties of dataflow models. We furthermore presented a number of existing analysis techniques in which dataflow models are applied in the field of temporal analysis of streaming applications.

Unified Dataflow Model for Data and Pipeline Parallelism

ABSTRACT – This chapter presents an analysis method which can determine the required amount of data parallelism by describing data parallelism in a dataflow model without replicating dataflow actors. Current dataflow analysis methods only exploit pipeline parallelism and neglect data parallelism or determine it in separation. By describing data parallelism in a dataflow model without replication of dataflow actors allows to make a trade-off between the amount of data and pipeline parallelism that is required to meet the temporal constraints of the application. It is also shown how large the buffers need to be such that the determined amount of data and pipeline parallelism required for the satisfaction of the throughput constraint, can be realized. Furthermore, it is shown that the use of a new type of circular buffer enables the proposed dataflow modeling.

Streaming applications exploit the inherent pipeline parallelism of the applications to increase their achieved throughput. However, exploiting only pipeline parallelism is not always sufficient to meet the temporal constraints of applications. Exploiting coarse-grained data parallelism is often required as well and can also increase the throughput significantly [GTA06]. Data parallelism uses the fact that stateless tasks can be replicated such that different data samples can be processed in parallel by these replicated tasks. Replicating tasks can result in smaller buffers than is possible when only pipeline parallelism is used to meet the throughput requirement. On the other hand, pipeline parallelism often requires

The material in this chapter is based on work presented in [HGWB14b].

less additional processing resources. Efficient implementations of applications therefore trade off data parallelism with pipeline parallelism to achieve their throughput requirements.

As is shown in the previous chapter, the temporal behavior of streaming applications is often analyzed using a dataflow model. The actors of the dataflow model, model the tasks of the application. Traditionally, task replication is modeled by duplicating actors in split-join constructs [GTA06, KM08]. However, such split-join constructs require explicit replication of the actors which means that the graph structure of the dataflow model needs to be modified when the amount of task replication is changed. Such a modification of the graph structure is difficult to parameterize, which hinders the analysis of the trade-off between data and pipeline parallelism.

In this chapter, we present an analysis method based on SDF models [LM87] in which data parallelism is modeled simultaneously with the pipeline parallelism of the application. Data parallelism is modeled in the SDF model without replicating actors which allows us to make the trade-off between data and pipeline parallelism.

Dataflow analysis methods exist which can determine minimal buffer sizes by using the trade-off between the amount of pipeline parallelism in different buffers [SGB06, WBS08b, MBGS10, GHKB13]. These methods model the size of buffers as initial tokens which are parameterized in the dataflow model. In this chapter we extend these approaches and also use initial tokens that are parameterized to model the amount of replication of a task. Because buffers as well as data parallelism are modeled with initial tokens that are parameterized, existing buffer sizing techniques can be used to determine the required amount of data and pipeline parallelism simultaneously. Furthermore, the analysis methods are applicable for cyclic task graphs.

Previous dataflow analysis methods use self-edges to limit the auto-concurrency of actors. These self-edges contain one initial token to obtain the property that firings of the actors do not overlap. As shown in the previous chapter, in current dataflow analysis methods, an actor component is used to model one task. Typically at least one of the actors in this actor component has a self-edge with one initial token to model the property of tasks that consecutive executions of a task may not overlap.

Contrary to other dataflow analysis methods we use in this chapter the number of self-edge tokens to model the replication of the corresponding task. We show that this model can be accurate and that it is conservative. To show this, we introduce a new expansion method from an SDF graph to an equivalent HSDF graph. For the proposed dataflow model it is required that the amount of synchronization operations is independent of the amount of replication of the task. This is enabled by the applied circular buffers by which tasks communicate. The used circular buffer has support for multiple producers and consumers. Implementing the communication with the replicated tasks by separate FIFO buffers

is in general not possible for our approach because it does not correspond with the proposed dataflow modeling.

The outline of the chapter is as follows. Related work is discussed in Section 3.1. The basic idea of the analysis method is presented in Section 3.2. One of the enablers of the proposed analysis method is the use of a specific type of circular buffer. The properties and the implementation of this circular buffer are discussed in Section 3.3. Section 3.4 presents the modeling of data parallelism by means of an SDF model and contains proofs regarding the accuracy and conservativeness of the modeling. In Section 3.5 the combination of pipeline parallelism and data parallelism is described in more detail and the trade-off between data and pipeline parallelism is discussed. The approach is evaluated in a case-study in Section 3.6 and we conclude this chapter in Section 3.7.

3.1 RELATED WORK

Data parallelism is often used to increase the amount of parallelism of applications. For example polyhedral models are used in [SKD02] to exploit the available data parallelism in loops with a finite number of iterations. However, the temporal behavior of the applications is not modeled and also the combination with pipeline parallelism is not considered. In [MNS10] the throughput of applications is determined for cyclic polyhedral models with data parallelism. However, the combination with pipeline parallelism is not considered.

It is shown in the context of the StreamIt language [TKA02] that the combination of pipeline parallelism and coarse-grained data parallelism is important [GTA06]. Other methods acknowledge the benefits of the combination of data and pipeline parallelism [SYBT13, BL13] and use dataflow models to analyze the temporal behavior of data parallel tasks. However, none of these methods determines the amount of data parallelism that is required to meet the throughput constraint of an application.

In [KM08] a method is presented that also does compute the required amount of data parallelism. It uses an Integer Linear Programming (ILP) formulation to determine which of the predefined set of replications ensures that the temporal constraints can be met. However, just as the other methods mentioned above, it only considers acyclic task graphs. Next to that, it does not determine the required amount of data parallelism and pipeline parallelism simultaneously. The above mentioned papers model data parallelism in a dataflow model by means of duplicating the dataflow actor corresponding to a task. Next to that, they use explicit split and join tasks to communicate with the data parallel tasks. Such split-join constructs are difficult to parameterize because the graph topology depends on the amount of data parallelism. This hinders the trade-off between data and pipeline parallelism. In [KM08], pipeline parallelism is determined separately from data parallelism, which means that the trade-off between data and pipeline parallelism is not considered.

We model data parallelism by using only one dataflow actor corresponding to a task and allow this actor to occur multiple times in parallel in the schedule. This is modeled using a self-edge with multiple tokens which is parameterized in the dataflow model. This allows to use existing buffer sizing algorithms to determine the required amount of data parallelism simultaneously with the required amount of pipeline parallelism. Furthermore, the analysis method of this chapter is applicable for cyclic task graphs and also explicitly makes the trade-off between pipeline parallelism and data parallelism. In our approach communication between tasks is done via buffers with support for multiple readers and writers [DBC⁺07, BBS11]. Therefore, we do not need split and join tasks.

Buffer sizes as well as data parallelism are modeled with initial tokens that are parameterized in the dataflow model. Buffer sizing methods exist which determine the minimal amount of initial tokens that are required to meet a throughput constraint. From our analysis follows that such buffer sizing methods can also be used to determine the required amount of data parallelism simultaneous with the minimal buffer sizes. As we will show in Section 3.5, a trade-off exists between increasing the size of buffers and increasing the amount of data parallelism. This trade-off can be resolved by using cost functions for initial tokens in combination with existing buffer sizing methods. Buffer sizing methods which give exact solutions exist [SGB06, MBGS10]. The main drawback of these methods is that they have a non-polynomial time-complexity. In fact, finding the minimal amount of buffering that is required to meet a throughput constraint is shown to be an NP-complete problem [MBGS10]. Buffer sizing algorithms with a polynomial time-complexity are also developed [WBJ06, WBS08b, GHKB13]. They are based on linearizations of dataflow schedules and find conservative (not exact) buffer sizes which guarantee the throughput constraint. Such linearizations are also used in this thesis in the context of a compositional temporal analysis method which is presented in Chapter 6. We can also use approximation algorithms to determine the required amount of data parallelism. However, due to the linearization of schedules, they do not accurately capture the trade-off between pipeline and data parallelism, which is detailed in Section 3.5.

3.2 BASIC IDEA

This section illustrates the basic idea behind the proposed unified analysis method for data and pipeline parallelism.

Traditionally, self-edges with one token are used to limit the auto-concurrency of dataflow actors. Such self-edges prevent overlap of firings of actors which corresponds with the fact that the current execution of a task needs to be finished before the next execution can start. We propose to use more than one token on such a self-edge to model data parallelism. On the model side this looks straightforward. The maximum number of overlapping firings of the dataflow actor is limited by the number of self-edge tokens. However, two important issues arise. First, we need to prove that these overlapping firings indeed correspond

with the executions of the data parallel tasks. And second, we have to show that the temporal behavior is modeled conservatively.

For the second problem it is important that the amount of synchronization operations that the task performs during an execution is independent of the amount of data parallelism. Furthermore, we have to show that buffers are modeled conservatively. In the proposed modeling, the space of a buffer is shared between the data parallel task replications. This is not possible when a separate buffer is used for each replication. We therefore use a circular buffer with support for multiple producers and multiple consumers [DBC⁺07, BBS11]. In Section 3.3 we show that also separate buffers for each replication can be used as long as the total buffer size is a multiple of the amount of replication.

Figure 3.1 illustrates the idea of our dataflow modeling approach and the problems that arise when dataflow actors are replicated to model data parallelism. Figure 3.1(a) shows a task graph of two tasks, τ_i and τ_j , connected by a buffer with δ_b locations. As shown in Chapter 2, the temporal behavior of this task graph can be conservatively expressed using the dataflow model shown in Figure 3.1(b). The buffer is modeled with two opposite directed edges and the two tasks are modeled with two actors. Dataflow actor v_i , corresponding to τ_i , cannot be replicated because τ_i has state. This is modeled with a self-edge containing one token which is equal to how self-edges were previously used to limit the auto-concurrency of dataflow actors. The self-edge of actor v_j , corresponding to τ_j , illustrates the new modeling technique and can contain more than one token to denote the replication of task τ_j . Both the parameterized number of tokens modeling buffer size and data parallelism can be determined simultaneously by existing buffer sizing methods such that the required throughput is guaranteed.

In Figure 3.1(c), the buffer size δ_b is chosen to be equal to three locations and task τ_j is replicated twice into tasks $\tau_{j,0}$ and $\tau_{j,1}$. Figure 3.1(d) presents the corresponding dataflow model. As can be seen in Figure 3.1(c), tasks τ_i , $\tau_{j,0}$ and $\tau_{j,1}$ share the same buffer. This is implemented by using a circular buffer with support for multiple consumers and multiple producers. Task τ_i writes data to the buffer in a consecutive order. Furthermore, each location is read by exactly one consuming task. The locations that are read by a task are fixed. In this example, execution n of task $\tau_{j,0}$ uses the data of location $(2 \cdot n) \bmod 3$ in the buffer and execution n of task $\tau_{j,1}$ uses the data from location $(2 \cdot n + 1) \bmod 3$. Task τ_i can reuse a location when both tasks $\tau_{j,0}$ and $\tau_{j,1}$ no longer need that location.

Figure 3.1(e) shows the expanded dataflow model which is used to show the correctness of the modeling of data parallelism. This expanded dataflow model is only used to show the relation between the task graph and the dataflow model and is not used to determine the right values for the parameters. The expanded dataflow model is equivalent to the dataflow model of Figure 3.1(d). As can be seen, actor v_i is split into two actors, $v_{i,0}$ and $v_{i,1}$, which are tightly coupled by a cycle with one token. This means that these two actors fire consecutively. On the other hand, actor v_j is split into actors $v_{j,0}$ and $v_{j,1}$ which both have their own self-edge. This means that they can fire independently of each other, which

Figure 3.1: Illustration of the basic idea of the proposed modeling of data parallelism using dataflow models.

corresponds to a data parallel execution of the corresponding tasks. Actor $v_{j,0}$ models task $\tau_{j,0}$ of Figure 3.1(c) and actor $v_{j,1}$ models task $\tau_{j,1}$. The firings of both actors $v_{i,0}$ and $v_{i,1}$ correspond to executions of task τ_i .

The edges from actors $v_{i,0}$ and $v_{i,1}$ to actors $v_{j,0}$ and $v_{j,1}$ and from actors $v_{j,0}$ and $v_{j,1}$ to actors $v_{i,0}$ and $v_{i,1}$ model the buffer between the tasks. The topology of these edges depends on the size of the buffer. As can be seen in Figure 3.1(e), for a buffer of size three, they form the cycle $((v_{i,0}, v_{j,0}), (v_{j,0}, v_{i,1}), (v_{i,1}, v_{j,1}), (v_{j,1}, v_{i,0}))$ through all four actors which illustrates that the corresponding tasks reuse each others buffer locations.

When data parallelism is directly modeled using replication of the corresponding dataflow actors, one would directly end up with the expanded dataflow model of Figure 3.1(e). However, in such a model it would not be possible to model a parameterized size of a buffer because the topology of the edges cannot be changed dynamically. Therefore, with such an approach it would not be possible to make a trade-off between data and pipeline parallelism.

3.3 IMPLEMENTATION

The previous section illustrated how applications with data parallel tasks can be analyzed using dataflow models. In this section we present how such applications can be implemented. Specifically, we present an implementation of the buffers between tasks for which the synchronization costs are not dependent on the amount of data parallelism that is used.

Often, tasks with data parallel behavior are implemented by adding split and join tasks and by using a separate buffer for each replication of a task [GTA06, KM08]. This approach has a number of drawbacks. First of all, because each replication has a separate buffer, the space of the buffers cannot be used efficiently. Secondly, the split and join tasks need to be mapped to the platform which means that less resources remain for the other tasks. Furthermore, the join of the data stream is typically implemented as an implicit barrier [Ope07]. The join task waits until each of the data parallel task replications is finished before it continues the execution. Also the split task sometimes acts as an implicit barrier. The split task splits a data block into smaller sub blocks which are processed by the different task replications. When the splitting is implemented in such a way that the complete data block needs to be available before the block is split then the split task also acts as an implicit barrier. Such additional synchronization points increase the required amount of buffering and also increase the end-to-end latency of the application compared to our solution.

This is illustrated in Figure 3.2. This example consists of four tasks of which tasks b^0 and b^1 are data parallel replications of a task b . Task c can only start when the results of both tasks b^0 and b^1 is available which is illustrated in the schedule beneath the task graph. Task c produces its first item 50 time units after the first start of task a . The buffer sizes that are required to ensure that task a can execute every 10 time units are equal to four locations for both the buffer

Figure 3.2: An example schedule of a split join construct.

Figure 3.3: An example schedule of a data parallel task that communicates via a shared buffer.

between task a and the *split* task as well as the buffer between the *join* task and task c . When the *split* did not act as a barrier, the buffer between task a and the *split* can be reduced to three locations. However, the latency would not improve.

When the communication with a data parallel task is implemented with a buffer with support for multiple producers and multiple consumers, all the tasks can use buffer space that is at that moment not used by other tasks. The same buffer space can thus be used by different tasks. Therefore, the required size of

such a shared buffer is often smaller than the sum of the sizes of the separate buffers. Also the latency improves compared to the split-join solution. This is illustrated in Figure 3.3. The two buffers are now shared between tasks a , b^0 and b^1 and between tasks b^0 , b^1 and c . To ensure that task a can execute every 10 time units, it suffices to let both buffers consist of three locations. Also the latency is reduced compared to the solution of Figure 3.2. Task c now produces its first item 40 time units after the first start of task a which is 10 time units earlier than in the schedule of Figure 3.2.

The type of buffer we use in this chapter is a circular buffer that uses overlapping windows [BBS11]. Each producer and each consumer has its own window in which it is allowed to write and read respectively. The buffer also allows for multiple producer windows and multiple consumer windows at once. Synchronization is decoupled from the actual read/write operations. Consumer tasks can immediately acquire a data location for reading as soon as the producer has written it. Consumer tasks thus synchronize by checking if the required location is already written. Furthermore, each consumer task moves the tail of their window such that locations that are not required anymore fall out of their window. Producer tasks synchronize by acquiring new space locations by moving the head of their write window and by signaling that a location is written. Because the buffer is circular, the movement of the windows is implemented in a cyclic manner. When the window reaches the end of the buffer, it wraps around and continues again at the beginning of the buffer. The administration of these wraparounds can be done by storing the wrap count for each task.

The head of a write window is never allowed to overtake the tail of any of the read windows because that would mean that data can be overwritten before it is read. Overtaking happens when the head of the write window moves to a further location in the buffer than the tail of a read window. Initially, each write window can acquire an amount of space equal to the buffer size. The head pointer of a write window thus starts one wrap around earlier than the read windows and overtakes the tail of a read window when it is more than buffer size locations ahead.

We modify this type of buffer in such a way that it fits the data parallel implementation we propose. We make use of the fact that consumer tasks can skip locations, i.e., remove them from their read window without have to acquire them first. Consuming tasks thus never block on data that is not required by that task. The order in which the different replications of a task read/write locations in the buffer is fixed. The replications follow a round robin schedule. The first replication always reads/writes the first locations in the buffer and then skips the locations that are read/written by the other replications. The second replication always skips the locations that are read by the first replication, reads/writes the second set of locations in the buffer and then skips locations again. The other replications behave in the same manner. We therefore know that each location in the buffer is written by exactly one replication of a task and is also read by exactly one of the replications of a task. To acquire a location for reading, a consuming

task thus needs to verify the availability of the location by checking the write flag of exactly one task. To acquire a location for writing, only one of the reading tasks, i.e., the one that actually reads the location, needs to be checked if that replication does not need the particular location anymore.

The order in which replications execute is fixed and we can thus derive which task writes/reads a location of the buffer in a certain wrapping of that buffer. Consider a buffer that consists of δ_b locations and a producing task which is replicated δ_{dp} times. The amount of locations that is written per execution of the task is equal to π . Each replication is executed in a fixed order and we thus have that the i th block of π locations is written by replication $(i \bmod \delta_{dp})$. Location k of wrapping N of a buffer belongs to block $\left\lfloor \frac{k+N \cdot \delta_b}{\pi} \right\rfloor$ and is thus written by the following replication of the producing task: $\left(\left\lfloor \frac{k+N \cdot \delta_b}{\pi} \right\rfloor \bmod \delta_{dp} \right)$. Similarly, location k of wrapping N of a buffer is read by the following replication of the consuming task: $\left(\left\lfloor \frac{k+N \cdot \delta_b}{\gamma} \right\rfloor \bmod \delta_{dc} \right)$ with δ_b the size of the buffer, γ the amount of locations that are read per execution of the task and δ_{dc} the amount of replication of the consuming task.

We introduce an initial offset of the read/write windows to correspond with this fixed order of reads and writes. Each replication of a task τ_d is identified by a unique number n , $0 \leq n < \delta_d$, with δ_d the amount of replication of the task. Each of the replications of τ_d initially skip $n \cdot \omega$ locations where ω is the number of read/written locations per execution of τ_d from/to a certain buffer by a consuming or producing task respectively. Each replication then tries to acquire the next ω locations in its window to read/write them. The availability of each of the ω locations is checked individually. As shown in the previous paragraph, it can be calculated which replication has read/written the location that the task wants to acquire which means that only that replication needs to be checked to find out if the location is available.

When the locations are available, the execution of the task can continue. During the execution, the ω locations are released by moving the tail pointer corresponding to the replication. This tail pointer is moved by an additional $(\delta_d - 1) \cdot \omega$ at the end of the execution of the task. This last movement skips the locations that are read/written by the other replications. In total, the tail pointer is moved by $\delta_d \cdot \omega$ locations per execution of a replication. The implementation of the buffer is such that the movement of the window is done in only one operation. The time it takes to move the window with n locations does not depend on how large n is. Therefore, the amount of synchronization operations that have to be performed to skip the locations does also not depend on the amount of replication of the task. Together with the fact that the availability of a location can be found out by checking exactly one task we thus have that the amount of operations that are performed to synchronize with the buffer does not depend on the amount of data parallelism of the tasks.

We illustrate the global idea of this implementation of the buffers with Example 3.1.

Figure 3.4: The state of the buffer of Figure 3.1(c) at different points in time. $\hat{w}$ is the head of the write window of task τ_i and $\check{r}_0$ and $\check{r}_1$ are the tails of the read windows of tasks $\tau_{j,0}$ and $\tau_{j,1}$ respectively. Locations with a green background are empty and locations with a blue background are full.

Example 3.1:

Consider the task graph of Figure 3.1(c). The circular buffer in this task graph contains three locations and has one producing and two consuming tasks. The initial state of this buffer is illustrated in Figure 3.4(a). The head pointer of task τ_i is denoted by $\hat{w}$. The tail pointers of tasks $\tau_{j,0}$ and $\tau_{j,1}$ are denoted by $\check{r}_0$ and $\check{r}_1$ respectively. The three cells in the circle correspond with the locations in the circular buffer. Initially, all locations are empty which is denoted with the green background of the cells. The head pointer of τ_i and the tail pointer of $\tau_{j,0}$ point to location 0. Task $\tau_{j,1}$ has initially skipped one location and its tail pointer therefore points to location 1.

Figure 3.4(b) shows the state of the buffer after one execution of task τ_i . Task τ_i has acquired location 0 of the buffer which is denoted by the movement of the head pointer $\hat{w}$. Furthermore, task τ_i has written the location 0 of the buffer which is illustrated by the blue background of the first cell in Figure 3.4(b).

The state of the buffer after one execution of both tasks τ_i and $\tau_{j,0}$ and no execution of $\tau_{j,1}$ is shown in Figure 3.4(c). Location 0 of the buffer which was written by task τ_i is read by task $\tau_{j,0}$. After $\tau_{j,0}$ finishes its first execution, it moves its tail pointer with two locations to location 2. Location 0 is therefore not needed anymore by

the consuming tasks and can thus eventually be reused by task τ_i . This is again illustrated in the figure by the green background of the cell.

Figure 3.4(d) illustrates the buffer state after six executions of τ_i , three executions of $\tau_{j,0}$ and one execution of $\tau_{j,1}$. In this situation the buffer is completely full with data though location 1 (denoted by 4) is already read by task $\tau_{j,0}$ which is illustrated by the green background. The other full locations are illustrated by the blue background of the cells. Task τ_i cannot make progress in this situation. Despite the fact that location 1 is empty, it first needs to acquire and write location 0 which is not yet available. Next to that, task $\tau_{j,0}$ has already used all of the data that belongs to it and can thus also not make progress. The only task that is enabled is task $\tau_{j,1}$ which can execute twice without having to wait for other tasks in this situation.

In addition to illustrating how the introduced buffer works, Example 3.1 also illustrates two important properties of the proposed implementation. In Figure 3.4(c), task $\tau_{j,0}$ has executed even when task $\tau_{j,1}$ is not yet enabled. Thus our approach does not have a blocking split behavior. Next to that, in Figure 3.4(d) task $\tau_{j,0}$ has already executed three times while task $\tau_{j,1}$ only executed once. This shows that task $\tau_{j,0}$ can progress before task $\tau_{j,1}$ has finished the same execution. Thus the proposed implementation does also not have blocking join behavior.

In the example it is also shown that the buffer locations are reused by the different tasks. The third execution of task $\tau_{j,0}$ reads from location 1 in the buffer just like the first execution of $\tau_{j,1}$ does. Such a reuse of buffer location cannot be achieved when separate FIFO buffers are used to communicate with the replicated tasks. However, there is a situation in which separate FIFO buffers can be used. When the total buffer size is equal to an integer multiple of both the producer replication factor as well as the consumer replication factor, the total buffer size can be split fair into separate FIFO buffers. Such a situation can always be created by increasing the buffer size to the smallest integer multiple of the least common multiple of the replication factors that is larger than the computed buffer size.

The idea of separate FIFO buffers is illustrated in Figure 3.5 in which the size of the buffer of Figure 3.1(c) is increased to four location. The expanded HSDF model of Figure 3.5 illustrates that in that case, two separate cycles can be distinguished. One cycle through actors $v_{i,0}$ and $v_{j,0}$ and a cycle through the other two actors. These separate cycles can be translated to separate FIFO buffers connecting tasks $\tau_{j,0}$ and $\tau_{j,1}$ to task τ_i of Figure 3.1(c). Which buffer a task needs to write to or read from during an execution can be computed with a modulo operation which only adds a constant cost, i.e., not dependent on the amount of data parallelism.

3.4 MODELING DATA PARALLELISM

In this section we describe the modeling of data parallel tasks. The first subsection summarizes the definition of the SDF analysis model and its properties that are used in this chapter. The second subsection presents the dataflow model of a

Figure 3.5: Situation in which the buffer of Figure 3.1(c) has a size of four location and can be split into separate FIFO buffers.

data parallel task and the conformance between buffers and the dataflow model. The third subsection shows that the dataflow model allows the same scheduling freedom as a data parallel task and in the fourth subsection we prove that such a data parallel task is modeled conservatively. These first four subsections assume that a tasks do not share resources. The last subsection combines data parallel tasks with run-time budget schedulers to illustrate that the method is more generally applicable.

3.4.1 DATAFLOW MODEL

The model we use to determine the required amount of data parallelism is the SDF model as defined in Chapter 2. We use the same definition of the SDF model as in Chapter 2 and only summarize its for this chapter relevant properties briefly.

The SDF model has a monotonic temporal behavior as is discussed in Chapter 2. The important implications of this monotonic behavior is that decreasing the firing duration of an actor can never lead to a later enabling of any of the actors in the graph. Furthermore, increasing the number of initial tokens on an edge in the graph can never lead to a later enabling of any actor in the graph.

Another important property of SDF models is consistency. For an inconsistent SDF model, either any finite number of initial tokens will result in deadlock or an unbounded accumulation of tokens on an edge.

In consistent SDF models, the average rate at which tokens are produced on an edge is equal to the average consumption rate on that edge. Therefore, a repetition vector $\mathbf{q}$ can be determined which contains the relative firing frequencies between the actors. We use q_i for the repetition factor of actor v_i where q_i is the i th component of $\mathbf{q}$. When every actor v_i in an SDF model fires exactly q_i times, it holds for each edge in the SDF model that the number of tokens produced on an edge is equal to the number of consumed tokens from that edge:

$$\forall_{e_{ij} \in E} : q_i \cdot \pi(e_{ij}) = q_j \cdot \gamma(e_{ij}) \quad (3.1)$$

Figure 3.6: Relation between task graph including data parallelism (bottom) and the corresponding dataflow model (top).

3.4.2 CONFORMANCE RELATION BETWEEN TASK GRAPH AND DATAFLOW MODEL INCLUDING DATA PARALLELISM

The task graph specifies the behavior of an application consisting of parallel tasks connected by buffers. The corresponding SDF model is used to determine temporal properties and suitable parameter values for the application. Figure 3.6 shows the conformance relation between the SDF model and the tasks in such a task graph. It differs from the conformance relation presented in Chapter 2. Each task is modeled using one dataflow actor. The amount of data parallelism of a task is modeled using an edge from the corresponding actor back to itself. The number of initial tokens on this self-edge corresponds with the amount of data parallelism of that task. This is in contrast to the correspondence relation introduced in Chapter 2 where the self-edge always contained one initial token. In Figure 3.6, task τ_j is replicated n times and is modeled using one SDF actor, v_j , with a self-edge containing n initial tokens. Note that we use a constant firing duration for the SDF actor corresponding to a task. Therefore, this one-actor model is a valid actor component as defined in Chapter 2.

Self-edges are also used to model stateful tasks. Such a self-edge then contains a fixed number of initial tokens to denote that the task communicates with itself. The number of tokens on this self-edge reflects the execution to which the state is written. For example a task which writes state to the next execution of the task corresponds with an actor which has a self-edge with one token. The self-edge which models the state of a task, forms an upper bound on the possible data parallelism of a task which is thus made explicit by modeling the this state in a

similar fashion as the data parallelism.

To illustrate the idea of the analysis approach, we assume in this section that tasks do not share resources with other tasks. The response time of a task, i.e., the maximum time a task takes to finish an execution after it is enabled, is thus less or equal to the WCET of the task. When the firing duration of each actor is chosen to be equal to the WCET of the corresponding task, the dataflow model is conservative, see the refinement relation presented in Section 2.5.5. Tasks communicate via buffers and the synchronization with these buffers also takes time. We assume that this synchronization time is included in the WCET. In Section 3.3 we have shown that the buffers can be implemented in such a way that the amount of synchronization operations is independent of the amount of data parallelism that is used. Therefore, the synchronization time can be included in the WCET of a task.

As discussed in Section 3.3, tasks communicate via circular buffers that have support for multiple producers and multiple consumers [DBC⁺07, BBS11]. This enables data parallel tasks to share a buffer.

We shortly repeat how tasks communicate and how this is modeled in the corresponding dataflow model. A producing task first acquires empty locations from the buffer and after the task has filled them with data, the full locations are signaled as written. Consuming tasks use these full locations and release them as empty locations after the locations are not needed anymore. This behavior of a buffer is modeled in the dataflow model using two oppositely directed edges, e_{ij} and e_{ji} as is shown in Figure 3.6. Edge e_{ij} models the flow of full locations and edge e_{ji} the flow of empty locations. The skipped locations are not modeled because they do not add additional synchronization. The number of initial tokens on edge e_{ij} , δ_{ij} , corresponds to the number of initially filled locations in the corresponding buffer and the δ_{ji} initial tokens on edge e_{ji} correspond to the initially empty locations in the buffer. The buffer contains in total $\delta_{ij} + \delta_{ji}$ locations.

Tasks read/write during every execution the same amount of locations in a buffer. The number of locations that are read/written each execution, corresponds with the number of tokens that are consumed from/produced to the corresponding queue in the dataflow model. Consider a buffer from task τ_i to task τ_j . Then we have $\gamma(e_{ji}) = \pi(e_{ij}) = \psi(c_{ij})$ with $\psi(c_{ij})$ the number of locations written by task τ_i in the buffer from task τ_i to the replications of task τ_j . And similarly, we have $\gamma(e_{ij}) = \pi(e_{ji}) = \chi(c_{ij})$ with $\chi(c_{ij})$ the number of locations that are read by a replication of task τ_j from that same buffer. This is also illustrated in Figure 3.6.

The order in which the different data parallel replications of a task read/write locations in the buffer is fixed. This allows us to define a static mapping between the executions of the data parallel tasks and the corresponding actor firings such that the read and written locations correspond with the consumed and produced tokens, respectively. We use this mapping in the next sections to prove that the correspondence relation between task graph and dataflow model on the one hand preserves scheduling freedom and on the other hand is conservative.

Executions of tasks are mapped to firings of dataflow actors as follows. Consider a data parallel task τ_d with δ_d replications and an actor v_d which corresponds to this task. We identify replication k of τ_d where $0 \leq k < \delta_d$. Execution n of replication k of τ_d then corresponds with firing $k + n \cdot \delta_d$ of actor v_d with $n \in \mathbb{N}$.

3.4.3 ACCURATE MODELING

In this section we show that the introduced modeling of data parallelism by a dataflow actor allows the same scheduling freedom as the corresponding data parallel task replications.

Without loss of generality we assume for the proof that there is only one task which is data parallelized. We call the actor corresponding to this task, actor v_d . The amount of data parallelism of this task is equal to δ_d which means that the self-edge dependency of the corresponding dataflow actor should allow δ_d simultaneous firings.

To show this we use the expansion of an SDF graph to an equivalent HSDF graph [LM87, SB00]. As is already presented in Chapter 2, an HSDF graph is an SDF graph in which all the numbers of consumed and produced tokens per firing, are equal to 1. Note that we use this expansion to an equivalent HSDF graph merely to prove the scheduling freedom. We do not use this HSDF graph itself to model the behavior of the task graph. We use a modified expansion method to make the data parallelism visible.

The expansion of an SDF graph to an equivalent HSDF graph makes use of the balance equation from (3.1). Each SDF actor v_i is modeled using q_i HSDF actors. Each firing of an HSDF actor corresponds to a firing of the SDF actor. After every HSDF actor has fired exactly once, the tokens are again on the same edge as in the initial situation. This corresponds to q_i firings of SDF actor v_i .

The expansion to HSDF translates the edges of the SDF model to multiple edges in the HSDF model based on the communication between firings of the actors in the SDF model. Also self-edges are translated to multiple edges in the HSDF model. A self-edge for actor v_d with δ_d tokens specifies in an SDF model that firing n of v_d can only start when firing $n - \delta_d$ of v_d is finished. This is modeled in the HSDF model with an edge between the actors modeling firing n and firing $n - \delta_d$.

To show the scheduling freedom of the dataflow model we use an expansion of an SDF model to an equivalent HSDF model which differs from the expansion method in [LM87, SB00]. The balance equation from Equation (3.1) which couples the relative firing frequencies of actors, can be rewritten to include the amount of data parallelism of actor v_d :

$$\forall_{e_{ij} \in E} : q_i \cdot \delta_d \cdot \pi(e_{ij}) = q_j \cdot \delta_d \cdot \gamma(e_{ij})$$

Instead of expanding each SDF actor v_i to q_i HSDF actors, we can thus also model an SDF actor v_i with $q_i \cdot \delta_d$ HSDF actors. Actor v_d is then modeled using

Figure 3.7: Expansion of an SDF actor modeling data parallelism (left) to HSDF actors (right).

$q_d \cdot \delta_d$ HSDF actors. We call this new HSDF model the expanded dataflow model.

Figure 3.7 shows how an SDF actor v_d with a self-edge containing δ_d initial tokens is modeled using $q_d \cdot \delta_d$ HSDF actors. Because we model v_d with $q_d \cdot \delta_d$ HSDF actors, the self-edge dependencies in the HSDF graph have the illustrated structure. We can distinguish δ_d groups of q_d HSDF actors as is shown in Figure 3.7. The different groups in Figure 3.7 start with firing the actors denoted by v_d^0 to $v_d^{\delta_d-1}$. No self-edge dependencies exist between these δ_d groups, only inside the group itself.

Each of these groups models exactly one of the data parallel replications of the task. Group k , with $0 \leq k < \delta_d$ models the firings $i = k + n \cdot \delta_d$, $n \in \mathbb{N}$ of the SDF actor. These firings correspond, as defined in Section 3.4.2, to execution n of replication k of the corresponding data parallel task. Group k of HSDF actors thus models the executions of task replication k .

Because there are no self-edge dependencies between the different groups of HSDF actors, there is also no dependency in the schedule between the groups when the self-edge dependencies are considered. Each of the δ_d groups of HSDF actors can thus be scheduled individually which means that the dataflow model allows the same scheduling freedom as the data parallel task.

3.4.4 PROOF OF CONSERVATIVENESS

The previous section showed that the data parallelism of tasks is accurately modeled with the proposed dataflow model. In this section we prove that the dataflow model for data parallel tasks as proposed in this chapter is also conservative. We prove that the production times of tokens in the dataflow model form an upper bound on the times on which the tasks write to the corresponding buffer locations.

Figure 3.8: Illustration of the refinement relation for data parallel tasks.

In Chapter 2 sufficient conditions are derived under which dataflow models are temporally conservative to task graphs. One condition is that the number of consumed and produced tokens on each edge in the dataflow model needs to be equal to the number of read and written locations of the corresponding connection in the task graph. This is ensured by the correspondence between dataflow model and task graph presented in Section 3.4.2. Given this correspondence it is shown in Chapter 2 that when the following refinement equation holds for every task and corresponding actor, the whole dataflow model is conservative to the task graph.

$$\forall i \in \mathbb{N} : e\langle i \rangle \sqsubseteq e'\langle i \rangle \implies f\langle i \rangle \sqsubseteq f'\langle i \rangle \quad (3.2)$$

We use $e(i)$ for the external enabling time of execution i of a task, i.e., the time at which the task can read sufficient locations from the adjacent buffers. The external enabling time of the corresponding actor firing is denoted by $e'(i)$. This is the earliest time at which the required tokens are present on the incoming queues. Furthermore, $f(i)$ is the finish time of execution i of the task and $f'(i)$ is the finish time of firing i of the corresponding actor.

As shown in Chapter 2, when the local condition of Equation (3.2) holds for every task and corresponding actor, all the production times of tokens form upper bounds on the times at which buffer locations are filled. This means that the complete dataflow model is temporally conservative to the task graph.

However, Equation (3.2) is only applicable when one actor corresponds with one task whereas we address the case in which an actor corresponds to multiple replications of the same task. Therefore, we extend Equation (3.2) to data parallel tasks and show that it holds for the dataflow analysis method as proposed in this chapter.

We use the mapping between task executions and dataflow firings presented in Section 3.4.2 to derive Equation (3.3) in which the sequence e is split into the different sequences corresponding to the data parallel task replications. Execution n

of replication k of a task τ_d corresponds with firing $k + n \cdot \delta_d$ of the corresponding dataflow actor v_d . We use $e^k(n)$ for the external enabling time of execution n of replication k of task τ_d and $f^k(n)$ for the finish time of execution n of replication k . This refinement relation for data parallel tasks is illustrated in Figure 3.8.

$$\forall_{0 \leq k < \delta_d, n \in \mathbb{N}} : e^k\langle n \rangle \sqsubseteq e'\langle k + n \cdot \delta_d \rangle \implies f^k\langle n \rangle \sqsubseteq f'\langle k + n \cdot \delta_d \rangle \quad (3.3)$$

We now prove this relation for the proposed dataflow model of data parallel tasks. By using max-plus algebra semantics [BCOQ92] of dataflow actors on actor v_d we obtain:

$$f'(i) \geq \max(e'(i), f'(i - \delta_d)) + \rho_d \quad (3.4)$$

The firing duration, ρ_d , of actor v_d is equal to the WCET of the corresponding task τ_d . Next to that, different executions of replication k of task τ_d cannot overlap. It therefore holds that:

$$f^k(n) \leq \max(e^k(n), f^k(n - 1)) + \rho_d \quad (3.5)$$

We now prove Equation (3.3) by using induction on execution number n . We make use of the left part of Equation (3.3), i.e., the enabling times of the dataflow actor are conservative to the enabling times of the corresponding task execution, to prove the right part of Equation (3.3). We can safely assume that the finish times of executions numbers smaller than 0 are equal to $-\infty$ to denote that they do not exist. With this assumption the base case of the induction proof, $n = 0$, can be proven. We prove $\forall_{0 \leq k < \delta_d} : e^k(0) \leq e'(k) \implies f^k(0) \leq f'(k)$ which is equivalent to Equation 3.3 with $n = 0$.

$$\begin{aligned} \forall_{0 \leq k < \delta_d} : f^k(0) &\leq \max(e^k(0), f^k(-1)) + \rho_d \\ &\leq \max(e^k(0), -\infty) + \rho_d \\ &\leq e^k(0) + \rho_d \\ &\leq e'(k) + \rho_d \\ &\leq f'(k) \end{aligned} \quad (3.6)$$

For the induction step we prove that when Equation (3.3) holds for execution $n - 1$ of a replication k , it also holds for execution n of replication k . When Equation (3.3) holds for execution $n - 1$ and given $e^k\langle n \rangle \sqsubseteq e'\langle k + n \cdot \delta_d \rangle$ we have: $f^k\langle n - 1 \rangle \sqsubseteq f'\langle k + (n - 1) \cdot \delta_d \rangle$ and thus $f^k(n - 1) \leq f'(k + (n - 1) \cdot \delta_d)$. With $e^k\langle n \rangle \sqsubseteq e'\langle k + n \cdot \delta_d \rangle$ we have $e^k(n) \leq e'(k + n \cdot \delta_d)$ and thus:

$$\begin{aligned} \forall_{0 \leq k < \delta_d} : f^k(n) &\leq \max(e^k(n), f^k(n - 1)) + \rho_d \\ &\leq \max(e'(k + n \cdot \delta_d), f'(k + n \cdot \delta_d - \delta_d)) + \rho_d \\ &\leq f'(k + n \cdot \delta_d) \end{aligned} \quad (3.7)$$

Figure 3.9: Actor component modeling the temporal behavior of a task scheduled with a run-time budget scheduler.

$$\forall_{0 \leq k < \delta_d} : f^k \langle n-1 \rangle \sqsubseteq f' \langle k + (n-1) \cdot \delta_d \rangle \wedge f^k(n) \leq f'(k + n \cdot \delta_d) \implies f^k \langle n \rangle \sqsubseteq f' \langle k + n \cdot \delta_d \rangle \quad (3.8)$$

This proves that Equation (3.3) holds for every execution n which means that data parallel tasks are conservatively modeled by the proposed dataflow model.

3.4.5 COMBINATION WITH RUN-TIME SCHEDULERS

The previous sections assumed that resources are not shared between tasks. In this section we show that the approach is also applicable when tasks do share resources, for example the processor on which they execute. We do this by proving that the analysis method introduced in this chapter is also applicable for latency-rate actor components which are used to model the temporal behavior of tasks scheduled with run-time budget schedulers [WBS09]. Such latency-rate actor components are more widely applicable and can for example also be used to model variations in execution times as will be shown in Chapter 4.

Run-time budget schedulers guarantee every task a minimum time budget S in every time interval of length Q . The temporal behavior of a task can be conservatively analyzed using a latency-rate actor component. Such an actor component is shown in Figure 3.9 for a data parallel task τ_d . Each of the replications of τ_d have the same time budget S_d . In this Figure, C_d is used for the WCET of task τ_d and δ_d corresponds to the amount of data parallelism of τ_d .

It can be shown by using the same method as in Subsection 3.4.3 that the actor component from Figure 3.9 models the amount of data parallelism accurately. We therefore only provide the proof that the actor component is temporally conservative to the corresponding task.

The finish time of a task τ_d , scheduled with a run-time budget scheduler can be bounded as follows [WBS09]:

$$f(i) \leq \max(e(i) + Q_d - S_d, f(i-1)) + \frac{Q_d \cdot C_d}{S_d} \quad (3.9)$$

Each replication of a task has its own budget. Therefore, this finish time can also be used for a replication k of a task:

$$f^k(n) \leq \max(e^k(n) + Q_d - S_d, f^k(n-1)) + \frac{Q_d \cdot C_d}{S_d} \quad (3.10)$$

Using the max-plus semantics of the actor component of Figure 3.9 we obtain:

$$\hat{f}(i) \geq \max \left(\hat{e}(i) + Q_d - S_d, \hat{f}(i - \delta_d) \right) + \frac{Q_d \cdot C_d}{S_d} \quad (3.11)$$

By using Equations (3.10) and (3.11) we can prove Equation (3.3) by induction on execution number n . This proof is omitted because it is analogous to the induction proof of Section 3.4.4.

3.5 DETERMINING DATA AND PIPELINE PARALLELISM

With the introduced modeling of data parallelism by means of one actor with a self-edge containing multiple tokens, we can also derive the required amount of data parallelism. In this section we present a method for this derivation.

Data parallelism of a task is modeled as a cyclic dependency on the corresponding SDF actor itself. Because buffers are also modeled as cyclic dependencies, determining data parallelism is strongly related to analysis methods which determine the required buffer sizes given a throughput constraint. When these buffer sizing methods are instrumented correctly, they can also be used to determine data parallelism.

Buffer sizing methods such as [SGB06, WBS06, MBGS10, GHKB13] use graph algorithms to determine the minimum required amount of tokens that allow the required throughput. Also the costs of tokens can be taken into account when determining this minimum required amount of tokens. The methods make use of the fact that increasing the amount of tokens on edges can make tokens on other edges redundant. The costs of tokens can be used to resolve the trade-off between the amounts of tokens on the different edges. These costs are thus used to express a preference for which edges should contain the least amount of tokens.

The dataflow schedules that are used in these buffer sizing methods inherently exploit pipeline parallelism. Given the data dependencies as specified in the dataflow model it is explicit which firings of actors may overlap. The amount of tokens in the cycles form an upper bound on the amount of pipeline parallelism that can be used.

The buffer sizing methods can also be used to determine the required amount of data parallelism. The number of tokens can be determined for the self-edge cycle modeling the data parallelism such that the temporal constraints are met. This allows to determine the required amount of data parallelism simultaneously with the amount of pipeline parallelism and the corresponding buffer sizes which makes the trade-off between data and pipeline parallelism explicit. By using suitable token costs, the choice can be made between increasing buffer sizes or increasing the amount of data parallelism. Also it can be determined how large the buffers need to be to exploit a certain degree of data parallelism.

Example 3.2:

Figure 3.10 shows an example demonstrating the trade-off between data and

Figure 3.10: Example of a trade-off between data and pipeline parallelism.

δ_{ba}	δ_b	μ
4	1	80
5	1	70
6	1	60
4	2	60
6	2	40
7	2	35
8	2	30
7	3	30
10	3	20

Table 3.1: Results for Figure 3.10 illustrating the average period for different token assignment.

pipeline parallelism. It presents the dataflow model corresponding to two tasks, one producer and one consumer. The producer, modeled with actor v_a , has a WCET of 10 time units and is a stateful task which cannot be data parallelized. Each execution, three locations are filled with data by this task. The consumer task, modeled with actor v_b , reads two locations per execution. This task has a WCET of 20 time units and can be data parallelized. A choice can be made between increasing the buffer connecting the two tasks (increasing δ_{ba}) and allowing more pipeline parallelism or increasing the amount of data parallelism of the consuming task (increasing δ_b). This illustrates the trade-off between data and pipeline parallelism.

The results of exploiting this trade-off are shown in Table 3.1. The first two columns show the different assignments of initial tokens to edge e_{ba} and the self-edge of actor v_b . The third column, denoted by μ , shows the average period that can be achieved given the token assignment. It corresponds to the average period in which each SDF actor v_i fires q_i times. For example, for the first row μ is equal to 80 time units which means that on average, v_a can fire twice per 80 time units and v_b on average can fire three times per 80 time units. This period is the inverse of the throughput of the application and is calculated using an MCM algorithm [Das04, SB00]. Only the minimum numbers tokens are illustrated per average period. These minimum numbers of tokens are determined using an exhaustive search algorithm which determines the minimum number of tokens given a throughput constraint. This exhaustive search algorithm tries the relevant token assignments and searches for the token assignment which adheres to the throughput constraint (checked with the MCM algorithm) and which gives the minimum sum of tokens multiplied by their token costs. The MCM algorithm is exact and also the exhaustive search is complete which implies that we find the exact numbers of required tokens for each average period. For each average period, two exhaustive searches are performed. One version in which the token costs of tokens corresponding to buffer locations are increased and which thus prefers data parallelism and the other version prefers smaller buffer sizes over data parallelism.

Figure 3.11: Two different schedules for Figure 3.10 with an average period of 60 time units.

As is shown in Table 3.1, the average period decreases when the amount of tokens increases. The lowest average period that can be reached by the task graph is 20 time units. For this period, actor v_a limits the throughput due to its self-edge. Table 3.1 also shows the trade-off between increasing the buffer size (increasing δ_{ba}) and increasing the amount of data parallelism (increasing δ_b). For example, the third and the fourth row in the table result in an equal μ . This means that the throughput that is achieved when the size of the buffer is equal to six locations and no data parallelism is used, is equal to the throughput that is achieved when the buffer size is equal to four locations and the consuming tasks is replicated twice.

As discussed, buffer sizing methods can be used to determine the required amount of data parallelism next to the required buffer sizes. Exact buffer sizing methods such as [SGB06, MBGS10] have a non-polynomial time-complexity which can be problematic for larger dataflow graphs. Buffer sizing methods exist with a polynomial time-complexity [WBJ06, GHKB13] and are based on linearizations of schedules. The schedules of the dataflow actors are approximated with linear equations which actually model a strictly periodic token production and consumption. Such linear equations can be used to solve the buffer sizing problem efficiently, but can result in non-optimal token placements. Because of the strictly periodic token production, the trade-off between data and pipeline parallelism cannot be exploited as good as with the exact buffer sizing methods.

This is illustrated in Figure 3.11 which shows two different schedules with an average period of 60 time units for the dataflow model of Figure 3.10. Figure 3.11(a) shows a schedule that exploits data parallelism. Edge e_{ba} contains four tokens and δ_b is equal to two. The horizontal axis denotes time and the rectangles in the figure represent the different firings of actors v_a and v_b . The value in the top right corner of each rectangle indicates the sequence number of each firing. The schedule of Figure 3.11(a) exploits data parallelism and is not strictly periodic. Figure 3.11(b) shows the only possible strictly periodic schedule with an average period of 60 time units. The schedule from Figure 3.11(a) is not possible when only strict periodic schedules are considered. The required number of tokens for the periodic schedule of Figure 3.11(b) is six for edge e_{ba} .

Figure 3.12: Dataflow model of the packet decoding mode of a WLAN 802.11p transceiver.

and δ_b is equal to one. The trade-off as shown in Table 3.1 for an average period of 60 time units can not be made when periodic schedules are considered.

3.6 CASE-STUDY

In this section we show the applicability of the analysis method by applying it to a WLAN 802.11p transceiver [AHG07]. The application is executed on an Multiprocessor System-on-Chip (MPSoC) and data parallelism as well as pipeline parallelism is required to meet the throughput constraint of the application. We will consider the part of the WLAN 802.11p transceiver which is active during packet decoding mode.

Figure 3.12 shows the SDF model corresponding to this packet decoding mode. The numbers of tokens that are produced and consumed are only drawn when they are unequal to one. The throughput constraint is imposed by the periodic source (SRC) which produces one sample every $8\mu s$ (frequency = $125kHz$). The samples that are produced by the source are first processed by a filter and then by an FFT. The packets are then subsequently decoded using the channel equalizer (EQ), demapper (DEMAP), deinterleaver (DEINT) and finally the viterbi decoding (VIT) task. The packets are encoded for error correction with a redundancy rate of $\frac{1}{2}$ which means that each transmitted m bits contains $\frac{m}{2}$ actual data bits. The viterbi decoding task thus decodes m input bits into $\frac{m}{2}$ error corrected output bits. The viterbi task outputs therefore only one output sample per two consumed input samples.

The result of the channel equalizer is improved by using a channel estimator (CH-EST). The output of the viterbi task contains the error corrected bits. These corrected bits are re-encoded (RE-ENC) and compared with the output of the FFT. Based on the difference between the bits, an estimation of the channel is computed by the channel estimation task. The estimation that is computed using sample i is used as the setting of the channel equalizer during the reception of sample $i + 2$. This is modeled using two initial tokens on the channel estimation cycle.

δ_{SRC}	1	$\delta_{FILTER, SRC}$	1
δ_{FILTER}	1	$\delta_{FFT, FILTER}$	3
δ_{FFT}	3	$\delta_{EQ, FFT}$	4
δ_{EQ}	1	$\delta_{CH-EST, FFT}$	6
δ_{DEMAP}	1	$\delta_{DEMAP, EQ}$	1
δ_{DEINT}	1	$\delta_{DEINT, DEMAP}$	1
δ_{VIT}	1	$\delta_{VIT, DEINT}$	2
δ_{RE-ENC}	1	$\delta_{RE-ENC, VIT}$	1
δ_{CH-EST}	2	$\delta_{CH-EST, RE-ENC}$	2
		$\delta_{EQ, CH-EST}$	0

Table 3.2: Result of the analysis method applied on the dataflow model of Figure 3.12.

Figure 3.12 only shows the edges on which the tokens cannot be chosen freely. The other edges are omitted for clarity. The behavior of the viterbi and re-encoding tasks depends on the result of the previous executions plus the current sample. They thus do have state which is modeled using self-edges with one token. On the other edges, self-edges as well as edges modeling buffer locations, the number of tokens that are required to meet the throughput are computed using the analysis method described in Section 3.5. The result of this analysis is shown in Table 3.2. It contains the required number of tokens for the omitted edges. According to the results, two tasks need to be replicated, the FFT task and the channel estimation task. The results are determined with the same exhaustive search approach as discussed in Example 3.2.

The FFT task has a high execution time and therefore needs to be replicated three times to meet the throughput constraint. To exploit this introduced data parallelism, the sizes of the buffers connecting the FFT task need to be sized correctly. The sizes are 3, 4 and 6 locations for the buffer with the FFT, the equalizer and the channel estimator, respectively.

Due to the fixed number of two tokens in the channel estimation loop, the amount of pipeline parallelism that can be used in this loop does not suffice to meet the throughput constraint. However, the analysis method has determined that when the channel estimation task is replicated twice, the throughput can be met.

3.7 CONCLUSION

The analysis method introduced in this chapter allows to model data parallelism accurately without having to replicate dataflow actors. Instead, the data parallelism is modeled using a cyclic self dependency with the number of tokens on this cycle corresponding to the amount of data parallelism. We have shown

that the parameterized dataflow model is conservative to the corresponding task graph by making use of a new SDF to HSDF conversion.

It is shown that with the introduced dataflow model, the required buffer sizes and the amount of data and pipeline parallelism can be determined simultaneously, which makes the trade-off between them explicit. Furthermore, it has been shown that existing buffer sizing methods can be used to choose between increasing buffer sizes or increasing data parallelism in order to meet the throughput constraint.

It is also shown that task graphs containing data parallelism can be implemented with circular buffers with support for multiple producers and multiple consumers. These buffers can be implemented such that the amount of synchronization operations are not dependent on the amount of data parallelism that is used. It is shown that this property enables the proposed analysis model combining data and pipeline parallelism. Furthermore, we have shown that if FIFO buffers are applied instead of circular buffers, the parameterized dataflow model is only conservative if the computed number of tokens is an integer multiple of the amount of replication of the tasks.

The applicability of the analysis method is demonstrated with a WLAN 802.11p application. It is shown that despite the cyclic re-encoding loop of this application, the amount of data parallelism can be derived simultaneously with the amount of pipeline parallelism such that the throughput requirements of the application can be met. Also the required buffer sizes have been determined by using the proposed analysis method. The proposed analysis method can thus be used to make the trade-off between data and pipeline parallelism.

Two Parameter Workload Characterization

ABSTRACT – Existing dataflow analysis techniques use the Worst-Case Execution Times (WCETs) of tasks to determine minimum throughput and maximum latency guarantees of applications. These methods characterize the workload of tasks with one parameter. However, a large difference between the WCETs of the tasks and their average execution times can result in a large difference between the computed worst-case throughput and the actual obtained throughput.

To reduce the difference between the computed throughput guarantee and the actual obtained throughput, we introduce in this chapter a two parameter (σ, ρ) workload characterization of the tasks to improve the accuracy of dataflow analysis. The (σ, ρ) workload characterization captures information on the maximum cumulative execution time of consecutive executions of a task and can therefore be seen as a generalization of the WCET characterization.

One important metric of worst-case temporal analysis methods is their accuracy. The accuracy of traditional analysis based on SDF graphs is limited because it is based on a workload characterization with one parameter, the WCET of the task [SB00]. Such a simple workload characterization can result in a large difference between the computed minimum throughput and the actual throughput if there is large difference between the WCETs and the average execution time of the tasks. The same holds for tasks modeled as a CSDF [EBLP94, BLP96,

The material in this chapter is based on work presented in [HGWB13b].

WBS07] actor which uses only one parameter per phase. Until now a constant workload is thus considered for SDF graphs and a periodic workload is considered for CSDF graphs.

The difference between the computed worst-case and the actual throughput can be reduced by making use of knowledge about the maximum sum of execution times for a number of consecutive executions. The fact that the worst-case cumulative execution time of n subsequent executions of a task is often smaller than n times the WCET of this task can be exploited. Knowledge of the worst-case cumulative execution time can be captured in so-called workload functions [MKT04]. The use of aperiodic workload functions has not been considered in combination with dataflow analysis methods.

This chapter introduces an analysis technique to represent workload functions characterized with two parameters, σ and ρ , in a dataflow model. This dataflow model can be analyzed with standard dataflow analysis techniques. The analysis of this dataflow model will result in a tighter minimum throughput bound and can be applied in combination with run-time scheduling.

The presented approach can be used in combination with functional deterministic dataflow models for which a minimum throughput and maximum latency can be computed at design time. Examples of these models are the HSDF, SDF [LM87, SB00], CSDF [MBBM07] and VRDF [WBS08b] models.

The case-study shows how the (σ, ρ) workload characterization can be used. We will show that tighter analysis results can be obtained for a DVB-T radio application when using the (σ, ρ) workload characterization instead of the traditional WCET workload characterization. Next to that, we illustrate the effect of a latency constraint on the temporal results that can be obtained using a (σ, ρ) workload characterization with a car-radio application. We also combine the (σ, ρ) characterization with a data-dependent MP3 playback application which is modeled using a VRDF graph.

The outline of this chapter is as follows. We discuss related work in Section 4.1. In Section 4.2 we describe the basic idea behind the (σ, ρ) based analysis method and in Section 4.3 we present the improved dataflow model and prove its correctness. Section 4.4 combines (σ, ρ) information with the modeling of cyclic temporal behavior. An evaluation of the (σ, ρ) workload characterization in combination with dataflow analysis techniques is presented in Section 4.5 and we conclude this chapter in Section 4.6.

4.1 RELATED WORK

While [WBS09] introduced a modeling relation between actor components and tasks to analyze the effects of run-time scheduling, state of the art in modeling the task workload function is to have a conformance relation between dataflow firings and task executions. Each firing has a duration equal to an upper bound on the corresponding task execution time the WCET of the task [WBS09, MVB07,

WBS06, GGBS08]. Our method does not have this restrictive one-to-one correspondence between firings and executions, instead we introduce a method that uses the modeling relation between actor components and tasks to capture the workload functions of the tasks. In [WBS07b], similar actor components as we use in this chapter are used but the model from [WBS07b] also only allows to use one single (worst-case) execution time per task. We show in this chapter that our analysis method can be combined with [WBS09] to also take the effects of run-time scheduling into account.

Methods that use knowledge about the execution time of consecutive executions and which aims at improving the accuracy of the temporal analysis results are presented in [MKT04, QHE12]. They use workload functions to express changes in execution times of different modes of an application. However, these workload functions cannot be used directly in combination with dataflow models because they have an arbitrary infinite structure which cannot be described by a finite number of parameters. Dataflow models currently use only one parameter, the firing duration, and can therefore not include the effects of an arbitrary workload function. We present a workload characterization which can be combined with existing dataflow models and analysis techniques, and which can be described by two parameters. This workload characterization can be seen as an upper bound on a workload function. Methods which use an arbitrary workload function [MKT04, QHE12] use iterative fixed-point computation which results in a high worst-case computational complexity. Dataflow analysis techniques do not necessarily require such iterative fixed-point computation and do therefore often have a better worst-case computational complexity.

Workload functions can also be used to express cyclic temporal behavior. However, the methods of [MKT04] and [QHE12] have no mechanism to express this cyclic behavior. This loss of information can lead to an overestimation. Consider a task with cyclic temporal behavior that always starts with three executions which take maximally 1 time unit, followed by an execution that takes maximally 4 time units. The methods of [MKT04] and [QHE12] cannot distinguish between this sequence and, for example, its reverse; i.e., starting with the longest execution. They cannot distinguish between such sequences because their service curves are the time-interval domain, see Chapter 1. In this time-interval domain, the longest execution must be assumed as the first execution. The actual order of the consecutive executions is lost by definition.

Our method will behave similarly, when we model this task with actors without phases; e.g., using HSDF or SDF models. However, as we will explain further in Section 4.4, we can model the cyclo-static sequence of execution times explicitly by modeling this task with for example a CSDF actor, resulting in a more accurate model.

Other related methods are based on probabilistic execution times [BCP02, HZS01, TDS⁺95]. However, all these methods are unsuitable for the derivation of the worst-case throughput and typically have a high computational complexity [SB00]. Our introduced model enables to give temporal guarantees while

taking variation of execution times into account by making use of information about consecutive execution times. Existing dataflow analysis methods can be used to analyze the temporal behavior of the application when the introduced model is used.

The method presented in this chapter is independent of the number of tokens consumed and produced per execution. This means that our method can be combined with tasks that have data-dependent behavior [WBS08b] which is not considered for [MKT04] and [QHE12].

4.2 BASIC IDEA

This section contains the basic idea behind the (σ, ρ) workload characterization and how it can be applied. We first informally introduce the (σ, ρ) workload characterization. We then show the basic idea behind the conservative temporal analysis methods based on dataflow models and we conclude this section with an example that illustrates the benefits of using a (σ, ρ) workload characterization in combination with a dataflow model.

4.2.1 (σ, ρ) WORKLOAD CHARACTERIZATION

The (σ, ρ) workload characterization forms an upper bound on the worst-case cumulative execution time of a task. It bounds this worst-case cumulative execution time with a function which can be described by two parameters, σ and ρ . These two parameters can be used as the execution time description of a task for which we can provide worst-case temporal analysis results.

Figure 4.1 illustrates the idea of the (σ, ρ) workload characterization. Figure 4.1(a) shows the worst-case trace of execution times of seven executions of a task. Figure 4.1(b) contains the worst-case cumulative execution time $\omega(n)$ for up to seven consecutive executions, where ω is what we call the workload function of the task. The first bar in the figure equals the WCET of the task. The second bar is found by searching in the trace of Figure 4.1(a) for the worst-case sum of execution times of two consecutive executions. The other bars are determined similarly.

The solid line in Figure 4.1(b) denoted by $\hat{\omega}$ shows the upper bound specified by the σ and ρ parameters which will be introduced in this chapter. Figure 4.1(c) shows the average worst-case execution time corresponding to the worst-case cumulative execution time of Figure 4.1(b). The solid line denoted by α is the average worst-case execution time corresponding to the upper bound $\hat{\omega}$ of Figure 4.1(b).

Figure 4.1(c) also contains the average worst-case execution time when only using the WCET of the task (the horizontal gray line denoted by WCET). It illustrates the throughput improvement that can be achieved when using the (σ, ρ) workload characterization for this example. The minimum throughput, that can be guaranteed by temporal analysis methods, doubles for this example if the (σ, ρ) workload characterization is used instead of the WCET.

Figure 4.1: Worst case execution time information of consecutive executions.

4.2.2 CONSERVATIVE TEMPORAL ANALYSIS

To ensure that a dataflow model is temporally conservative to a task graph, sufficient conditions on the relation between such a dataflow model and a corresponding task graph are presented in [WBS09]. This means that the temporal analysis results (minimum throughput and maximum latency) calculated using the dataflow model are also valid for the corresponding task graph.

The remainder of this subsection summarizes these results from [WBS09]. The sufficient conditions are based on a conformance relation between the task graph and the dataflow model. Containers in the buffers of the task graph correspond with tokens in the dataflow model. Finite buffers in the task graph are modeled with two queues in the dataflow model. One forward queue, modeling the data production, and one backward queue modeling the back-pressure of the finite buffer. The number of initial tokens on the forward and backward queues represent the number of initially available full and empty containers in the buffer respectively. The sum of the initial tokens on these queues corresponds with the buffer size.

Tasks in the task graph correspond with actor components in the dataflow model. An actor component is a set of actors. The execution of a task corresponds with a firing of the actor component. A task is externally enabled when sufficient containers are available on all adjacent buffers. The corresponding actor component is externally enabled if the tokens that correspond with the required containers are present on the queues. We call the time at which a execution or firing is externally enabled the enabling time.

With the conformance relation between the task graph and the dataflow model, it is sufficient to show that for each task the corresponding actor component is temporally conservative. This can be done by showing that when the enabling time of execution i of a task is smaller or equal to the enabling time of firing i of the corresponding actor component, also the time at which execution i finishes is less or equal to the time at which firing i finishes. With this it can be shown that containers never arrive later in the buffers than the corresponding tokens arrive at their queues and thus the temporal analysis results of the dataflow model are conservative.

The rectangle v_0 in Figure 4.5 shows how we model the worst-case temporal behavior of a task with a (σ, ρ) workload characterization as an actor component. The actor component consists of two dataflow actors, one modeling the rate and together modeling the latency. The actor component v_0 of Figure 4.5 consists of the rate actor $v_{0,1}$ (with self-edge to prevent overlapping firings) and an actor $v_{0,0}$ (without self-edge). The idea is that the rate actor uses ρ as its firing duration such that the minimum throughput is computed using a value closer to the average execution time than the WCET. The actor $v_{0,0}$ ensures that the actor component is still temporally conservative to the corresponding task. In the next subsection we use an example to illustrate what the benefits are of modeling the (σ, ρ) workload characterization with two actors.

Figure 4.2: Task graph consisting of two tasks connected by a buffer.

Figure 4.3: Dataflow model corresponding to the task graph of Figure 4.2.

4.2.3 DATAFLOW MODELING EXAMPLE

Consider the task graph of Figure 4.2 which contains two tasks, τ_0 and τ_1 which communicate via a finite buffer of δ locations. We assume a constant execution time for task τ_1 of 2 time units. This task is modeled with an actor component v_1 in Figures 4.3, 4.4 and 4.5. For task τ_0 we assume that we have information on the maximum cumulative execution time of consecutive executions. The WCET of this task is equal to 4 and we know that four consecutive executions of the task take maximally 8 time units which means that the average execution time equals 2. The worst-case execution time trace for four consecutive executions can be used to find corresponding values for σ and ρ which are 6 and 2 time units respectively (see Section 4.3.4). We model task τ_0 as an actor component v_0 in three different ways. The buffer is modeled with two queues between the actor components. The number of required initial tokens on the queue from v_1 to v_0 corresponds with the required capacity δ of the buffer between the two tasks.

The simplest case is illustrated in Figure 4.3. The two actor components are modeled with a single dataflow actor, each having a firing duration equal to the WCET of their corresponding task. For this case, two initial tokens are required to ensure the highest minimum throughput which is for this case equal to $\frac{1}{4}$ tokens per time unit. Two tokens are sufficient because then the self edge of v_0 forms the critical cycle which determines that the MCM [Rei68] equals 4. The throughput of the dataflow model is the inverse of the MCM.

We would now like to exploit the information on the execution time of consecutive executions to increase the minimum throughput that can be guaranteed by the temporal analysis method. A solution for this is illustrated in Figure 4.4. It combines four firings of actor v_0 into one firing which now consumes and produces four times the number of tokens consumed/produced in the original dataflow model. One firing of v_0 now corresponds with four executions of τ_0 . We

Figure 4.4: Dataflow model exploiting information about the cumulative execution time of task τ_0 of Figure 4.2 by combining firings of actor v_0 .

Figure 4.5: Dataflow model exploiting information about the cumulative execution time of task τ_0 of Figure 4.2 by using the (σ, ρ) based modeling with two actors.

know that the cumulative execution time of four consecutive executions of τ_0 is not equal to four times the WCET, but is actually maximally 8 time units. We thus can use a firing duration of 8 time units for the aggregated firing. This increases the minimum throughput to $\frac{4}{8}$ tokens per time unit. To achieve this throughput, eight initial tokens are needed. This corresponds to a buffer size, δ , of eight locations which thus needs to be four times as high to guarantee the increased minimum throughput. Note that for this solution the more expressive SDF model is required instead of the original HSDF model. The solution proposed in this chapter can be used in combination with the original type of dataflow model.

Figure 4.5 shows how including execution time information of consecutive executions in a dataflow model can refine the dataflow actor v_0 of Figure 4.4. In the remainder of this chapter we will focus on this solution. Actor component v_0 of Figure 4.5 consists of two actors instead of only one. These two actors together correspond to task τ_0 . We have one actor without a self-edge ($v_{0,0}$) and one actor with a self-edge ($v_{0,1}$). The firing duration of actor $v_{0,1}$ equals ρ , which in this example corresponds to the average worst-case execution time of the producing task (2 time units). The firing duration of actor $v_{0,0}$ is equal to $\sigma - \rho$, which is equal to 4 in this example. Modeling the information on consecutive executions like this, still leads to a guaranteed minimum throughput of $\frac{1}{2}$ tokens per time

unit. However, according to the temporal analysis for this dataflow model, four instead of eight initial tokens are sufficient to achieve this throughput.

In the next section we show that this last modeling alternative can be proven to be temporal conservative to the task graph.

4.3 DATAFLOW ANALYSIS USING THE (σ, ρ) WORKLOAD CHARACTERIZATION

This section first formalizes the (σ, ρ) workload characterization. We then discuss in detail how to include the information captured in a (σ, ρ) workload characterization in an actor component and prove the temporal conservativeness of this actor component with respect to the corresponding task. After that we show how to use the (σ, ρ) workload characterization in combination with the temporal analysis of tasks scheduled with a run-time scheduler. This section is concluded with a method to transform execution time information of a finite number of consecutive executions to a (σ, ρ) workload characterization which contains execution time information for every number of consecutive executions.

4.3.1 FORMALIZING THE (σ, ρ) WORKLOAD CHARACTERIZATION

The (σ, ρ) workload characterization contains execution time information of consecutive executions of a task. The ρ parameter is required to be an upper bound on the average execution time of an infinite number of executions of a task. The σ parameter must account for the maximum possible deviation from this average execution time. The (σ, ρ) workload characterization forms an upper bound on the maximum sum of execution times of consecutive executions of a task.

The upper bound defined by the (σ, ρ) workload characterization needs to hold for every number of consecutive executions and the two parameters should be chosen such that Equation (4.1) holds.

$$\forall_{k,i} \ k \leq i : \sum_{j=k}^i x(j) \leq \omega(i - k + 1) \leq \hat{\omega}(i - k + 1) \quad (4.1)$$

With $x(j)$ the execution time of execution j , $\omega(n)$ the worst-case cumulative execution time and the function $\hat{\omega}$ as follows:

$$\hat{\omega}(n) = \sigma + (n - 1) \cdot \rho \quad (4.2)$$

This means that $\hat{\omega}(n)$ should be an upper bound on the sum of execution times of any sequence of n consecutive task executions. In particular, $\hat{\omega}(1)$ should be at least as large as the WCET of the task. Note that if we choose $\sigma = \rho = WCET$, the original constant workload characterization is obtained.

Figure 4.6: The conformance relation between a task τ in a task graph and an actor component v in a dataflow model.

Example 4.1:

Consider a task which has an alternating execution time of 8 and 4 time units. The average execution time of an infinite number of executions equals 6 time units. When we select σ equal to 8 and ρ equal to 6, Equation (4.1) holds.

Using the definition of the (σ, ρ) workload characterization we can prove the conservativeness of the conformance relation between a task graph with (σ, ρ) workload characterizations for the tasks and a dataflow model which uses the σ and ρ parameters.

4.3.2 CONSERVATIVE DATAFLOW ANALYSIS

Figure 4.6 shows the conformance relation between a task and an actor component. Let $e(i)$ from this figure be the external enabling time of execution i of the task, i.e., the time at which the required number of containers is present on the adjacent buffers. The actor component is externally enabled at time $e'(i)$ which is the time at which the required tokens, corresponding to the containers in the task graph, are present on the adjacent queues. Let $f(i)$ be the time at which execution i of the task finishes, and we use $f'(i)$ as the finish time of the actor component. We assume that tasks execute self-timed, i.e., they start as soon as they are enabled. Next to this, we define task executions such that different executions of the same task do not overlap in time. Furthermore, we will use $x(i)$ for the execution time of execution i of the task.

As shown in Chapter 2, when proving that an actor component is temporally conservative to the corresponding task it is sufficient to prove that when the enabling times of the dataflow model are later than or equal to the enabling times of the corresponding task executions, also the finish time of the firings are later than or equal to the finish times of the corresponding task executions:

$$\forall_{i \in \mathbb{N}} : e(i) \sqsubseteq e'(i) \implies f(i) \sqsubseteq f'(i) \quad (4.3)$$

Furthermore, given that (4.3) holds for every component in the dataflow model, it has been proven in Chapter 2 that the results concerning the temporal behavior of the dataflow model are conservative (pessimistic) to the task graph.

We now first introduce the notion of a consecutive execution with which we can derive an upper bound on the finish time of execution i of a task τ . Afterwards, the actor component is presented for which it is proven that (4.3) holds. The derived upper bound on the finish time of execution i of task τ is used in this proof.

Definition 4.1:

Execution i of task τ is part of a consecutive execution that starts with execution k of τ if for all executions $k < j \leq i$ of τ holds that $e(j) \leq f(j-1)$.

Every execution i is part of a consecutive execution because it either belongs to a consecutive execution that starts with execution k or it is the first firing of a consecutive execution, and thus $i = k$. Therefore, the finish time of execution i can be defined as:

$$f(i) = e(k) + \sum_{j=k}^i x(j) \quad (4.4)$$

The information on the execution times as provided by the (σ, ρ) workload characterization of task τ can now be used to find an upper bound $f^u(i)$ on the finish time of execution i of τ . We substitute (4.1) and (4.2) in (4.4) and find the upper bound:

$$\begin{aligned} f(i) &\leq f^u(i) = e(k) + \hat{\omega}(i - k + 1) \\ &= e(k) + \sigma + (i - k) \cdot \rho \end{aligned} \quad (4.5)$$

Applying the bound $f^u(i)$ as given by (4.5) is difficult in general because the start of the consecutive execution cannot always be determined. We therefore rewrite $f^u(i)$ to an upper bound that does not depend on the knowledge of consecutive executions. This upper bound only uses the external enabling time of the current execution and the finish time of the previous execution and is presented in Theorem 4.1.

Theorem 4.1. *Given a (σ, ρ) characterization of task τ , the upper bound on the finish time of execution i of task τ is given by:*

$$f(i) \leq f^u(i) \leq \max(e(i) + \sigma - \rho, f^u(i-1)) + \rho \quad (4.6)$$

Proof. For any execution i of task τ we can distinguish two cases. Either $e(i) > f(i-1)$ which means that i is the first execution in a consecutive execution, or $e(i) \leq f(i-1)$.

Case $e(i) > f(i-1)$: when $e(i) > f(i-1)$ we have that execution i starts a new consecutive execution and thus $k = i$. Filling in k in Equation 4.5 gives $f^u(i) = e(i) + \sigma + (i - i) \cdot \rho = e(i) + \sigma$.

Case $e(i) \leq f(i-1)$: if $e(i) \leq f(i-1)$ then execution i starts as soon as the previous execution of the task is finished. Execution i belongs to the same

Figure 4.7: Conservative actor component for (σ, ρ) characterization.

consecutive execution as execution $i - 1$ of task τ . The maximum time between two finishes of task τ can then be found by subtracting $f^u(i - 1)$ from $f^u(i)$ which are both given by Equation 4.5:

$$\begin{aligned} f^u(i) - f^u(i - 1) &= e(k) + \sigma + (i - k) \cdot \rho - (e(k) + \sigma + ((i - 1) - k) \cdot \rho) \\ &= e(k) - e(k) + \sigma - \sigma + (i - k) \cdot \rho - ((i - 1) - k) \cdot \rho \\ &= \rho \end{aligned}$$

We thus have $f^u(i) = f^u(i - 1) + \rho$.

Combining $f^u(i)$ as derived in the two cases results in the conservative max expression $f^u(i) \leq \max(e(i) + \sigma, f^u(i - 1) + \rho)$ which can be rewritten to Equation (4.6). $\square$

Figure 4.7 shows the actor component with which we model the (σ, ρ) workload characterization. It consists of one actor, v_0 , without a self-edge and a firing duration of $\sigma - \rho$ and one actor with self-edge with a firing duration of ρ , actor v_1 . Note that this actor component can consume and produce an arbitrary amount of tokens. It can thus for example be used as a component in an SDF graph or in a VRDF graph.

For this actor component we can derive the following max-expression by making use of the max-plus semantics [BCOQ92] of dataflow actors:

$$f'(i) = \max(e'(i) + \sigma - \rho, f'(i - 1)) + \rho \quad (4.7)$$

We now show that the actor component as shown in Figure 4.7 is conservative to its corresponding task τ .

Lemma 4.1. *Given the (σ, ρ) workload characterization of a task τ , Equation (4.3) holds for the relation shown in Figure 4.6 if the actor component is chosen as shown in Figure 4.7.*

Proof. With (4.6) and (4.7) and assuming $f^u(-1) \leq f'(-1)$ we can prove with induction on the execution number that $e\langle i \rangle \sqsubseteq e'\langle i \rangle \implies f^u\langle i \rangle \sqsubseteq f'\langle i \rangle$ holds for every execution i of task τ . This is sufficient to prove Equation (4.3) because $\forall_i : f(i) \leq f^u(i)$ holds.

Base: When $e(0) \leq e'(0)$ and because $f^u(-1) \leq f'(-1)$ we have that:

$$\begin{aligned} \max(e(0) + \sigma - \rho, f^u(-1)) + \rho &\leq \max(e'(0) + \sigma - \rho, f'(-1)) + \rho \\ f^u(0) &\leq f'(0) \end{aligned}$$

Therefore, we have $e\langle 0 \rangle \sqsubseteq e'\langle 0 \rangle \implies f^u\langle 0 \rangle \sqsubseteq f'\langle 0 \rangle$.

Step: Given that $e\langle i-1 \rangle \sqsubseteq e'\langle i-1 \rangle \implies f^u\langle i-1 \rangle \sqsubseteq f'\langle i-1 \rangle$ holds we need to show that $e\langle i \rangle \sqsubseteq e'\langle i \rangle \implies f^u\langle i \rangle \sqsubseteq f'\langle i \rangle$ holds.

When $e\langle i \rangle \sqsubseteq e'\langle i \rangle$ we also have $e\langle i-1 \rangle \sqsubseteq e'\langle i-1 \rangle$ and given the induction hypothesis we can conclude $f^u\langle i-1 \rangle \sqsubseteq f'\langle i-1 \rangle$. We have that $f^u\langle i \rangle \sqsubseteq f'\langle i \rangle$ is equivalent to $f^u\langle i-1 \rangle \sqsubseteq f'\langle i-1 \rangle \wedge f^u(i) \leq f'(i)$ which means that we only have to prove $f^u(i) \leq f'(i)$.

Given $e\langle i \rangle \sqsubseteq e'\langle i \rangle$ and $f^u\langle i-1 \rangle \sqsubseteq f'\langle i-1 \rangle$ we conclude $e(i) \leq e'(i)$ and $f^u(i-1) \leq f'(i-1)$ and thus:

$$\begin{aligned} \max(e(i) + \sigma - \rho, f^u(i-1)) + \rho &\leq \max(e'(i) + \sigma - \rho, f'(i-1)) + \rho \\ f^u(i) &\leq f'(i) \end{aligned}$$

□

4.3.3 RUN-TIME SCHEDULING

In [WBS07b, WBS09, LMC12] it is shown that run-time budget schedulers allow for an upper bound on the finish times of task executions. Budget schedulers guarantee a task a minimum amount of executing time S in a maximum time interval Q . The methods from [WBS07b, LMC12] assume one single worst-case execution time for a task and can not be combined with the (σ, ρ) characterization. On the other hand, the method from [WBS09] is applicable with upper bounds on execution times which are not constant over the different executions of a task and therefore allows us to combine it with the (σ, ρ) workload characterization.

We start this section with the upper bound on the finish times of task executions as derived in [WBS09]. We then combine this upper bound with the (σ, ρ) workload characterization. Lastly, the conservative actor component is derived with a similar method as in the previous section.

As shown in [WBS09] an upper bound on the finish times is $f^w(i)$ with:

$$f(i) \leq f^w(i) = e(k) + \sum_{j=k}^i x(j) + (Q - S) \left\lceil \frac{\sum_{j=k}^i x(j)}{S} \right\rceil \quad (4.8)$$

We can combine this upper bound with the knowledge of the (σ, ρ) workload characterization of task τ as given by Equations (4.1) and (4.2). We rewrite it to a new upper bound $f^x(i)$:

$$f(i) \leq f^x(i) = e(k) + \sigma + (i - k) \cdot \rho + (Q - S) \left\lceil \frac{\sigma + (i - k) \cdot \rho}{S} \right\rceil \quad (4.9)$$

Again, we provide an upper bound on the finish times which does not depend on the knowledge of consecutive executions:

Theorem 4.2. *For every scheduler that guarantees a task τ , a minimum amount of time S in every interval of time Q and given a (σ, ρ) workload characterization for τ , an upper bound on the finish time of execution i is given by:*

$$f(i) \leq f^x(i) \leq \max \left(e(i) + (Q - S) + \frac{Q \cdot (\sigma - \rho)}{S}, f^x(i - 1) \right) + \frac{Q \cdot \rho}{S} \quad (4.10)$$

Proof. As in [WBS09], we conservatively substitute $\lceil x \rceil$ by $x + 1$ in (4.9) such that:

$$\begin{aligned} f^x(i) &\leq e(k) + \sigma + (i - k) \cdot \rho + (Q - S) \left(\frac{\sigma + (i - k) \cdot \rho}{S} + 1 \right) \\ &= e(k) + Q - S + \frac{Q \cdot (\sigma + (i - k) \cdot \rho)}{S} \end{aligned} \quad (4.11)$$

Similar to the proof of Theorem 4.1 we distinguish two cases for an execution i of task τ .

The first case we distinguish is when execution i starts a new consecutive execution. Then $e(k) > f(i - 1)$ holds and we have $i = k$. Given Equation 4.11 and using $i = k$ we have:

$$f^x(i) \leq e(i) + Q - S + \frac{Q \cdot \sigma}{S}$$

In the second case, case $e(k) \leq f(i - 1)$, we have that execution i belongs to the same consecutive execution as execution $i - 1$. The difference between $f^x(i)$ and $f^x(i - 1)$ is then equal to $f^x(i) - f^x(i - 1) = \frac{Q \cdot \rho}{S}$ and thus:

$$f^x(i) \leq f^x(i - 1) + \frac{Q \cdot \rho}{S}$$

When we combine these two cases we get:

$$\begin{aligned} f^x(i) &\leq \max \left(e(i) + Q - S + \frac{Q \cdot \sigma}{S}, f^x(i - 1) + \frac{Q \cdot \rho}{S} \right) \\ &= \max \left(e(i) + Q - S + \frac{Q \cdot (\sigma - \rho)}{S}, f^x(i - 1) \right) + \frac{Q \cdot \rho}{S} \end{aligned}$$

□

The actor component which combines budget scheduler settings with a (σ, ρ) workload characterization is shown in Figure 4.8. The max-plus expression corresponding to this actor component is as follows:

$$f'(i) = \max \left(e'(i) + (Q - S) + \frac{Q \cdot (\sigma - \rho)}{S}, f'(i - 1) \right) + \frac{Q \cdot \rho}{S} \quad (4.12)$$

We can now show that the actor component as shown in Figure 4.8 is conservative to its corresponding task τ .

Figure 4.8: A conservative actor component for the (σ, ρ) characterization including the effect of budget scheduler settings.

Lemma 4.2. *Given the (σ, ρ) workload characterization of a task τ and budget scheduler settings Q and S , Equation (4.3) holds for the relation shown in Figure 4.6 if the actor component is chosen as shown in Figure 4.8.*

Proof. Analogous to the proof for Lemma 4.1 with Equations 4.10 and 4.12 instead of Equations 4.6 and 4.7 respectively. $\square$

4.3.4 TRANSFORMATION OF FINITE WORKLOAD CHARACTERIZATIONS

In some cases, only the workload characterization of a finite number of N consecutive executions is known. In this section we first define a method for transforming such information on a finite number of consecutive executions to a (σ, ρ) workload characterization, which holds for every number of consecutive executions. The second paragraph of this section improves this transformation for the case the WCET of the task is known. The idea behind the transformation to a (σ, ρ) workload characterization is the fact that every N consecutive executions can be individually bounded by the known finite workload characterization.

Transformation to (σ, ρ) Workload Characterization

An upper bound on the cumulative execution time for up to N consecutive executions can be defined if the workload characterization of a finite number of consecutive executions is known. This upper bound is formalized as follows:

$$\forall_{i,n} \ 1 \leq n \leq N : \sum_{j=i}^{i+n-1} x(j) \leq \omega^N(n) \quad \text{with} \quad \omega^N(n) = \sigma^N + (n-1) \cdot \rho^N \quad (4.13)$$

Where ρ^N is an upper bound on the average worst-case execution time of every N consecutive executions and σ^N accounts for the maximum possible deviation from this average.

Because this finite workload characterization holds for every $n \leq N$ consecutive executions, we can transform this characterization to a (σ, ρ) workload characterization that also holds for $n > N$. Equivalent to Equation (4.1) we have

that the following equation needs to hold for the (σ, ρ) workload characterization:

$$\forall_{i,n} \ 1 \leq n : \sum_{j=i}^{i+n-1} x(j) \leq \sigma + (n-1) \cdot \rho \quad (4.14)$$

To derive a (σ, ρ) workload characterization that holds for every number of consecutive executions we split the consecutive executions into groups. Every number of consecutive executions, say n , can be split into $k = \lfloor \frac{n-1}{N} \rfloor$ groups of exactly N executions plus a group containing the remainder of the n executions: $n - k \cdot N$ executions. We have that $n-1 = k \cdot N + (n-1) \bmod N$. Therefore, $n - k \cdot N = (n-1) \bmod N + 1$ and thus $1 \leq n - k \cdot N \leq N$. The sum of the execution times of n executions can be written as:

$$\sum_{j=i}^{i+n-1} x(j) = \underbrace{\sum_{j=i}^{i+N-1} x(j) + \dots + \sum_{j=i+(k-1) \cdot N}^{i+k \cdot N-1} x(j)}_{k \text{ groups of } N \text{ executions}} + \sum_{j=i+k \cdot N}^{i+n-1} x(j) \quad (4.15)$$

Because the finite workload characterization given in Equation (4.13) holds for every $n \leq N$ consecutive executions, we can use it to bound the execution times of the groups of consecutive executions of Equation (4.15):

$$\sum_{j=i}^{i+n-1} x(j) \leq \sum_{j=i+k \cdot N}^{i+n-1} x(j) + k \cdot \omega^N(N) \quad (4.16)$$

$$\leq \omega^N(n - k \cdot N) + k \cdot \omega^N(N) \quad (4.17)$$

$$= \sigma^N + (n - k \cdot N - 1) \cdot \rho^N + k \cdot (\sigma^N + (N-1) \cdot \rho^N) \quad (4.18)$$

$$= \sigma^N + (n-1) \cdot \rho^N + k \cdot (\sigma^N - \rho^N) \quad (4.19)$$

By using $k \leq \frac{n-1}{N}$ we have

$$\leq \sigma^N + (n-1) \cdot \rho^N + (n-1) \cdot \left(\frac{\sigma^N - \rho^N}{N} \right) \quad (4.20)$$

$$= \sigma^N + (n-1) \cdot \left(\rho^N + \frac{\sigma^N - \rho^N}{N} \right) \quad (4.21)$$

$$= \sigma^N + (n-1) \cdot \left(\frac{\sigma^N + (N-1) \cdot \rho^N}{N} \right) \quad (4.22)$$

From this we conclude that if we choose $\sigma = \sigma^N$ and $\rho = \frac{\sigma^N + (N-1) \cdot \rho^N}{N}$, Equation (4.14) holds and we thus have derived a valid (σ, ρ) workload characterization.

Example 4.2:

Assume we know from the worst-case execution time trace of a task that the worst-case cumulative execution time of up to $N = 4$ consecutive executions can be

Figure 4.9: Example transformation of a finite workload characterization to a (σ, ρ) characterization that holds for every number of consecutive executions.

bounded by $8 + (n - 1) \cdot 4$ with $n \leq N$ the number of consecutive executions. An example of such a worst-case trace is shown in Figure 4.9. The finite workload characterization can be transformed in a (σ, ρ) workload characterization with $\sigma = 8$ and $\rho = 5$ which is illustrated by the dashed line. Because the first 4 consecutive executions are determined using the worst-case trace, we know that the next 4 consecutive executions cannot take more time than these first 4 consecutive executions. This is illustrated in Figure 4.9 with the dotted lines. As can be seen in Figure 4.9, the bound corresponding to the computed (σ, ρ) workload characterization forms an upper bound to the first 4 consecutive executions as well as the repeated executions.

Improved Transformation by Using the WCET

The transformation defined in the previous section can be improved if the WCET of the task is known. The value for σ derived in the previous section might be too pessimistic if σ^N is larger than the WCET. We show in this section that we can decrease, and thus improve, the value of σ such that it is higher than the WCET and such that is still conservative to the finite workload characterization for $n \geq 2$. We choose ρ equal to the value derived in the previous section, $\rho = \frac{\sigma^N + (N-1) \cdot \rho^N}{N}$, and only derive a new value for σ . Note that we assume a value of N larger or equal to two.

Again a value for σ needs to be found such that Equation (4.14) holds. We

Figure 4.10: Transformation of a finite workload characterization to an (improved) (σ, ρ) characterization.

apply case distinction on n to find this value. We consider two cases: $n = k \cdot N + 1$ and $n = k \cdot N + a$ with $2 \leq a \leq N$. We derive upper bounds for both cases and use the maximum of these two upper bounds as the (σ, ρ) workload characterization.

If $n = k \cdot N + 1$ we can split the sequence of executions in k groups of N executions plus one extra execution. For these values of n we have $k = \frac{n-1}{N}$. Because we know that the execution time of every execution is less or equal to the WCET we get:

$$\begin{aligned}
 \sum_{j=i}^{i+n-1} x(j) &\leq WCET + k \cdot \omega^N(N) \\
 &= WCET + \frac{n-1}{N} \cdot \omega^N(N) \\
 &= WCET + (n-1) \cdot \rho
 \end{aligned}$$

In the other case we have $n = k \cdot N + a$ with $2 \leq a \leq N$. In this case we have $k = \frac{n-a}{N}$ and we use the fact that $(a-2) \geq 0$. Furthermore, we know that $\rho \geq \rho^N$

and thus $\rho^N - \rho \leq 0$. With this information we derive the following upper bound:

$$\begin{aligned}
 \sum_{j=i}^{i+n-1} x(j) &\leq \omega^N(a) + k \cdot \omega^N(N) \\
 &= \sigma^N + (a-1) \cdot \rho^N + k \cdot (\sigma^N + (N-1) \cdot \rho^N) \\
 \text{With } \rho &= \frac{\sigma^N + (N-1) \cdot \rho^N}{N} \text{ and } k = \frac{n-a}{N} \text{ we have} \\
 &= \sigma^N + (a-1) \cdot \rho^N + (n-a) \cdot \rho \\
 &= \sigma^N + (n-1) \cdot \rho + (a-1) \cdot (\rho^N - \rho) \\
 &= \sigma^N + \rho^N - \rho + (n-1) \cdot \rho + (a-2) \cdot (\rho^N - \rho) \\
 &\leq \sigma^N + \rho^N - \rho + (n-1) \cdot \rho
 \end{aligned}$$

We combine these two cases by choosing $\sigma = \max(WCET, \sigma^N + \rho^N - \rho)$. Together with $\rho = \frac{\sigma^N + (N-1) \cdot \rho^N}{N}$ we know that both cases are bounded and that Equation (4.14) holds for every n . This is an improvement of the (σ, ρ) workload characterization derived in the previous paragraph because $\sigma^N \geq WCET$ and $\rho \geq \rho^N$.

Example 4.3:

In Figure 4.10, the worst-case trace of 4 consecutive executions of a task is shown which can be bounded by $17 + (n-1) \cdot 1$ with $n \leq 4$. We furthermore know that the WCET of this task is equal to 10. The dashed line shows the original (σ, ρ) workload characterization as derived in the previous paragraph with $\sigma = 17$ and $\rho = 5$. As can be seen in the figure, this (σ, ρ) workload characterization is too conservative since it can be shifted down and still form an upper bound on all consecutive executions. The dash-dotted line illustrates the improved (σ, ρ) characterization with $\sigma = \max(10, 17 + 1 - 5) = 13$ and $\rho = 5$. It is still an upper bound for all the consecutive executions in the worst-case trace and improves the original (σ, ρ) workload characterization.

4.4 CYCLIC TEMPORAL BEHAVIOR

CSDF graphs can be used to model the temporal behavior of tasks with explicit cyclic temporal behavior. This adds an additional degree of freedom to the scheduling of tasks which means that both a larger class of applications can be modeled and that the behavior of applications can be modeled in more detail [BELP96, PPL95]. In this section we combine the modeling of this cyclic behavior with information captured in a (σ, ρ) workload characterization. This (σ, ρ) workload characterization does contain different (σ, ρ) information for each phase of the cyclo-static period.

As shown in Chapter 2, CSDF actors have a cyclo-static period consisting of phases. The cyclic temporal behavior of a task can be bounded by distinguishing

the phases of the cyclo-static period:

$$\forall_{k,i} \ k \leq i : \sum_{j=k}^i x(j) \leq \sum_{j=k}^i \hat{x}_{j \% M} \quad (4.23)$$

With $\hat{x}_m$ the worst-case execution time of phase m of the cyclo-static period, $\%$ the modulo operation and M the number of phases of the cyclo-static period of the CSDF actor.

Instead of one worst-case execution time of a phase we have (σ, ρ) information for each phase. We have ρ_m the average duration of phase m and $\sigma - \rho_m$ the maximum deviation from this average over all executions of all phases. Instead of (4.23) we have:

$$\forall_{k,i} \ k \leq i : \sum_{j=k}^i x(j) \leq \sigma + \sum_{j=k+1}^i \rho_{j \% M} \quad (4.24)$$

Note that Equation 4.24 is a generalization of the (σ, ρ) characterization shown in Equations (4.1) and (4.2) for non cyclo-static dataflow graphs.

This (σ, ρ) characterization can be used to define an upper bound, $f^z(i)$, on the finish of execution i by substituting (4.24) in (4.4):

$$f(i) \leq f^z(i) = e(k) + \sigma + \sum_{j=k+1}^i \rho_{j \% M} \quad (4.25)$$

Similar to the method used in Section 4.3.2 we can derive an upper bound on the finish times of a task which does not depend on the knowledge of consecutive executions:

Theorem 4.3. *Given a (σ, ρ) characterization of a task τ that has cyclic temporal behavior, the upper bound on the finish time of execution i of task τ is given by:*

$$f(i) \leq f^z(i) \leq \max \left(e(i) + \sigma - \min_{0 \leq m < M} (\rho_{m \% M}), f^z(i-1) \right) + \rho_{i \% M} \quad (4.26)$$

Proof. We again prove this with case distinction on the start of a consecutive execution.

Case $e(k) > f(i-1)$: execution i starts a new consecutive and thus $i = k$. Using Equation 4.25 we have: $f^z(i) = e(i) + \sigma$

Case $e(k) \leq f(i-1)$: execution i belongs to the same consecutive execution as execution $i-1$ and thus by using Equation 4.25 we have: $f^z(i) - f^z(i-1) = \rho_{i \% M}$ and thus $f^z(i) = f^z(i-1) + \rho_{i \% M}$.

When we combine these two cases we get:

$$\begin{aligned} f^z(i) &\leq \max(e(i) + \sigma, f^z(i-1) + \rho_{i \% M}) \\ f^z(i) &= \max(e(i) + \sigma - \rho_{i \% M}, f^z(i-1)) + \rho_{i \% M} \\ f^z(i) &\leq \max \left(e(i) + \sigma - \min_{0 \leq m < M} (\rho_{m \% M}), f^z(i-1) \right) + \rho_{i \% M} \end{aligned} \quad (4.27)$$

□

Figure 4.11: Conservative actor component for (σ, ρ) characterization in a CSDF graph.

Figure 4.12: Task graph of a DVB-T application.

Note that the left term of the max-expression is rewritten such that it is constant. This is because, as discussed in Chapter 2, we require that CSDF actors without a self-edge have a constant firing duration. The left term of the max-expression corresponds to the left actor of the actor component of Figure 4.11 which does not have a self-edge.

With this result we can show, analogous to the proof in Section 4.3.2, that the actor component shown in Figure 4.11 models the temporal behavior of a task with cyclic temporal behavior in combination with a (σ, ρ) workload characterization conservatively. Note that actor v_5 in this figure contains a cyclo-static response time denoted by $\rho_{i \% M}$ with M the number of phases of CSDF actor v_5 .

4.5 CASE STUDY

In this section we show how the (σ, ρ) workload characterization can be used to improve the accuracy of the temporal analysis of a DVB-T receiver application. Next to that we give an example of a car-radio application that shows the effect of a latency constraint on the temporal analysis results that can be obtained using a (σ, ρ) workload characterization. We also use the (σ, ρ) workload characterization to improve the temporal results of a MP3 playback application which has a data-dependent behavior.

4.5.1 DVB-T RECEIVER APPLICATION

Figure 4.12 shows the task graph of a DVB-T receiver application. The tasks in the task graph communicate via finite FIFO buffers. The demodulation task, *demod.*, and the deinterleaving task, *deint.*, process and communicate complete symbols

Figure 4.13: Software defined radio architecture with a shared bus, a shared SRAM, and processors with instruction and data caches.

F which consist of 8192 samples. The other tasks process and communicate per sample which is denoted by S .

Figure 4.13 illustrates the MPSoC architecture for software defined radio applications on which the DVB-T receiver application is executed. It contains an analogue front-end (AFE), a digital front-end bandpass filter (DFE), a Digital Signal Processor (DSP) with instruction and data caches for demodulation, an accelerator for deinterleaving and error correction (viterbi), a shared bus, a shared SRAM, and a microprocessor (μP) for control and interfacing with peripherals. The caches, the shared bus, and the shared SRAM memory port are hardware elements that can cause a significant difference between the WCET and the average execution time of a task executing for example on the DSP.

For this case study we consider the demodulation task because this task will be executed on the DSP while the other tasks are executed on dedicated hardware (front-ends and the accelerator) and have a fixed execution time.

The demodulation task has variability in its execution time due to sharing of the used memory port, control flow in the task and caches in the DSP. Despite this variability, the DVB-T receiver has a strict throughput constraint. The demodulation task should, on average, process one frame per $952\mu s$. Traditionally the WCET, which includes the maximum variability, should be used to give guarantees on the throughput. Typically sufficient buffer sizes need to be determined to meet the throughput constraint of 1 frame per $952\mu s$. If the WCET of the demodulation task is larger than $952\mu s$, analysis methods that use only this WCET information would indicate that the throughput constraint cannot be met, independent of the buffer sizes.

The throughput constraint can be met if we for example know that the average execution time of every eight consecutive executions is less or equal to $952\mu s$ and that of every eight executions, maximally four consecutive executions have a higher execution time than $952\mu s$. The execution time of these longer executions is maximally 15% higher than $952\mu s$.

With this information we can define a (σ, ρ) workload characterization. According to the information, a ρ equal to $952\mu s$ forms an upper bound on the cumulative execution because the maximum cumulative execution time of every

Figure 4.14: Dataflow model of the *DFE*, *demod.* and *deint.* tasks. The *demod.* task is modeled using its (σ, ρ) workload characterization.

8 consecutive executions equals $8 \cdot 952\mu s$. Maximally four per eight executions take longer than $952\mu s$ and we know that the maximum possible deviation from the average execution time is caused by these four executions taking 15% longer than $952\mu s$. The maximum deviation is thus 60% of $952\mu s$. After eight consecutive executions, the cumulative execution time is again smaller or equal to the average execution time which means that σ equal to $1.6 \cdot 952\mu s$ and ρ equal to $952\mu s$ forms an upper bound on the cumulative execution time of the demodulation task.

With these values for σ and ρ we can compute the buffer sizes which do allow the demodulation task to process one frame per $952\mu s$ on average. Figure 4.14 shows the dataflow model of the (σ, ρ) workload characterization corresponding to the demodulation task. It also shows the adjacent tasks, *DFE* and *deint.*, which have a constant execution time of $\frac{952}{8192}\mu s$ and $952\mu s$ respectively.

The required number of initial tokens such that the required throughput is met can be calculated with buffer sizing methods such as [SGB06]. Calculation of this required number of initial tokens for this dataflow model shows that $2.6 \cdot 8192$ tokens are required for δ_0 and $3 \cdot 8192$ tokens are required for δ_1 .

4.5.2 AUDIO ECHO CANCELLATION APPLICATION

In this section we model the temporal behavior of a car-radio application. The application is taken from [WBS07] in which a SDF model of the application is presented. We take a part of the model to illustrate the effect of a latency constraint on the temporal analysis results that can be obtained using a (σ, ρ) workload characterization.

Figure 4.15 shows the block diagram of the application. A Bluetooth device (*BT*) is used to make a phone call. Simultaneously, an MP3 file is playing at a lower volume. Next to that, the application contains an Audio Echo Cancellation (*AEC*)

Figure 4.15: Block diagram of an audio echo cancellation application.

algorithm to prevent the howling effect (feedback loop between microphone and speaker) and to prevent the sound from the speaker to be transmitted via the Bluetooth device. The latency between the microphone and the Bluetooth device should be small to prevent annoying delays in the speech.

The application contains three main streams of data. The first stream consists of a MP3 decoder which decodes songs provided by a block reader (BR). The second stream receives the speech of the caller via the Bluetooth (BT) device. This speech is then mixed with the stream of music and transmitted using a speaker. The last stream receives the speech of the user via a microphone. The sound from the speaker is removed from this speech using the audio echo cancellation task (AEC) and eventually transmitted to the caller via the Bluetooth device (BT).

We focus on the AEC task in this case-study. We use a (σ, ρ) workload characterization to model its varying temporal behavior which occurs due to sharing of resources. Figure 4.16 shows the SDF graph of the pipeline from microphone via the ADC, AEC and OUT tasks to the Bluetooth device. The ADC task executes strictly periodic at a frequency of 8kHz. The firing durations of the AEC and OUT tasks are equal to $\frac{1}{8}$ ms. The AEC task is modeled with a (σ, ρ) workload characterization where AEC_0 and AEC_1 have firing durations equal to $\sigma - \rho$ and ρ respectively. In [WBS07] the worst-case response time of the AEC task is determined to be 9.091ms. The AEC task also has an algorithmic delay of 6ms caused by its 48 taps filter. In this case-study we assume that the worst-case response time of the AEC task is measured and that it can be sporadically higher than 9.091ms. We assume that this behavior can be characterized using values for ρ and σ equal to 9.091 and $1.2 \cdot 9.091 = 10.909$.

Using this information one can compute the required buffer sizes δ_0 and δ_1 to meet the throughput constraint of 8kHz. However, the dataflow model of Figure 4.16 also contains a latency constraint which is modeled with the bottom edge. For a correct user experience, we enforce a maximum latency of 26ms between input and output. The algorithmic delay of the AEC task is subtracted

Figure 4.16: SDF model of a part of an audio echo cancellation application.

from this latency and we enforce the remaining 20ms by creating a cycle via the bottom edge. The *ADC* task operates at a frequency equal to 8kHz which means that its period is equal to $\frac{1}{8kHz} = 0.125\text{ms}$. We have that 20ms is a multiple of this average period and therefore we can enforce the maximum latency constraint by means of initial tokens on the edge from the *OUT* actor to the *ADC* actor. By adding this edge a cycle is created from the *ADC* actor to the *OUT* actor and back. The maximum latency of 20ms is enforced by including this edge in the model with in total $\frac{20}{0.125} = 160$ initial tokens on the edge. The number of tokens in the created cycle is then also equal to 160 which means that the finish of a firing of the *OUT* actor may be 160 firings later than the start of a firing of the *ADC* actor. The schedules of the *OUT* and *ADC* actors have a period equal to $\frac{1}{8}\text{ms}$ and their firing durations are also equal to $\frac{1}{8}\text{ms}$ which means that they fire strictly periodic. Therefore, the 160 initial tokens on the edge from the *OUT* actor to the *ADC* actor model that the finish of actor *OUT* may be maximally $160 \cdot \frac{1}{8} = 20\text{ms}$ later than the start of the corresponding firing of actor *ADC*.

When this latency constraint is taken into account the conclusion is that no buffer sizes can be found that adhere to all the constraints in the dataflow model. This is because the latency constraint is too tight to allow for the compensation of executions with a larger execution time than the average. When we relax the maximum latency constraint to for example 28ms, the throughput constraint can be met with this (σ, ρ) workload characterization.

4.5.3 MP3 PLAYBACK APPLICATION

This section shows how the (σ, ρ) workload characterization can be used in combination with applications containing data-dependent behavior. To illustrate this, we use the MP3 playback application of [WBS08b]. This application can be modeled with the VRDF model shown in Figure 4.17. We model the MP3 task using a (σ, ρ) actor component because we assume it has a varying execution

Figure 4.17: Data-dependent VRDF model of a MP3 playback application.

time. Actor $MP3_0$ has a firing duration equal to $\sigma - \rho$ and $MP3_1$ has a firing duration of ρ .

The amount of tokens consumed per execution of the MP3 task is dependent on the input and can vary during execution. Each firing, n tokens are consumed and produced. The only information we know about n is that n is less or equal to 960 for an MP3 file with a bit-rate of 320 kbit per second. Furthermore, we have a throughput constraint because the application requires that the DAC executes strictly periodic at a frequency of 44.1kHz.

In [WBS08b] the maximum response times of the different tasks are computed given this throughput constraint. The conclusion is that the throughput of the application can only be met if the response time of the MP3 task is less or equal to 24ms. However, the model as shown in Figure 4.17 also allows for worst-case response times larger than 24ms as long as ρ is less or equal to 24ms.

Because the model shown in Figure 4.17 is a VRDF graph, the method presented in [WBS08b] can be used to find sufficient buffer capacities that given a (σ, ρ) workload characterization meet the throughput constraint. If we choose ρ equal to 24ms, σ equal to $1.5 \cdot 24 = 36$ ms and all the other firing durations equal to the maximum allowed firing duration, we obtain the following sufficient buffer capacities from the analysis: $\delta_1 = 6491$, $\delta_2 = 3836$ and $\delta_3 = 882$. With a one-actor model using this higher response time for the MP3 task, the throughput constraint cannot be met.

4.6 CONCLUSION

This chapter presented a dataflow modeling technique which exploits knowledge about the maximum cumulative execution time of consecutive executions. To exploit this knowledge in a dataflow model, the (σ, ρ) workload characterization has been introduced. It is shown that this (σ, ρ) workload characterization can be temporal conservatively modeled with an actor component consisting of two

actors. One actor represents the maximum rate and the two actors together model the maximum latency.

We furthermore showed that the (σ, ρ) workload characterization can be combined with the temporal analysis of scheduler settings of a budget scheduler. The (σ, ρ) workload characterization can also be used in combination with cyclic temporal behavior of tasks. Next to that, a technique is presented which can derive a (σ, ρ) workload characterization from temporal information on a finite number of consecutive executions.

In the case study we have illustrated that the accuracy of the temporal analysis of a DVB-T application can be improved with the (σ, ρ) workload characterization. It is shown that temporal analysis methods can use an upper bound on the average execution time over a finite number of executions for calculating the throughput. Existing buffer sizing techniques have been used to compute the buffer sizes that are required to cope with the variability in the execution time of the tasks. The (σ, ρ) characterization also makes explicit how much additional buffer locations are required to compensate for the deviation from the average execution time of a task.

Dataflow Analysis for Non-Starvation-Free Schedulers

ABSTRACT – *This chapter presents a temporal analysis flow for SDF graphs which is applicable for systems with non-starvation-free schedulers such as static priority preemptive schedulers. The analysis flow uses an enabling rate characterization to calculate response times. This enabling rate characterization is determined using multi-dimensional periodic schedules.*

The presented approach is applicable for arbitrary (cyclic) graph topologies and can take buffer capacity constraints into account during analysis. Also cyclic resource dependencies can be analyzed. The presented analysis flow is the first approach that considers arbitrary SDF graph topologies in combination with cyclic resource dependencies that are caused by non-starvation-free schedulers. Next to that, the presented analysis method broadens the applicability of dataflow analysis techniques to systems with non-starvation-free schedulers.

For the temporal analysis of streaming applications using dataflow models, the current state of the art demands that all schedulers are run-time budget schedulers [BHM⁺05, WBS07b, WBS09]. Such budget schedulers (e.g. TDM) guarantee each task a minimum time budget in a maximum time interval. Therefore, by construction a bound is imposed on the maximum interference that tasks cause on other tasks sharing a resource.

This chapter shows that also systems with schedulers different from budget schedulers can be analyzed with timed SDF models. This is achieved by computing upper and lower bounds on the interference experienced by each task.

The material in this chapter is based on work presented in [HGWB13a, HGWB14a].

In a properly managed system these upper and lower bounds will not indefinitely diverge because of the strictly periodic source or sink of the real-time stream processing application. The considered class of schedulers is called non-starvation-free schedulers and is broader than budget schedulers. For such non-starvation-free schedulers (e.g. static priority (non-)preemptive), the interference from other tasks can only be bounded with knowledge about how often tasks are started, see Chapter 1.

Our contribution is a temporal analysis flow for SDF models with an arbitrary (cyclic) graph topology. Buffer capacity constraints can be taken into account during the analysis and sufficient buffer capacities can be determined afterwards. The analysis flow is based on enabling characterizations. An enabling characterization is a function that describes for each time interval the minimum and maximum number of possible task enablings. The enabling characterization which is used in this chapter allows to model bursts accurately. These bursts are caused by multi-rate behavior in an application and are modeled using SDF graphs. In order to temporally analyze the multi-rate behavior, an actor component consisting of two actors is required. The parameters of this two-actor component are computed differently and have a different meaning than the two-actor component of [WBS07b]. Furthermore, the class of non-starvation-free schedulers requires the use of fixed-point computation to determine these parameters.

The presented approach is applicable for single processor systems as well as multiprocessor systems. By using the actual data dependencies between tasks, a potentially more accurate analysis method can be defined for single processor systems than is possible when tasks are assumed to be independent.

The outline of this chapter is as follows. We first discuss related work in Section 5.1. The proposed temporal analysis flow is presented in Section 5.2 and the steps of this flow are detailed in Sections 5.3, 5.4 and 5.5. In Section 5.6 we define a conservative variant of the analysis flow which has a polynomial time-complexity. Section 5.7 presents a method to determine sufficient buffer sizes and the analysis flow and its properties are evaluated in Section 5.8. The analysis flow is used in two case-studies which are presented in Section 5.9 and we conclude this chapter in Section 5.10.

5.1 RELATED WORK

The focus of this thesis is on real-time stream processing applications of which the tasks are executed data-driven on MPSoCs. Data-driven execution of tasks allows to cope with varying execution times, i.e., shorter executions can compensate for long executions of a task. Such varying execution times can be captured in workload functions which can be included in temporal analysis flows [MKT04, QHE12, HGWB13b]. Non data-driven approaches, such as methods that are based on the periodic task model [BT13, BTV12, BS11], use strictly periodic schedules. Such strictly periodic schedules require the use of the WCETs of tasks and can therefore not use this additional knowledge about varying execution times. Next

to that, these strictly periodic schedules can not handle response times of tasks that are larger than the periods of the tasks.

The multi-rate behavior of stream processing applications can be accurately modeled using SDF models [LM87]. In [WBS07b, WBS09, LMC12] it has been shown how the effects of run-time schedulers that belong to the class of starvation-free schedulers can be included in SDF analysis models. As discussed in Chapter 1, for schedulers that belong to the starvation-free class it is possible to compute the worst-case response time of a task without having to know the execution rates of interfering tasks. Only the execution times of the interfering tasks have to be known. When non-starvation-free schedulers are used, conservative response times of tasks can only be found when both the execution times as well as the execution rates of tasks are known. The starvation-free schedulers are a subset of the non-starvation-free schedulers.

Compared to the analysis of starvation-free run-time schedulers with SDF analysis models, the generalization to non-starvation-free schedulers comes at the cost of a worst-case exponential computational complexity of the analysis algorithms. This increased complexity is caused by the requirement of fixed-point computation in our analysis flow. The accuracy of the analysis results can be improved by making use of knowledge about best-case execution times, which is not considered for the analysis of starvation-free run-time schedulers. Due to the required fixed-point computation in our analysis flow, buffer-sizing can only be performed after the analysis flow finishes in contrast to dataflow analysis methods for starvation-free schedulers, which can minimize buffer sizes by modifying the schedules [MBGS10, WBS07a]. We do present a conservative variant of the analysis flow which can analyze applications that use non-starvation-free schedules with a polynomial time-complexity. This conservative method can also be used to minimize buffer sizes but the used approximation can result in a lower accuracy of the obtained analysis results.

A number of existing approaches are related to the analysis method that we introduce in this chapter.

The approach from [FAS⁺14] considers systems with First-Come First-Served (FCFS) schedulers in combination with HSDF models. This approach considers a subclass of the systems we consider. FCFS schedulers belong to the class of non-starvation-free which is supported by our method. Next to that, only acyclic graphs are considered while we also support cyclic graphs and, furthermore, pipelined schedules are not considered in [FAS⁺14] which means that the source can only be triggered when the previous result is produced.

An iterative procedure of traffic characterization and response time calculation is used by the SymTA/S approach [HHJ⁺05]. However, this approach uses propagation of traffic instead of the computation of worst-case and best-case schedules which our method uses. Using traffic propagation can result in a low accuracy because the applied traffic characterization does not capture the correlation between different streams accurately [KJL13]. The dataflow analysis

techniques presented in this chapter, do not suffer from this disadvantage because the schedules that are used to compute the enabling characterization do capture the correlation of events. The use of best- and worst-case schedules in our solution is possible thanks to monotonicity of dataflow graphs, see Chapter 2. The enabling characterizations which we derive in this chapter are highly similar to the event models of the SymTA/S approach [HHJ⁺05].

In [SE09] the SymTA/S approach is extended by calculating more accurate temporal end-to-end properties. However, it relies on similar traffic models as the original approach and these traffic models cannot be determined for arbitrary cyclic graphs [Jer05].

The original MPA [WTVL06] approach, based on Real-Time Calculus [CKT03], also makes use of traffic propagation. However, MPA for cyclic HSDF graphs [TS09] solves the problems of traffic propagation by making use of a characterization of traffic in which the correlation between streams is captured by a characterization in the time-domain instead of the time-interval domain. In [TS09] also a schedulability check given a throughput constraint is presented. However, it only discusses (potentially cyclic) data dependencies and does not include cyclic resource dependencies. Cyclic resource dependencies are considered in [JPTY08] but the combination with cyclic data dependencies is not considered. This combination is difficult because it requires an accurate translation between a traffic characterization in the time-domain and in the time-interval domain.

The result from [TS09] can be generalized to SDF graphs by converting the SDF graph into an equivalent HSDF graph [LM87, SB00]. However, we see several issues with this approach. When an SDF actor is modeled using multiple HSDF actors, also multiple MPA components correspond to one task. Each of these components transfers remaining service to the next component but these service curves are in the time interval domain. As a result, the moment at which a resource is used by a component is lost. This leads to a severe accuracy reduction in the case of SDF graphs because the correlation between the usage of a resource by consecutive executions of a task, is lost. Next to that, modeling a task with multiple MPA components leads to a cyclic resource dependency. Each of the MPA components corresponding to a task transfer its remaining service to the next MPA component and the last MPA component transfers remaining service back to the first component. It is unclear how such a cyclic resource dependency can be analyzed in the MPA approach when there are also cyclic data dependencies.

Our method also uses the conversion to an equivalent HSDF graph. We use the schedules of the HSDF graph to determine the enabling characterizations of the tasks. The schedules of the HSDF actors are combined in such a way that the correlation between the resource usage of consecutive executions of a task is preserved. Next to that, we show in Section 5.9 that the derived enabling characterizations lead to better analysis results in the case of multi-rate behavior than can be obtained when a period and enabling jitter based enabling characterization is used as is done in [HHJ⁺05, HGWB13a].

Figure 5.1: Overview of the analysis flow.

5.2 ANALYSIS FLOW

Figure 5.1 shows the analysis flow that is used in this chapter. This section gives a brief overview of the different steps in the flow and indicates the section in which they are discussed.

The characteristics of an application are discussed in Section 5.3. The temporal constraints of such an application are included in this characterization by means of periodic sources. Given these characteristics, initial upper and lower bounds on the response times of tasks can be found using iterative fixed-point computation as is discussed in Section 5.4 (step 1).

The flow maintains a conformance relation between the given task graph and an SDF model. The conformance relation between the task graph and the SDF model ensures that the obtained analysis results also hold for the application. The conformance relation adheres to the rules specified for actor components in Chapter 2. The relation relies on the properties of the response times and on the derivation of the parameters (firing durations) of an actor component, as is presented in Section 5.5 (step 2). The SDF model is then translated into an equivalent HSDF model which is used to determine upper and lower bounds on the schedules of the SDF model. The calculated upper and lower bounds are used to derive an enabling characterization for each task which is also discussed in Section 5.5 (step 3). Initially, an optimistic enabling characterization is used which underestimates the amount of enablings.

Steps 1, 2 and 3 are repeated until the enabling characterizations no longer change and thus also the response times have converged. The analysis flow also stops when a violation of the temporal constraints is detected. Both the response

time analysis step (step 1) and the dataflow analysis step (step 3) can return infeasible which means that the analysis flow concludes that the application under the given mapping and given constraints is not schedulable.

Finally, if the converged response times are found, sufficient buffer sizes can be determined (step 5) such that the temporal results, obtained in the previous steps, are ensured. Buffer sizes are chosen such that the schedules from step 4 remain feasible. This buffer sizing step is presented in Section 5.7.

The complete analysis flow has a monotonic behavior. This monotonic behavior of the flow is guaranteed by the fact that an iteration of the analysis flow can never lead to an enabling characterization with less enablings. This is because the response time computation is monotonic in the enabling characterization and because of monotonicity of dataflow graphs which is explained in Section 5.5. As a result of this monotonicity, a conservative temporal analysis result is found after convergence of the response times.

5.3 APPLICATION CHARACTERISTICS

The characteristics of an application consists of the application described by one or more task graphs, and the mapping of the tasks of the application to the MPSoC (task to processor specification). Each task is mapped to one processor of the MPSoC. The tasks execute data-driven and are scheduled by the run-time scheduler of the processor to which the task is mapped.

A task graph is defined as in Chapter 2. It is a connected directed graph of tasks in which the directed edges represent finite size FIFO buffers. A task graph contains one source, τ_s , which executes strictly periodic with a period P_s and starts at time $s(\tau_s)$. This strictly periodic execution of the source imposes a throughput constraint on the other tasks in the task graph. We use P_i for the average period of a task τ_i .

All tasks in a task graph are (eventually) activated by the source of that task graph and we require that no (non-source) task can execute before the source of the task graph starts releasing data for the first time. This corresponds with the situation where the application has finished the initial behavior.

Tasks communicate using FIFO buffers which are directed from one task to another. We use c_{ij} for a FIFO buffer from task τ_i to τ_j . FIFO buffers have an optional predefined (maximum) capacity. The capacity of the FIFO buffers for which only a maximum capacity is specified, can be determined using buffer sizing techniques. The containers of a FIFO buffer can either be filled with data or be empty.

At the beginning of every execution, task τ_i acquires $\chi(c_{ki})$ full containers from each FIFO c_{ki} directed towards τ_i and acquires $\psi(c_{ij})$ empty containers from each output FIFO buffer. These empty containers are filled with data produced by task τ_i and then released at the end of the execution of τ_i as $\psi(c_{ij})$ full containers on the same FIFO from which they were acquired. The full containers which were

acquired at the beginning of an execution are released at the end of that execution of τ_i as $\chi(c_{ki})$ empty containers on FIFO c_{ki} . The number of acquired/released containers is constant for every execution of a task.

An execution of task τ_i can only start after τ_i is enabled. Task τ_i is enabled if the previous execution of task τ_i is finished and if it can acquire the required number of containers from the FIFO buffers.

The minimum amount of processor time a task τ_i requires to complete an execution is specified by its Best-Case Execution Time (BCET), B_i and the maximum required processor time is specified by its WCET, C_i . The time between an enabling and a corresponding finish of a task is defined by the response time of that execution of the task. Bounds on the response times of tasks are presented in the next section.

5.4 RESPONSE TIMES

In this section we provide upper and lower bounds on the response times of tasks. The bounds on response times will be used in the next sections to give guarantees on the temporal behavior of the application. Similar to [HHJ⁺05], we define the bounds on the response time of a task τ_i using its average period P_i and the enabling characterizations of the other tasks.

The same way as in [RRE03, SE09, TS09], only schedulers are supported for which the response times are monotonic in the enabling characterization. If the response times are not monotonic in the enabling characterizations of the tasks, then the complete analysis flow is not monotonic and convergence of the analysis cannot be guaranteed.

A small overview of the related work regarding the used response time analysis techniques is as follows. In [LL73] the concept of critical instants is introduced. Furthermore, in [LL73] a sufficient (not exact) schedulability bound based on the utilization of a processor is introduced which is applicable when a rate-monotonic priority assignment is used. An exact formulation of the worst-case response time of fixed priority scheduled processes is introduced in [JP86] and iterative determination of the worst-case response times is introduced in [ABR⁺93]. Finally, the concept of the level- i busy period is introduced in [Leh90] and extended in [TBW94]. A more advanced overview and history of the different response time analysis methods can be found in [ABD⁺95, SAA⁺04, But11]

To find response time bounds, we make use of local scheduling analysis techniques [RRE03, TBW94]. The basic idea is that the maximum/minimum interference of other tasks during an execution is added to the time that the task takes to execute. Using local scheduling analysis techniques enables to analyze the response times of the tasks running on one processor in isolation. Each processor of the MPSoC can thus be analyzed separately. The schedules that are obtained by dataflow analysis (see Section 5.5.4) ensure that all the precedence constraints are satisfied. For ease of understanding we use the most simple minimum response time definition, where the interference of other tasks is assumed to be

0. Therefore, the minimum response time $\check{R}_i$ of a task τ_i is defined as $\check{R}_i = B_i$. More accurate definitions of the minimum response time of a task are discussed in [RRE03].

As discussed in [HHJ⁺05], the maximum number of enablings of a task in a time interval Δt can be captured in a maximum enabling characterization, the $\hat{\eta}(\Delta t)$ function. Such an enabling characterization can also be expressed by the minimum distance between n enablings: $\check{\theta}(n)$. For the scope of this chapter we define them as follows:

Definition 5.1: (Maximum Enablings Function)

The maximum enablings function $\hat{\eta}_i(\Delta t)$ specifies the maximum number of times a task τ_i can get enabled in any time interval of length Δt .

Definition 5.2: (Minimum Enabling Distance Function)

The minimum enabling distance function $\check{\theta}_i(n)$ specifies the minimum distance between n , $n \geq 2$, consecutive enablings of task τ_i .

The $\check{\theta}(n)$ function can be converted into $\hat{\eta}(\Delta t)$ without loss of accuracy using:

$$\hat{\eta}(\Delta t) = \max_{n \in \mathbb{N}, n \geq 2} \left(\{n \mid \check{\theta}(n) < \Delta t\} \cup \{1\} \right) \quad (5.1)$$

A widely used enabling characterization uses the enabling jitter of the task. The maximum number of enablings of a task τ_i with a period P_i and an enabling jitter J_i can be bounded as follows [HHJ⁺05]:

$$\hat{\eta}_i(\Delta t) = \left\lceil \frac{J_i + \Delta t}{P_i} \right\rceil \quad (5.2)$$

The minimal distances between enablings of task τ_i can be expressed equivalent to this definition of the maximum number of enablings [HHJ⁺05]. Again for task τ_i with a period P_i and enabling jitter J_i , the minimum distance between n enablings is equal to:

$$\check{\theta}_i^J(n) = \max(0, (n - 1) \cdot P_i - J_i) \quad \text{with } n \geq 2 \quad (5.3)$$

The maximum number of enablings can be used to provide upper bounds on the response time of a task. We show this for a Static Priority Preemptive (SPP) scheduler, a Static Priority Non-Preemptive (SPNP) scheduler, a round-robin scheduler and a Time-Division Multiplexing (TDM) scheduler. We present for these schedulers a characterization of the busy period of a task. This busy period of task τ_i , $w_i(q)$, is equal to the maximum amount of time which a task τ_i takes to finish q consecutive executions, as long as task τ_i is constantly externally enabled during these q executions. Equivalent, we can say that $w_i(q)$ is equal to the maximum amount of processor demand of q consecutive executions of task τ_i plus the processor demand of the tasks that interfere with τ_i during this busy period.

This busy period can then be used to compute the maximum response times which we use in this chapter:

$$\hat{R}_i = \max_{q \in \mathbb{N}^+} (w_i(q) - (q - 1) \cdot P_i) \quad (5.4)$$

Only the values of q for which $w_i(q) \geq q \cdot P_i$ holds need to be considered [TBW94]. As long as this holds, the task is still externally enabling by the periodic enabling when an execution finishes. We define this as task τ_i being in a consecutive execution.

Note that the response time which we use differs from the response times that are used in related work [TBW94]. We define the response time of an execution of a task as the time between the enabling of the execution and the corresponding finish of the execution. Related work does use the arrival of events instead of the enabling of an execution and therefore includes queuing of events in the response time of a task. In our approach, this queuing of events is analyzed by the dataflow model and does therefore not need to be included in the response time. An important benefit of this difference is discussed in Section 5.8.

Furthermore, for the proposed analysis method we determine the response time of a task by assuming a periodic enabling of the task. The actual enablings of the tasks are not required to be periodic. In Section 5.5.3 we show that the busy period of a task together with such an assumed periodic enabling of task executions, can be used to determine a conservative upper bound on the actual temporal behavior of the tasks. This actual temporal behavior can thus be aperiodic. By using the busy period of a task we show that we can define a conservative upper bound on the finish times of that task.

We now present the busy period characterizations for the different types of run-time schedulers which we consider in this chapter.

5.4.1 STATIC PRIORITY PREEMPTIVE SCHEDULING

Each task mapped to a processor is assigned a distinct priority level when an SPP scheduler is used on that processor. The scheduler serves at each point in time the highest priority task that is enabled at that point in time.

The busy period of a task τ_i scheduled using an SPP scheduler, can be determined using Equation (5.5) [TBW94]. We have that for an SPP scheduler, this busy period is equal to the so-called level- i busy period [Leh90]. This level- i busy period is equal to the processor demand of the tasks with the highest i priority levels during q consecutive executions of task τ_i . The priority level of task τ_i is then equal to i .

$$w_i(q) = q \cdot C_i + \sum_{j \in hp(i)} \hat{\eta}_j(w_i(q)) \cdot C_j \quad (5.5)$$

We use $hp(i)$ as the set of tasks running on the same processor as τ_i and which have a higher priority than τ_i . As is shown in [TBW94], iterative fixed-point computation can be used to compute $w_i(q)$.

Note that for the focus of this chapter we choose to use the WCET for each task. Chapter 4 presents improvements on such a WCET characterization. These improvements exploit knowledge about varying execution times to improve the accuracy of the temporal results that can be obtained [MKT04, QHE12, HGWB13b]. Such methods use workload functions which capture the worst-case cumulative execution time of a task instead of its WCET. Workload functions can be easily incorporated in the computation of response times by using these workload functions instead of C_i , see [QHE12]. More information about workload functions can be found in Chapter 4

5.4.2 STATIC PRIORITY NON-PREEMPTIVE SCHEDULING

An SPNP scheduler is very similar to an SPP scheduler with the difference that tasks are not preempted after they are started. At each point in time the highest priority task that is enabled at that point in time is selected to start but the processor waits with serving the task until the running task is finished. The consequence is that tasks also can be delayed by lower priority tasks.

A task can maximally be blocked by one lower priority task. The maximum time that a task has to wait for lower priority to finish occurs when that lower priority task starts an infinitesimally small amount of time before the task itself is enabled. The maximum time the task is blocked, Ξ_i , is thus equal to time it takes to finish the longest lower priority task and is defined in Equation (5.6). We use $lp(i)$ as the set of tasks running on the same processor as task τ_i and which have a lower priority than τ_i .

$$\Xi_i = \max_{j \in lp(i)} C_j \quad (5.6)$$

Furthermore, during the execution of the task, no other task can interfere. The time it takes to finish q executions and the corresponding response time is thus equal to [DBBL07, NE12]:

$$w_i(q) = C_i + w'_i(q) \quad (5.7)$$

$$w'_i(q) = (q-1) \cdot C_i + \Xi_i + \sum_{j \in hp(i)} \hat{\eta}_j(w_i(q)) \cdot C_j \quad (5.8)$$

5.4.3 ROUND-ROBIN SCHEDULING

A round-robin scheduler activates the tasks one after each other and does not preempt the execution of a task. A round-robin scheduler is starvation-free because the interference of other tasks can be bounded when only the execution times of the other tasks are known. However, the accuracy of the temporal analysis results, of a system with a round-robin scheduler, can be improved by taking the actual traffic of the application into account.

A round-robin scheduler ensures that during q executions of a task τ_i , maximally q executions of each of the other tasks can interfere. With this extra

information we can refine the maximum number of activations of a task τ_i in an interval Δt given maximally q activations to:

$$\hat{\eta}'_i(q, \Delta t) = \min(q, \hat{\eta}_i(\Delta t)) \quad (5.9)$$

The busy period of task τ_i given round-robin scheduling can now, similarly as for SPP scheduling, be calculated with Equation 5.10 where $T(i)$ is the set of tasks running on the same processor as task τ_i without task τ_i itself. Again iterative fixed-point computation is used to compute $w_i(q)$.

$$w_i(q) = q \cdot C_i + \sum_{j \in T(i)} \hat{\eta}'_j(q, w_i(q)) \cdot C_j \quad (5.10)$$

5.4.4 TIME-DIVISION MULTIPLEXING SCHEDULING

A TDM scheduler belongs to the class of budget schedulers and guarantees every task a minimum time budget in a maximum replenishment interval. Despite the fact that TDM schedulers belong to the class of starvation-free schedulers the temporal analysis results might benefit from the analysis methods of this chapter.

We consider a TDM scheduler as described in [SBW09] where the tasks use a cooperative yield mechanism. Such a yield mechanism implies that when a task does not have sufficient data or space available, it stops its execution and allows other tasks to execute. Because of this mechanism, better temporal analysis results than for example [WBS07b] can be obtained when it is known when and how often tasks execute.

Identical to the method of [WBS07b] we use the fact that a task has a guaranteed time budget. A task τ_i receives every interval Q_i at least S_i budget. The interval Q_i is called the replenishment interval. To receive Δt processing time, a task τ_i requires maximally $\left\lceil \frac{\Delta t}{S_i} \right\rceil$ times its budgets. In the worst-case situation, task τ_i has to wait $(Q_i - S_i)$ time to receive its budget once. The maximum time that the task thus has to wait for Δt processing time is equal to $\left\lceil \frac{\Delta t}{S_i} \right\rceil \cdot (Q_i - S_i)$.

This upper bound on the waiting time can be combined with information on the interference from other task to obtain a tighter response time. We use $T(i)$ for the set of tasks running on the same processor as task τ_i without task τ_i itself. Again iterative fixed-point computation is used to compute $w_i(q)$.

$$w_i(q) = q \cdot C_i + \min \left(\sum_{j \in T(i)} \hat{\eta}'_j(w_i(q)) \cdot C_j, \left\lceil \frac{q \cdot C_i}{S_i} \right\rceil \cdot (Q_i - S_i) \right) \quad (5.11)$$

5.5 TEMPORAL ANALYSIS

This section provides a method to determine enabling characterizations of tasks based on dataflow analysis techniques. This method is used to update the enabling characterizations in each iteration of the analysis flow until the results have converged.

We use an SDF model for which we show that it is a temporally conservative abstraction of the task graphs of an application. The analysis method uses a conversion from the SDF model to an equivalent HSDF model [LM87, SB00]. We use the structure of this conversion to derive the enabling characterization for each task.

The idea of the analysis method is as follows. We use a conformance relation between task graphs and dataflow models as presented in Chapter 2 to define a dataflow model of a task graph (Section 5.5.2). We use the busy periods and response times of tasks to define this dataflow model. These busy periods and response times are updated in each iteration of the analysis flow and the dataflow model is thus also updated in each iteration of the analysis flow. The dataflow model is defined in such a way that with the assumed periodic schedule of tasks, an upperbound on the worst-case temporal behavior of tasks is determined, see Section 5.5.3.

We use two variants of the dataflow model. The first variant forms a lower bound on the best-case temporal behavior of the task graph. The second variant forms an upper bound on the worst-case temporal behavior of the task graph. With these two variants of the dataflow model, we determine lower and upper bounds on the so-called external enablings and finishes of tasks. These enablings and finishes of tasks are then used to update the enabling characterizations of tasks (Section 5.5.4).

5.5.1 ANALYSIS MODEL

We use the SDF model as the temporal analysis model in this chapter. We use the same definition of an SDF model as in Chapter 2 and shortly repeat its properties in this section.

Functionally deterministic timed dataflow graphs, such as SDF graphs, have a monotonic temporal behavior. This means that increasing the firing duration of one actor in the graph can never lead to an earlier enabling of any of the actors in the graph and also, decreasing the firing durations of one of the actors can never lead to any later enabling. Similarly, increasing the amount of initial tokens in the graph can never lead to a later enabling of any of the actors in the graph and the opposite, decreasing the amount of initial tokens can never lead to any earlier enabling. Modifying the number of tokens on an edge is only allowed when this modification does not change the functional behavior. See Section 2.5.3 for more details on monotonicity of SDF graphs.

Also consistency of an SDF model is an important property. For an inconsistent SDF model, either no schedule is deadlock-free or there is an unbounded accumulation of tokens.

For consistent SDF models a repetition vector $\mathbf{q}$ can be determined. This repetition vector contains the relative firing frequencies between the actors. We use q_i for the repetition factor of actor v_i where q_i is the i th component of $\mathbf{q}$. If each actor v_i in the SDF model fires exactly q_i times then it holds for each edge in

Figure 5.2: Conformance relation between the SDF model and the task graph.

the SDF model that the number of tokens produced on an edge is equal to the number of consumed tokens from that edge. See Section 2.3.2 for more details on consistency of SDF graphs.

The average period at which a task executes is different than the period of the source in the task graph, P_s . The period of a task can be computed by using the relative firing frequencies. The average rate at which tasks consume data should also be equal to the rate at which the source produces data. The average period of a task is equal to the period of its corresponding SDF actor and can be calculated with Equation (5.12).

$$\forall v_i \in V : P_i = \frac{q_s}{q_i} \cdot P_s \quad (5.12)$$

5.5.2 RELATION BETWEEN SDF MODEL AND TASK GRAPH

Figure 5.2 illustrates the conformance relation between the SDF model and the task graph. This relation adheres to the conformance relation between actor components and tasks as presented in Chapter 2. A task τ_i in the task graph corresponds to an actor component consisting of two unique actors, $v_{i,0}$ and $v_{i,1}$, in the figure. One token is produced/consumed per firing on the edge from $v_{i,0}$ to $v_{i,1}$ which is thus homogeneous. Furthermore, $v_{i,1}$ has a self-edge with one token to denote that its firings may not overlap. Actor $v_{i,1}$ models the maximum rate at which the actor component can transfer tokens and actors $v_{i,0}$ and $v_{i,1}$ together, model the latency introduced by the actor component.

As presented in in Chapter 2 we have that a FIFO buffer c_{ij} is modeled using two oppositely directed edges. The first edge, edge e_{ij} , connects actor $v_{i,1}$ to $v_{j,0}$. The second edge, edge e_{ji} , is directed from $v_{j,1}$ to $v_{i,0}$. The forward edge, e_{ij} , models the flow of full containers and contains initially $\delta(e_{ij})$ tokens. Similarly,

Figure 5.3: Actor component for modeling the worst-case behavior of task τ_i .

the backwards edge, e_{ji} , models the flow of empty containers and has initially $\delta(e_{ji})$ tokens. The number of tokens that are produced and consumed on these edges is equal to the number of released and acquired containers as specified in Section 5.3, i.e., $\pi(e_{ij}) = \gamma(e_{ji}) = \psi(c_{ij})$ and $\pi(e_{ji}) = \gamma(e_{ij}) = \chi(c_{ij})$. The source τ_s is modeled with a single actor v_s which starts firing at the start time of τ_s : $s(v_s) = s(\tau_s)$.

The size of the FIFO from task τ_i to τ_j is equal to $\delta(e_{ij}) + \delta(e_{ji})$. If only a maximum size for this FIFO buffer is specified, the number of tokens on e_{ji} is chosen to be equal to the maximum buffer size. This allows the maximum scheduling freedom. Furthermore, as shown in Chapter 2 increasing the amount of tokens on e_{ji} is a valid refinement because this edge models the flow of empty containers for which the value of the tokens is undefined, i.e., the value is equal to $\emptyset$.

5.5.3 TEMPORALLY CONSERVATIVE MODELING

A dataflow model is temporally conservative to a task graph if the temporal behavior of the task graph is captured conservatively by the dataflow model. As shown in Chapter 2 this can be proven for a complete task graph and dataflow model by proving for each pair of a task and corresponding actor component the following:

$$\forall_{k \in \mathbb{N}} : e\langle k \rangle \sqsubseteq \hat{e}\langle k \rangle \implies f\langle k \rangle \sqsubseteq \hat{f}\langle k \rangle \quad (5.13)$$

Where $e\langle k \rangle$ is the sequence of enabling times up to execution k of a task and $\hat{e}\langle k \rangle$ the enabling sequence of the corresponding actor component firing up to firing k . The finish times of the task are denoted with f and $\hat{f}$ represents the finish times of the firings of the corresponding actor component.

The best-case behavior of the task graph can be bounded similarly as is shown in Chapter 2. For each pair of a task and the corresponding actor component the following property needs to hold.

$$\forall_{k \in \mathbb{N}} : e\langle k \rangle \sqsupseteq \hat{e}\langle k \rangle \implies f\langle k \rangle \sqsupseteq \hat{f}\langle k \rangle \quad (5.14)$$

The worst-case behavior of task τ_i is modeled with the actor component of Figure 5.3. We now prove that this actor component is conservative given the definition of response times of Section 5.4.

Without loss of generality we can state that execution k of task τ_i belongs to a consecutive execution that starts at execution $n \leq k$. Furthermore, the function $w_i(q)$ is defined as the maximum amount of time it takes to finish q executions of task τ_i . The finish time of an execution k of task τ_i is thus bounded by:

$$f_i(k) \leq \max_{n \leq j \leq k} (e_i(j) + w_i(k - j + 1))$$

Furthermore, given the definition of response times of Equation (5.4) we know that:

$$\forall q : w_i(q) \leq \hat{R}_i + (q - 1) \cdot P_i$$

When the begin of the consecutive execution (n) is known and by using the maximum response time of τ_i we can define an upper bound, $f_i^u(k)$, on the finish time of execution k of task τ_i with Equation (5.15).

$$f_i(k) \leq f_i^u(k) = \max_{n \leq j \leq k} (e_i(j) + \hat{R}_i + (k - j) \cdot P_i) \quad (5.15)$$

However, during analysis it is unknown at which execution a consecutive execution starts. Therefore, we derive an additional upper bound, $f_i^w(k)$, on $f_i^u(k)$ which is independent of this start of the consecutive execution. After this derivation it is shown with this new upper bound that the actor component of Figure 5.3 is conservative to the corresponding task.

Theorem 5.1. *An upper bound on the finish time of execution k of task τ_i is given by:*

$$f_i(k) \leq f_i^w(k) = \max(e_i(k) + \max(0, \hat{R}_i - P_i), f_i^w(k - 1)) + \min(\hat{R}_i, P_i) \quad (5.16)$$

Proof. We can distinguish two cases: either $\hat{R}_i \leq P_i$ or $\hat{R}_i > P_i$ holds.

Case $\hat{R}_i \leq P_i$: when $\hat{R}_i \leq P_i$ then $f_i^w(k) = \max(e_i(k), f_i^w(k - 1)) + \hat{R}_i$. Using the definition of enabling and response times of tasks it holds that: $f_i(k) \leq \max(e_i(k), f_i(k - 1)) + \hat{R}_i$. With induction on k it can be shown that $f_i(k) \leq f_i^w(k)$ holds.

Case $\hat{R}_i > P_i$: to prove this case, the upper bound function f_i^u from Equation (5.15) is used. If $\hat{R}_i > P_i$, then $f_i^w(k) = \max(e_i(k) + \hat{R}_i - P_i, f_i^w(k - 1)) + P_i$. If for $f_i^u(k)$, $j = k$ gives the maximum value then $f_i^u(k) = e_i(k) + \hat{R}_i$ and thus $f_i^u(k) \leq f_i^w(k)$.

Otherwise, $j = k$ does not give the maximum value and then we have $f_i^u(k) = \max_{n \leq j \leq k-1} (e_i(j) + \hat{R}_i + (k - j) \cdot P_i)$. Because $j \neq k$ we have that execution k does not start a new consecutive execution and thus $f_i^u(k - 1) = \max_{n \leq j \leq k-1} (e_i(j) + \hat{R}_i + (k - 1 - j) \cdot P_i)$. By subtracting $f_i^u(k - 1)$ from $f_i^u(k)$ we get: $f_i^u(k) - f_i^u(k - 1) = P_i$ and thus $f_i^u(k) = f_i^u(k - 1) + P_i$. We had $f_i^w(k) = \max(e_i(k) + \hat{R}_i - P_i, f_i^w(k - 1)) + P_i$ and thus by induction it can be showed that $\forall k : f_i^u(k) \leq f_i^w(k)$ holds.

Combining the cases $j = k$ and $j \neq k$ proofs that $f_i^u(k) \leq f_i^w(k)$. Because $f_i(k) \leq f_i^u(k)$ holds, see Equation (5.15), also $f_i(k) \leq f_i^w(k)$ holds. $\square$

Using max-plus algebra [BCOQ92] on the actor component of Figure 5.3 it can be shown that:

$$\hat{f}_i(k) = \max(\hat{e}_i(k) + \max(0, \hat{R}_i - P_i), \hat{f}_i(k-1)) + \min(\hat{R}_i, P_i) \quad (5.17)$$

With these derived equations we show that the actor component as shown in Figure 5.3 is conservative to its corresponding task τ .

Lemma 5.1. *Equation (5.13) holds for the conformance relation between a task and an actor component if the actor component is chosen as is shown in Figure 5.3.*

Proof. With (5.16) and (5.17) and assuming $f^w(-1) \leq \hat{f}(-1)$ we can prove with induction on the execution number that $e\langle i \rangle \sqsubseteq \hat{e}\langle i \rangle \implies f^w\langle i \rangle \sqsubseteq \hat{f}\langle i \rangle$ holds for every execution i of task τ . This is sufficient to prove Equation (5.13) because $\forall_i : f(i) \leq f^w(i)$ holds.

Base: When $e(0) \leq \hat{e}(0)$ and because $f^w(-1) \leq \hat{f}(-1)$ we have that:

$$\begin{aligned} \max(e_i(0) + \max(0, \hat{R}_i - P_i), f_i^w(-1)) + \min(\hat{R}_i, P_i) \leq \\ \max(\hat{e}_i(0) + \max(0, \hat{R}_i - P_i), \hat{f}_i(-1)) + \min(\hat{R}_i, P_i) \end{aligned}$$

And thus $f^w(0) \leq \hat{f}(0)$. Therefore, we have $e\langle 0 \rangle \sqsubseteq \hat{e}\langle 0 \rangle \implies f^w\langle 0 \rangle \sqsubseteq \hat{f}\langle 0 \rangle$.

Step: Given that $e\langle i-1 \rangle \sqsubseteq \hat{e}\langle i-1 \rangle \implies f^w\langle i-1 \rangle \sqsubseteq \hat{f}\langle i-1 \rangle$ holds we need to show that $e\langle i \rangle \sqsubseteq \hat{e}\langle i \rangle \implies f^w\langle i \rangle \sqsubseteq \hat{f}\langle i \rangle$ holds.

When $e\langle i \rangle \sqsubseteq \hat{e}\langle i \rangle$ holds we also have $e\langle i-1 \rangle \sqsubseteq \hat{e}\langle i-1 \rangle$ and given the induction hypothesis we can conclude $f^w\langle i-1 \rangle \sqsubseteq \hat{f}\langle i-1 \rangle$. We have that $f^w\langle i \rangle \sqsubseteq \hat{f}\langle i \rangle$ is equivalent to $f^w\langle i-1 \rangle \sqsubseteq \hat{f}\langle i-1 \rangle \wedge f^w(i) \leq \hat{f}(i)$ which means that we only have to prove $f^w(i) \leq \hat{f}(i)$.

Given $e\langle i \rangle \sqsubseteq \hat{e}\langle i \rangle$ and $f^w\langle i-1 \rangle \sqsubseteq \hat{f}\langle i-1 \rangle$ we conclude $e(i) \leq \hat{e}(i)$ and $f^w(i-1) \leq \hat{f}(i-1)$ and thus:

$$\begin{aligned} \max(e(i) + \max(0, \hat{R}_i - P_i), f^w(i-1)) + \min(\hat{R}_i, P_i) \leq \\ \max(\hat{e}(i) + \max(0, \hat{R}_i - P_i), \hat{f}(i-1)) + \min(\hat{R}_i, P_i) \end{aligned}$$

Which means that $f^w(i) \leq \hat{f}(i)$. □

It is shown in the next section that the schedule of the used two-actor SDF component is multi-dimensional periodic. The average period of this SDF component is equal to the average period of the corresponding task but the time between two consecutive firings of the SDF component can be different. The response time definition of task τ_i that is used requires a minimal distance of P_i which is, as we proved, ensured by the two-actor SDF component of Figure 5.3.

The best-case behavior of task τ_i is modeled by choosing the firing duration of actor $v_{i,0}$ equal to 0, $\check{\rho}_{i,0} = 0$, and the firing duration of actor $v_{i,1}$ equal to $\check{R}_i$, $\check{\rho}_{i,1} = \check{R}_i$. This is illustrated in Figure 5.4. Because actor $v_{i,0}$ has a firing duration equal to 0 we have that the actor component is actually a one-actor component consisting of actor $v_{i,1}$. With this model, Equation (5.14) can be proven similarly.

Figure 5.4: Actor component for modeling the best-case behavior of task τ_i .

5.5.4 DETERMINE ENABLING CHARACTERIZATIONS

To determine the enabling characterization of tasks, we first transform the SDF models for the best- and worst-case behavior into equivalent HSDF models using the method of [LM87, SB00]. This conversion transforms each SDF actor v_a of the SDF model into q_a actors in the HSDF model. Each of these q_a actors model a certain firing of the SDF actor v_a and each of these firings can cause different enablings of other actors in the SDF model. After q_a firings of actor v_a , all the different enablings have occurred. We call this the repetition period of an actor.

We number the actors in the HSDF model as $v_a^0 \dots v_a^{q_a-1}$ where v_a^0 corresponds to the first firing in the repetition period of actor v_a and $v_a^{q_a-1}$ to the last firing in this repetition period. Furthermore, we use V_H as the set of HSDF actors and E_H as the set of edges in the HSDF graph.

Each SDF actor v_a has an average period P_a as defined in Equation (5.12). Because each SDF actor is split in q_a actors in the HSDF graph, on average, each of these HSDF actors have to fire once per $q_a \cdot P_a$. We use P_i for the period of HSDF actor v_i . As can be seen with Equation (5.12), the periods of all the HSDF actors in the HSDF model are identical and are in fact equal to $q_s \cdot P_s$. We thus have that for a pair of actors v_i and v_j holds that $P_i = P_j$.

Also the SDF actor v_s , corresponding to the the source task τ_s , is modeled using q_s actors. Each of these actors should fire periodically with a period equal to $q_s \cdot P_s$. Actor v_s^0 corresponds to the first firing of actor v_s and therefore starts at time $s(\tau_s)$. Each next firing corresponds to a new execution of the source and thus is defined to start exactly P_s time later: $s(v_s^k) = s(v_s^{k-1}) + P_s$.

With the HSDF models for the best- and the worst-case behavior and the derived constraints we can determine periodic lower and upper bound schedules for the actors. The determined HSDF schedules bound the self-timed schedule of the SDF actors from below and from above respectively. Recall that in such a self-timed schedule of a dataflow model, each actor starts its firing as soon as it is enabled. The start time of an actor is thus equal to its enabling time. Furthermore, we know that for HSDF graphs a periodic schedule exists which is rate-optimal [MB07].

Linear Programming (LP) Algorithm 1 determines the earliest possible start times of the first firing of the HSDF actors of the best-case HSDF model. The algorithm has a polynomial time-complexity.

We have that edges that initially contain tokens are omitted because they will not impose a precedence constraint on the first firing and thus do not influence the enabling time of the first firing. Constraint-wise this corresponds with assuming an infinite number of tokens on the edges that initially contain at least one token. Because all the actors are enabled by a source, the earliest possible schedule after the first firing of an HSDF actor that corresponds to SDF actor v_a , is a periodic schedule with period $q_a \cdot P_a$. Thus, by omitting the edges that initially contain tokens and by determining the earliest possible schedule for the actors such that all the constraints of the dataflow model are met, a safe lower bound on the schedule of each actor is found.

We determine an admissible schedule. A schedule is admissible when all the tokens are consumed from an edge after they are produced on that edge. This can be ensured by adhering to the precedence constraint imposed by each edge e_{ij} [MB07]. We use $\check{s}_i$ for the earliest start time of the first firing of HSDF actor v_i and $\check{s}_i(k)$ for the start time of firing k of HSDF actor v_i . Each firing of an actor v_i finishes $\check{\rho}_i$ after its start and we thus have an admissible schedule when for each edge e_{ij} holds that:

$$\forall k \in \mathbb{N} : \check{s}_j(k + \delta(e_{ij})) \geq \check{s}_i(k) + \check{\rho}_i \quad (5.18)$$

We have that $\delta(e_{ij})$ is equal to zero for each edge. Furthermore, we have that the schedule of the HSDF actors is strictly periodic and that the period of each HSDF actor is equal. We thus have that Constraint 5.20 ensures that Equation 5.18 holds for each edge. Furthermore, the periodic schedule of the actors corresponding to the source task is enforced by Constraint 5.19.

Algorithm 1

Minimize

$$\sum_{v_i \in V_H} \check{s}_i$$

Subject to

$$\forall 0 \leq k < q_s : \check{s}_{s_k} = s(\tau_s) + k \cdot P_s \quad (5.19)$$

$$\forall e_{ij} \in E_{H'} : \check{s}_j - \check{s}_i \geq \check{\rho}_i \quad (5.20)$$

$$\text{with } E_{H'} = \{e_{ij} \mid e_{ij} \in E_H \wedge \delta(e_{ij}) = 0\}$$

The worst-case enabling times of the HSDF actors in the worst-case HSDF model are determined with LP Algorithm 2. Start times are calculated such that all the precedence constraints are satisfied and such that the number of initial tokens allows a periodic schedule that starts at the determined enabling time. We use $\hat{s}_i$ for the latest possible moment in time after which an HSDF actor v_i can start a strictly periodic with a period P_i . We use $\hat{s}_i(k)$ for the start time of firing k of HSDF actor v_i in this periodic schedule.

We again determine an admissible schedule and thus for each edge e_{ij} in the HSDF graph it should hold that:

$$\forall_{k \in \mathbb{N}} : \hat{s}_j(k + \delta(e_{ij})) \geq \hat{s}_i(k) + \hat{\rho}_i \quad (5.21)$$

We have that the schedule of each HSDF actor v_i is strictly periodic with a period equal to P_i . We thus have that the start time of firing k of HSDF actor v_i in the worst-case model is equal to:

$$\hat{s}_i(k) = \hat{s}_i + k \cdot P_i$$

With this periodic schedule we have that Equation 5.21 is ensured when the following holds:

$$\forall_{k \in \mathbb{N}} : \hat{s}_j + (k + \delta(e_{ij})) \cdot P_j \geq \hat{s}_i + k \cdot P_i + \hat{\rho}_i$$

Because for each edge e_{ij} holds that $P_i = P_j$, we have that Equation 5.5.4 is ensured by Constraint 5.23 of Algorithm 2. The periodic schedule of the actors corresponding to the source task is enforced by Constraint 5.22. If Algorithm 2 returns infeasible, a schedule with the required throughput does not exist, i.e., the period is too small or not enough initial tokens are present.

Algorithm 2

Minimize

$$\sum_{v_i \in V_H} \hat{s}_i$$

Subject to

$$\forall_{0 \leq k < q_s} : \hat{s}_{s^k} = s(\tau_s) + k \cdot P_s \quad (5.22)$$

$$\forall_{e_{ij} \in E_H} : \hat{s}_j - \hat{s}_i \geq \hat{\rho}_i - \delta(e_{ij}) \cdot P_i \quad (5.23)$$

Algorithm 2 as well as Algorithm 1 can be written as a dual of the uncapacitated network flow problem which can be solved efficiently [BT97]. In fact it can even be written as a shortest path problem that can be solved using the Bellman-Ford algorithm. The Bellman-Ford algorithm can detect negative cycles and the detection of such a negative cycle indicates infeasibility of the specified temporal constraints.

The determined schedules give for each of the HSDF actors strictly periodic enabling times of the actors. Although the schedule for every HSDF actor v_i is periodic (with a period P_i), they are not necessarily strictly periodic for the schedule of the corresponding SDF actor. The time between two consecutive enablings of the SDF actor v_a can be different than P_a . The only information that

is known is that the schedules are periodic over q_a firings. The SDF schedules are thus multi-dimensional periodic.

Using the best-case schedules for the HSDF actor components, we define a lower bound, $\check{s}_{a,1}$, on the enabling times of an SDF actor $v_{a,1}$ in Equation (5.24). Note that $\check{s}_{a,1}(i)$ is defined recursively and makes use of the periodicity (period $q_a \cdot P_a$) of the schedules of the HSDF actors. We use $\check{s}(v_{a,1}^i)$ for the start time of the periodic schedule of HSDF actor $v_{a,1}^i$ as is determined by Algorithm 1.

$$\check{s}_{a,1}(i) = \begin{cases} \check{s}(v_{a,1}^i) & \text{if } 0 \leq i < q_a \\ \check{s}_{a,1}(i - q_a) + q_a \cdot P_a & \text{otherwise} \end{cases} \quad (5.24)$$

An SDF actor component in the best-case model consists of only actor $v_{a,1}$ and it thus holds that this lower bound is also a lower bound on the external enabling times of task τ_a . We have that the firing duration of this actor is smaller or equal than period P_a otherwise the task would not be schedulable by definition. Therefore, the actual enabling time of task τ_a is equal to its external enabling time. We use $\check{e}_a$ for the lower bound on the actual enabling times of task τ_a :

$$\check{e}_a(i) = \check{s}_{a,1}(i) \quad (5.25)$$

We define the upper bound on the enabling time of task τ_a by using the maximum finish time, $\hat{f}_{a,1}$, of the SDF actor component and the worst-case external enabling time of the SDF actor component. The maximum finish time is defined in Equation (5.26) using the finish times of the HSDF actor components which are equal to the worst-case start times of actors $v_{a,1}^i$ plus their firing durations. We define the finish time of non existing firings to be equal to $-\infty$ to ease the definition of the enabling times of the tasks. We use $\hat{s}(v_{a,1}^i)$ for the start time of the periodic schedule of HSDF actor $v_{a,1}^i$ as is determined by Algorithm 2.

$$\hat{f}_{a,1}(i) = \begin{cases} -\infty & \text{if } i < 0 \\ \hat{s}(v_{a,1}^i) + \rho(v_{a,1}) & \text{if } 0 \leq i < q_a \\ \hat{f}_{a,1}(i - q_a) + q_a \cdot P_a & \text{otherwise} \end{cases} \quad (5.26)$$

Task τ_a always finishes its execution i earlier than SDF actor $v_{a,1}$ finishes its firing i (see Section 5.5.3). Furthermore, the external enabling time of SDF actor $v_{a,0}$ forms an upper bound on the time that task τ_a can acquire the required number of containers, i.e., the external enabling time (see Section 5.5.3). This external enabling time of SDF actor $v_{a,0}$, $\hat{s}_{a,0}$, is recursively defined using the HSDF schedule as follows:

$$\hat{s}_{a,0}(i) = \begin{cases} \hat{s}(v_{a,0}^i) & \text{if } 0 \leq i < q_a \\ \hat{s}_{a,0}(i - q_a) + q_a \cdot P_a & \text{otherwise} \end{cases} \quad (5.27)$$

As defined in Section 5.3, an execution of a task is enabled when the previous execution is finished and when it can acquire the required number of containers from the buffers. Therefore, the actual enabling of task τ_a is never later than:

$$\hat{\epsilon}_a(i) = \max(\hat{f}_{a,1}(i-1), \hat{s}_{a,0}(i)) \quad (5.28)$$

With these earliest and latest enabling times of the tasks, the distance between $n \geq 2$ consecutive enablings, $\check{\theta}_a(n)$, can be determined. The smallest distance between n enablings can be computed as follows:

$$\check{\iota}_a(n \geq 2) = \min_{0 \leq k \leq q_a} (\check{\epsilon}_a(k+n-1) - \hat{\epsilon}_a(k)) \quad (5.29)$$

$$\check{\theta}_a(n \geq 2) = \max(0, \check{\iota}_a(n)) \quad (5.30)$$

Note that the bounded domain of the $\min$ term in Equation (5.29) is a direct result of the periodic nature of Equations (5.24), (5.26) and (5.27). For $k > q_a$ we indeed have:

$$\begin{aligned} \hat{\epsilon}_a(k) &= \max(\hat{f}_{a,1}(k-1), \hat{s}_{a,0}(k)) \\ &= \max(\hat{f}_{a,1}(k-1-q_a), \hat{s}_{a,0}(k-q_a)) + q_a \cdot P_a \\ &= \hat{\epsilon}_a(k-q_a) + q_a \cdot P_a \end{aligned}$$

It also holds that $\check{\epsilon}_a(k) = \check{\epsilon}_a(k-q_a) + q_a \cdot P_a$ and thus values $k > q_a$ can not result in smaller distances in the function $\check{\iota}_a$. Furthermore, it can be shown that $\check{\iota}_a$ is periodic:

Lemma 5.2. $\check{\iota}_a(n+q_a) = \check{\iota}_a(n) + q_a \cdot P_a$

Proof. For $i \geq q_a$ we have $\check{\epsilon}_a(i) = \check{\epsilon}_a(i-q_a) + q_a \cdot P_a$ and thus:

$$\begin{aligned} \check{\iota}_a(n+q_a) &= \min_{0 \leq k \leq q_a} (\check{\epsilon}_a(k+n+q_a-1) - \hat{\epsilon}_a(k)) \\ &= \min_{0 \leq k \leq q_a} (\check{\epsilon}_a(k+n-1) - \hat{\epsilon}_a(k)) + q_a \cdot P_a \end{aligned}$$

□

Because $\check{\iota}_a$ is periodic, $\check{\theta}_a$ will eventually also be periodic and $\check{\theta}_a$ thus has a finite length.

The function $\check{\theta}_a(n)$ can be transformed into $\hat{\eta}_a(\Delta t)$ by using Equation (5.1) to find the new enabling characterizations for each task.

The used derivation of the maximum number of enablings of a task ensures that the analysis flow is monotone.

Lemma 5.3 (Monotonicity of the analysis flow). *The response times of tasks monotonically increase in each iteration of the analysis flow of Figure 5.1.*

Proof. Because of monotonicity of functional deterministic timed dataflow models we know that increasing the firing duration of an actor can never result in an earlier enabling of one of the actors in the dataflow model. Now assume that none of the maximum response times is decreased in an iteration of the analysis flow. Then the firing durations of the actors in the worst-case analysis model do not decrease. Because of monotonicity of dataflow models we know that then also the worst-case enabling times of all the actors in the graph do not decrease. When these enabling times do not decrease, then it can easily be verified that the minimum distances as derived in Equation 5.30 do not increase. The maximum numbers of enablings of tasks do therefore also not decrease. Then also the response times of tasks that are used in the next iteration of the analysis flow do not decrease which ensures the monotonicity of the analysis flow. $\square$

5.5.5 ENABLING JITTER

Also the enabling jitter based characterization of a task can be derived. Such an enabling jitter characterization can be useful because it belongs to the class of standard event models [HHJ⁺05]. Furthermore, the enabling characterization of applications that can be modeled using an HSDF model can be accurately described by using such an enabling jitter characterization [HGWB13a]. In Section 5.6 we show that we can also use this enabling jitter characterization to linearize response times and define a conservative analysis flow with a polynomial time-complexity. And, finally, we use this enabling jitter characterization to compare its accuracy with the accuracy of the more general enabling characterization which is derived in the previous section. This comparison is performed in Section 5.9.

As show in Section 5.4, the minimal distances between enablings of a task τ_a is conservatively expressed by its enabling jitter J_a and its period P_a with Equation (5.31).

$$\check{\theta}_a^j(n) = \max(0, (n-1) \cdot P_a - J_a) \quad \text{with } n \geq 2 \quad (5.31)$$

With this expression we can derive an enabling jitter such that the minimal distances are conservative (smaller) to the minimal distances that are derived in the previous section.

Lemma 5.4. *The enabling jitter of task τ_a is conservative when we choose:*

$$J_a = \max_{2 \leq n < (q_a+2)} ((n-1) \cdot P_a - \check{i}_a(n))$$

Proof. From Lemma 5.2 we know that $\check{i}_a(n)$ has a periodic behavior. For $n \geq 2$ it holds that $\check{i}_a(n + q_a) = \check{i}_a(n) + q_a \cdot P_a$. Therefore, when we calculate the enabling jitter as above we have:

$$\forall_{n \geq 2} : (n-1) \cdot P_a - \check{i}_a(n) \leq J_a$$

This can be rewritten to show that the obtained minimal distances are smaller, and thus conservative, when the enabling jitter based characterization is used:

$$\begin{aligned} \forall_{n \geq 2} : (n-1) \cdot P_a - J_a &\leq \check{l}_a(n) \\ \forall_{n \geq 2} : \max(0, (n-1) \cdot P_a - J_a) &\leq \max(0, \check{l}_a(n)) \\ \forall_{n \geq 2} : \check{\theta}_a^j(n) &\leq \check{\theta}_a(n) \end{aligned}$$

□

HSDF model As is shown in [HGWB13a], an enabling jitter characterization is accurate for HSDF models. However, the results obtained in [HGWB13a] are potentially optimistic. The derived maximum enabling time does not take into account that a task can be blocked by itself, it only uses the latest time at which the task can acquire the containers from the buffers, i.e., the external enabling time of a task. In this section we solve this optimism by also taking the finish time of the previous execution into account. Furthermore, we derive the enabling jitter characterization for HSDF models by rewriting the corresponding formulas.

As is already mentioned, for HSDF graphs there exists a periodic schedule which is rate-optimal [MB07]. With this additional information, the derived formulas of the previous sections can be simplified and a closed-form expression for the enabling jitter of a task can be derived.

The best-case enabling time of a task τ_a is as derived in Section 5.5.4. The worst-case enabling time of task τ_a will be simplified. By using a strict periodic schedule for each actor, the equations for the enabling times can also be formulated strict periodically.

Furthermore, in an HSDF model with strict periodic schedules for each actor we know that the enabling of actor $v_{a,1}$ of each actor component does only depend on the finish of actor $v_{a,0}$. The self-edge is not limiting the schedule because its firing duration is always smaller or equal than the period of the used schedule. The latest enabling time of actor $v_{a,1}$ is thus equal to:

$$\hat{s}_{a,1}(i) = \hat{s}_{a,0}(i) + \rho(v_{a,0}) \quad (5.32)$$

We know from Section 5.5.4 that task τ_a is always enabled earlier than:

$$\hat{\varepsilon}_a(i) = \max(\hat{f}_{a,1}(i-1), \hat{s}_{a,0}(i)) \quad (5.33)$$

Using the additional information about the strict periodic schedules of the

HSDF model we rewrite this maximum enabling time to:

$$\begin{aligned}
\hat{\varepsilon}_a(i) &= \max \left(\hat{f}_{a,1}(i-1), \hat{s}_{a,0}(i) \right) \\
&= \max \left(\hat{s}_{a,1}(i-1) + \rho(v_{a,1}), \hat{s}_{a,0}(i) \right) \\
&= \max \left(\hat{s}_{a,1}(i-1) + \rho(v_{a,1}), \hat{s}_{a,1}(i) - \rho(v_{a,0}) \right) \\
&= \max \left(\hat{s}_{a,1}(0) + (i-1) \cdot P_a + \rho(v_{a,1}), \hat{s}_{a,1}(0) + i \cdot P_a - \rho(v_{a,0}) \right) \\
&= \hat{s}_{a,1}(0) + i \cdot P_a + \max \left(\rho(v_{a,1}) - P_a, -\rho(v_{a,0}) \right) \\
&= \hat{s}_{a,1}(0) + i \cdot P_a + \max \left(\rho(v_{a,0}) + \rho(v_{a,1}) - P_a, 0 \right) - \rho(v_{a,0})
\end{aligned}$$

We know that: $\rho(v_{a,0}) + \rho(v_{a,1}) = \hat{R}_a$ and

$$\begin{aligned}
\rho(v_{a,0}) &= \max \left(0, \hat{R}_a - P_a \right) \text{ and thus:} \\
&= \hat{s}_{a,1}(0) + i \cdot P_a + \max \left(\hat{R}_a - P_a, 0 \right) - \max \left(0, \hat{R}_a - P_a \right) \\
&= \hat{s}_{a,1}(0) + i \cdot P_a
\end{aligned} \tag{5.34}$$

This simplified equation for the maximum enabling time of a task can now be used to formulate the smallest distance between n enablings of task τ_a easier:

$$\begin{aligned}
\check{\imath}_a(n \geq 2) &= \min_{0 \leq k \leq 1} (\check{\varepsilon}_a(k+n-1) - \hat{\varepsilon}_a(k)) \\
&= \min_{0 \leq k \leq 1} \left(\check{s}_{a,1}(0) + (k+n-1) \cdot P_a - \left(\hat{s}_{a,1}(0) + k \cdot P_a \right) \right) \\
&= \min_{0 \leq k \leq 1} \left(\check{s}_{a,1}(0) + (n-1) \cdot P_a - \hat{s}_{a,1}(0) \right) \\
&= (n-1) \cdot P_a + \check{s}_{a,1}(0) - \hat{s}_{a,1}(0)
\end{aligned} \tag{5.35}$$

With these derived equations we now are able to derive a closed-form expression for the enabling jitter of tasks. The enabling jitter of task τ_a is equal to:

$$\begin{aligned}
J_a &= \max_{2 \leq n < (q_a+2)} ((n-1) \cdot P_a - \check{\imath}_a(n)) \\
J_a &= \max_{2 \leq n < (q_a+2)} \left((n-1) \cdot P_a - ((n-1) \cdot P_a + \check{s}_{a,1}(0) - \hat{s}_{a,1}(0)) \right) \\
J_a &= \hat{s}_{a,1}(0) - \check{s}_{a,1}(0)
\end{aligned} \tag{5.36}$$

5.6 LINEARIZED RESPONSE TIME

In this section we provide a temporally conservative abstraction of the steps 1, 2, 3 and 4 of the analysis flow of Figure 5.1. This conservative analysis flow has a polynomial time-complexity which is accomplished by linearizing the response times and by using the enabling jitter characterization of a task. We present this conservative analysis flow for HSDF models.

The enabling jitter characterization for HSDF models is linear in the enabling times of a task and can therefore be used to make the response time of tasks

linear in the start times of the tasks [HGWB13a]. However, the enabling jitter characterization of a task modeled with an SDF actor is not linear in the enabling times of that task. To define a linear relation between the enabling times of a task and the response time of that task, strict periodic abstractions of the schedules of the corresponding SDF actors can be used. With such periodic schedules, the enabling jitter characterization becomes linear in the enabling times. This approach is similar to the approach used by temporal analysis methods based on linear bounds of the schedules [WBS07a, WBS07]. For SDF models we only present how to determine these strict periodic schedules and for simplicity focus on HSDF models in the remainder of this section.

We define a strict periodic upper bound, $\hat{s}_{a,1}^p$, on the maximum start times of actor $v_{a,1}$ and a strict periodic lower bound, $\check{s}_{a,1}^p$, on the minimum start times of actor $v_{a,1}$. We have $\hat{s}_{a,1}^p(k) = \hat{s}_{a,1}^p + k \cdot P_a$ and $\check{s}_{a,1}^p(k) = \check{s}_{a,1}^p + k \cdot P_a$ in which $\hat{s}_{a,1}^p$ and $\check{s}_{a,1}^p$ are defined as follows:

$$\begin{aligned}\hat{s}_{a,1}^p &= \max_{0 \leq k < q_a} (\hat{s}_{a,1}(k) - k \cdot P_a) \\ \check{s}_{a,1}^p &= \min_{0 \leq k < q_a} (\check{s}_{a,1}(k) - k \cdot P_a)\end{aligned}$$

Or equivalently by directly using the enabling times of the HSDF actors:

$$\begin{aligned}\hat{s}_{a,1}^p &= \max_{0 \leq k < q_a} (\hat{s}(v_{a,1}^k) - k \cdot P_a) \\ \check{s}_{a,1}^p &= \max_{0 \leq k < q_a} (\check{s}(v_{a,1}^k) - k \cdot P_a)\end{aligned}$$

With these definitions we have periodic upper and lower bounds on the schedules: $\forall_{k \in \mathbb{N}} : \hat{s}_{a,1}^p(k) \geq \hat{s}_{a,1}(k)$ and $\forall_{k \in \mathbb{N}} : \check{s}_{a,1}^p(k) \leq \check{s}_{a,1}(k)$. We can then use a derivation of the enabling jitter of tasks which is similar to the derivation presented in Section 5.5.5 for HSDF models. The enabling jitter of task τ_a would then be equal to:

$$J_a = \hat{s}_{a,1}^p - \check{s}_{a,1}^p \quad (5.37)$$

The main idea of the conservative analysis flow is that the computation of the linearized response times does not require iterative fixed point computation. The linearized response time of a task is linear in its enabling jitter which means that, together with the linear relation between the enabling jitter and the enabling times, we can combine it with Algorithms 2 and 1 to obtain an algorithm that calculates the complete flow using an LP algorithm which has a polynomial time-complexity.

In this chapter we provide this linearization for a system with an SPP scheduler:

$$\hat{R}_i = \max_{q \in \mathbb{N}} (w_i(q) - (q-1) \cdot P_i) \quad \text{with} \quad (5.38)$$

$$w_i(q) = q \cdot C_i + \sum_{j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{P_j} \right\rceil \cdot C_j \quad (5.39)$$

$$a = f(a) \leq \tilde{a} = \tilde{f}(\tilde{a}) \text{ when } \forall_x : f(x) \leq \tilde{f}(x) \wedge f(0) > 0$$

Figure 5.5: Comparison of the fixed-points of two monotonic increasing functions $f(x)$ and $\tilde{f}(x)$.

We use the fact that an over-approximation of the interference caused by higher tasks leads to a conservative upper bound $\tilde{w}_i(q)$ on $w_i(q)$. As is illustrated in Figure 5.5, then also the fixed-point calculated for $\tilde{w}_i(q)$ is conservative to the fixed-point of $w_i(q)$.

We first define this upper bound $\tilde{w}_i(q)$ on $w_i(q)$ by bounding $\lceil x \rceil$ with $(x + 1)$:

$$\begin{aligned} \forall_{q \in \mathbb{N}} w_i(q) &\leq \tilde{w}_i(q) \quad \text{with} \\ \tilde{w}_i(q) &= q \cdot C_i + \sum_{j \in hp(i)} \left(\frac{J_j + \tilde{w}_i(q)}{P_j} + 1 \right) \cdot C_j \end{aligned} \quad (5.40)$$

This upper bound $\tilde{w}_i(q)$ can be written as an equation linear in q and linear in the enabling jitters of the tasks. We use $\alpha_i = \sum_{m \in hp(i)} \frac{C_m}{P_m}$ as a shorthand notation.

$$\begin{aligned} \tilde{w}_i(q) &= q \cdot C_i + \sum_{j \in hp(i)} \left(\frac{J_j}{P_j} + 1 \right) \cdot C_j + \tilde{w}_i(q) \cdot \alpha_i \\ \tilde{w}_i(q) \cdot (1 - \alpha_i) &= q \cdot C_i + \sum_{j \in hp(i)} \left(\frac{J_j}{P_j} + 1 \right) \cdot C_j \\ \tilde{w}_i(q) &= \frac{q \cdot C_i + \sum_{j \in hp(i)} \left(\frac{J_j}{P_j} + 1 \right) \cdot C_j}{1 - \alpha_i} \\ \tilde{w}_i(q) &= q \cdot \frac{C_i}{1 - \alpha_i} + \sum_{j \in hp(i)} \frac{J_j \cdot \frac{C_j}{P_j} + C_j}{1 - \alpha_i} \end{aligned} \quad (5.41)$$

We can substitute this upper bound $\tilde{w}_i(q)$ in Equation 5.4 to define and upper

bound $\tilde{R}_i$ on the maximum response time of task τ_i :

$$\begin{aligned}\tilde{R}_i &= \max_{q \in \mathbb{N}} (\tilde{w}_i(q) - (q-1) \cdot P_i) \\ \tilde{R}_i &= \max_{q \in \mathbb{N}} \left(q \cdot \frac{C_i}{1-\alpha_i} + \sum_{j \in hp(i)} \frac{J_j \cdot \frac{C_j}{P_j} + C_j}{1-\alpha_i} - (q-1) \cdot P_i \right) \\ \tilde{R}_i &= \max_{q \in \mathbb{N}} \left(q \cdot \left(\frac{C_i}{1-\alpha_i} - P_i \right) + \sum_{j \in hp(i)} \frac{J_j \cdot \frac{C_j}{P_j} + C_j}{1-\alpha_i} + P_i \right)\end{aligned}\quad (5.42)$$

The equation inside the *max* expression of Equation (5.42) is strictly increasing in q if:

$$\frac{C_i}{1-\alpha_i} > P_i \quad (5.43)$$

If Equation 5.43 holds, $\tilde{R}_i$ thus would become infinite. This means that τ_i is not schedulable. If Equation 5.43 does not hold, the equation inside the *max* expression of Equation (5.42) is non-increasing which means that the response time is finite and can be found by choosing q equal to 1. In fact the upper bound on the response time of τ_i is in that case equal to $\tilde{w}_i(1)$ and no iterative fixed-point computation is required to determine it. This removes the inner iteration loop of the analysis flow as defined in Figure 5.1. The upper bound on the response time which we use in the remainder of this section, is linear in the enabling jitters of the task and is defined as follows.

$$\tilde{R}_i = \sum_{j \in hp(i)} J_j \cdot \frac{\frac{C_j}{P_j}}{1-\alpha_i} + \frac{C_i + \sum_{j \in hp(i)} C_j}{1-\alpha_i} \quad (5.44)$$

To determine if a task set is schedulable, it is sufficient to verify whether Equation 5.43 does not hold for every lowest priority task on each processor. If the task set is schedulable we can use the upper bounds on the response times of tasks to define an analysis algorithm with a polynomial time complexity. We use $\tilde{R}_i$ as defined in Equation 5.44 as the maximum response time of task τ_i and B_i as the minimum response time of task τ_i .

Similarly as the method described in Section 5.5.4 we use two instances of the dataflow model of the application, a best-case dataflow model and a worst-case dataflow model. We use the minimum and maximum response times of task τ_i as the firing durations of actor v_i in the best and worst-case schedule respectively.

We again use an LP algorithm to calculate the start times of the best- and worst-case schedule. The chosen minimum response time of a task does not dependent on the jitters of tasks. We can therefore use Algorithm 1 to compute the minimum start times of the tasks which are constant throughout the remainder of this section. If a minimum response time is used that is dependent on the jitters of tasks we could have linearized it similarly as the maximum response time and

we could include the determination of the minimum schedule in the algorithm below. However, this is out of the scope for this chapter.

We use a similar algorithm as Algorithm 2 to calculate the maximum start time of tasks. However, we use the fact that $\tilde{R}_i$ is linear in the jitters of the tasks. Because of this linearity we can include the jitters in the LP algorithm which makes the outer iteration loop of the analysis flow, as defined in Figure 5.1, redundant.

Algorithm 3 is used to compute the start times of the upper bound on the worst-case schedule. Compared to Algorithm 2, constraint (5.23) is changed to constraint (5.46) by using Equation (5.44) and moving the part of the equation that is dependent on the jitter to the left side of the constraint. Furthermore, constraint (5.47) is added. Note that the helper functions, β and φ , do not contain any variable and thus are constant.

Algorithm 3

Minimize

$$\sum_{v_i \in V} \hat{s}_i$$

Subject to

$$\hat{s}_s = s(\tau_s) \quad (5.45)$$

$$\forall e_{ij} \in E : \hat{s}_j - \hat{s}_i - \sum_{k \in hp(i)} J_k \cdot \beta(i, k) \geq \varphi(i) - \delta(e_{ij}) \cdot P_i \quad (5.46)$$

$$\forall v_i \in V : J_i = \hat{s}_i - \check{s}_i \quad (5.47)$$

$$\text{With } \beta(i, k) = \frac{\frac{C_k}{P_k}}{1 - \alpha_i} \quad \text{and} \quad \varphi(i) = \frac{C_i + \sum_{k \in hp(i)} C_k}{1 - \alpha_i}$$

5.7 BUFFER SIZING

In this section we provide methods to determine the sizes of the buffers such that the temporal constraints can be met. These buffer sizes are determined after the enabling characterizations of the tasks have converged. Existing buffer sizing algorithms for dataflow graphs, such as [MBGS10], can not be applied because they choose arbitrary start times for the actors such the buffers are minimal. These arbitrary start times can lead to changes in the enabling characterizations compared to the characterizations that are calculated in steps 1, 2, 3 and 4 of the analysis flow. These changes can potentially lead to increased interferences which invalidates the temporal results that are obtained with the analysis flow. This is of course not allowed.

We first present a method to determine sufficient buffer sizes which do not invalidate the temporal results. This method chooses the buffers such that the

periodic schedules as defined in step 3 in the analysis flow remain admissible. Second, we present a buffer minimization algorithm similar to the algorithm of [MBGS10]. This algorithm is possible because we base it on the linearized version of the analysis flow as presented in Section 5.6.

5.7.1 SUFFICIENT BUFFER SIZES

Because of monotonicity of functional deterministic timed dataflow graphs, we know that a larger number of tokens can never lead to later enabling times of actors. This holds also the other way around; a smaller number of tokens can never lead to earlier enabling times of actors. The lower bound on the best-case schedule as defined in Section 5.5.4 basically assumes an infinite number of tokens on each edge that may contain initial tokens. Lowering this number of initial tokens can never lead to earlier enabling times. This means that the determined periodic lower bound is still a lower bound on the best-case scheduling of the SDF model after choosing a number of initial tokens that is smaller than infinity.

Therefore, for each edge e_{ij} in the SDF model, the number of tokens $\delta(e_{ij})$ is sufficient if it does not delay the periodic upper bound on the worst-case schedule of the dataflow model. This can be ensured by determining the amount of tokens such that the maximum start times as calculated with Algorithm 2 are feasible. We have to guarantee that for the determined start times it holds that for each SDF edge e_{ij} the following constraint is true:

$$\forall_{k \in \mathbb{N}} : \hat{s}_j(k + \delta(e_{ij})) \geq \hat{s}_i(k) + \hat{\rho}_i \quad (5.48)$$

The start time of an SDF actor v_i is periodic in q_i firings. We can therefore use the least common multiple, lcm , of the repetition factors of the actors on an edge to find the possible interleavings. When we consider $lcm(q_i, q_j)$ firings of actors v_i and v_j on edge e_{ij} , we have that all the possible interleavings are considered. We can thus ensure Equation 5.48 by choosing $\delta(e_{ij})$ equal to:

$$\delta(e_{ij}) = \min \left(\left\{ \delta \in \mathbb{N} \mid \forall_{0 \leq k < lcm(q_i, q_j)} : \hat{s}_j(k + \delta) \geq \hat{s}_i(k) + \hat{\rho}_i \right\} \right) \quad (5.49)$$

When the task graph can be modeled using an HSDF model, we can simplify this method. For HSDF models we know that the repetition factor q_i of each actor v_i is equal to 1. Furthermore, the actors in the HSDF model fire strictly periodic and thus: $\hat{s}_i(k) = \hat{s}_i(0) + k \cdot P_i$. These properties can be used to simplify Equation 5.49 for HSDF models to:

$$\begin{aligned} \delta(e_{ij}) &= \min \left(\left\{ \delta \in \mathbb{N} \mid \hat{s}_j(\delta) \geq \hat{s}_i(0) + \hat{\rho}_i \right\} \right) \\ &= \min \left(\left\{ \delta \in \mathbb{N} \mid \hat{s}_j(0) + \delta \cdot P_j \geq \hat{s}_i(0) + \hat{\rho}_i \right\} \right) \\ &= \min \left(\left\{ \delta \in \mathbb{N} \mid \delta \geq \frac{\hat{s}_i(0) + \hat{\rho}_i - \hat{s}_j(0)}{P_j} \right\} \right) \\ &= \left\lceil \frac{\hat{s}_i(0) + \hat{\rho}_i - \hat{s}_j(0)}{P_j} \right\rceil \end{aligned} \quad (5.50)$$

5.7.2 BUFFER MINIMIZATION

The linearized analysis flow as presented in Section 5.6 can be used to calculate minimum sufficient buffer capacities when the task graph can be modeled using an HSDF model. We again choose the best-case schedule independent of the jitter of the tasks by only using the BCET of tasks. The earliest start times of the first firings can be calculated using Algorithm 1. Algorithm 4 is used to calculate minimum sizes for the buffers which do ensure that a periodic schedule exists.

Minimizing the amount of tokens does not lead to the smallest possible maximum start times and does also not lead to minimum jitters. However, Algorithm 4 takes the effect that increasing jitters has on the maximum response time of tasks, into account and uses the available freedom to find a minimum amount of tokens that ensures the temporal constraints.

The number of tokens δ , should be integer. If this is enforced in the algorithm, Algorithm 4 becomes an ILP algorithm which has a worst-case non-polynomial time-complexity. Thanks to the monotonicity property of dataflow graphs, sufficient buffer sizes can also be found by keeping $\delta(e)$ a real-value and by rounding $\delta(e)$ up to nearest integer value after the execution of the algorithm finishes. This potentially leads to pessimistic buffer sizes however they can be determined with a polynomial time-complexity.

Algorithm 4

Minimize

$$\sum_{e_{ij} \in E} \delta(e_{ij})$$

Subject to

$$\hat{s}_s = s(\tau_s) \quad (5.51)$$

$$\forall_{e_{ij} \in E} : \hat{s}_j - \hat{s}_i - \sum_{k \in hp(i)} J_k \cdot \beta(i, k) \geq \varphi(i) - \delta(e_{ij}) \cdot P_i \quad (5.52)$$

$$\forall_{v_i \in V} : J_i = \hat{s}_i - \check{s}_i \quad (5.53)$$

$$\text{With } \beta(i, k) = \frac{\frac{C_k}{P_k}}{1 - \alpha_i} \quad \text{and} \quad \varphi(i) = \frac{C_i + \sum_{k \in hp(i)} C_k}{1 - \alpha_i}$$

Example 5.1:

Figure 5.6(a) shows the task graph of an application which consists of four tasks. The application is executed on two processors which both have a Static Priority Preemptive scheduler. The first processor executes tasks τ_a (with the highest priority) and τ_d (lowest priority). The other processor executes the other two tasks. Task τ_c has the highest priority on this processor and task τ_d the lowest priority. Tasks τ_a , τ_c and τ_d have a constant execution time of 1 time unit and task τ_b has an execution time between 2 and 4 time units. The objective is to find minimum buffer sizes

Figure 5.6: Buffer minimization example.

for the buffers c_{ad} (denoted with x) and c_{dc} (denoted with y) which guarantee the throughput constraint imposed by the period of 6 time units of task τ_a .

The dataflow graph that models the temporal behavior of the application is shown in Figure 5.6(b). The number of required initial tokens on edge e_{da} (denoted with x) corresponds with the required size of c_{ad} . The number of initial tokens on e_{cd} (denoted with y) corresponds with the size of c_{dc} . We first use this dataflow model with the analysis method presented in Section 5.2 to find bounds on the possible schedules of the tasks. After convergence of the analysis flow we find the following start times, jitters and response times:

	v_a	v_b	v_c	v_d
$\check{s}$	0	1	3	1
$\hat{s}$	0	1	7	1
J	0	0	4	0
$\hat{R}$	1	6	1	2

Using these start times and response times we can compute sufficient buffer sizes with Equation 5.50: $x = 1$ and $y = 2$.

When we use the method of section 5.7.2 we compute the following start times, jitters and response times:

	v_a	v_b	v_c	v_d
$\check{s}$	0	1	3	1
$\hat{s}$	0	1	8	3
J	0	0	5	2
$\hat{R}$	1	7	1	$\frac{12}{5}$

With Algorithm 4 we compute that a size of 1 location is sufficient for both the buffers c_{ad} and c_{dc} ($x = 1$ and $y = 1$). Despite of the fact that the jitters and worst-case response times are over-approximated, the computed buffer sizes are less. Note

Figure 5.7: Two streams scheduled with a round robin scheduler.

that the algorithm uses the freedom in the worst-case start time of actor v_d to find smaller buffers.

5.8 EVALUATION

In this section we evaluate the presented analysis approach. We give four examples of the use of the analysis approach and show some of the consequences regarding the differences with other temporal analysis methods.

The first example is shown in Figure 5.7. It shows a snippet of an application which contains two tasks, τ_a and τ_b . Both these tasks process a periodic stream of data with a different period (10 and 100 time units respectively). The tasks share a resource which is scheduled using a round-robin scheduler. The WCET of each task is shown above the vertex. Traditional dataflow analysis would use a response time for a task equal to the sum of the WCETs. Using this analysis, the actor corresponding to a task can only fire once per response time. The response time of τ_a is equal to 15 time units which means that its throughput constraint (execute once per 10 time units) cannot be met.

If we use the analysis method provided in this chapter, we will find maximum response times for both the tasks equal to 15 time units. The analysis method uses the fact that the interference of τ_b on τ_a is sporadic and that on average τ_a can execute with a period of 10 time units. With the analysis method of this chapter we thus can conclude that the throughput constraints can be met. Note that to determine the response time of task τ_b the structure of the round-robin scheduler is used. We use the in Section 5.4.3 presented bound on the interference of tasks to find out that task τ_a can interfere only once during an execution of task τ_b .

The second example considers the linearization approach presented in this chapter. This linearization can be used to speed up the temporal analysis of applications. However, it over-approximates jitter which can in some cases result in very conservative results. This is because jitter is also used in the definition of busy periods and thus also in response times of the tasks. An increase in the

Figure 5.8: Example of a topology for which linearization of the response time can become problematic 5.8(a) and corresponding normal and linearized maximum response times of the low-priority task 5.8(b).

response time of a task can again lead to an increase of the jitters which can result in additional over-approximations.

Figure 5.8(a) illustrates this issue. The interference from the high-priority task (HP) on the low-priority task (LP) is over-approximated when linearization is used. Figure 5.8(b) shows the response times of the LP task for different periods of the source. Decreasing the period leads to a larger difference between the normal maximum response time ($\hat{R}_{LP}$) and the linearized maximum response time ($\hat{R}_{LP}(l)$) and eventually leads to an unnecessary conclusion of infeasibility. Note that this effect does not occur in the case of starvation-free schedulers. This is because when such a scheduler is used, the response time of tasks can be defined independent of the execution rate of the other tasks [WBS09]. Therefore, no circular resource dependencies between tasks have to be taken into account which prevents high inaccuracies in case of linearized response times.

The next example considers bursts in the application. Such bursts can lead to inaccurate temporal analysis results. The end-to-end latency is in fact often computed using a notion of response time that next to the processing time also includes the time that data resides in the queues, i.e., the queuing time. Because in a path, often a burst can happen at the input of each task, the local worst-case response time of each of the tasks in the path becomes large due to the worst-case queuing time in each queue. However, in case of a burst, the same event that gets processed along a path does not experience this delay at each task.

The temporal analysis method we propose uses dataflow analysis techniques. Therefore, the correlation of events is preserved and the time that data resides in the buffer is analyzed using the dataflow model and does therefore not need to be taken into account in the response time of a task. The end-to-end latency can be calculated using the schedules of the dataflow graph which use the actual enablings of actors. This means that we do not accumulate times that data resides in the different buffers. Only the buffer time at the beginning of the path needs to be taken into account. This means that our analysis method thus has the property that bursts only have to be taken into account once for the latency of a path. This phenomenon is known as the pay-bursts-only-once property [BT01].

Figure 5.9: Illustration of burst in an application.

An example of burst in an application is illustrated in Figure 5.9. The graph consists of two tasks, τ_a and τ_b . The incoming queue of task τ_a contains 3 tokens to indicate that a burst of 3 events occurred. Task τ_a has a response time between 0 and 5 time units and task τ_b has a constant response time equal to 5 time units. The worst-case schedule of this task graph is shown in Figure 5.9(b). The downward-pointing arrows indicate the start of executions and the finish of an execution is indicated with an arrow pointing upwards. A simultaneous start and finish of two executions is depicted by a bidirectional arrow. In the schedule of Figure 5.9(c) the second and third executions of task τ_a are assumed to take 0 time units, equal to the minimum response time of the task. The tuple above the arrow at time 5 indicates the amount of executions that are started and finished respectively at time 5. In this schedule, a burst of 3 events happens at c_{ab} because 3 tokens are produced at time 5. However, despite of this burst at c_{ab} , the worst-case production times of τ_b are in Figure 5.9(c) equal to the production times in Figure 5.9(b). The worst-case latency from begin to end is thus not influenced by the presence of burst behavior on c_{ab} . In fact, the worst-case latency of a path can be determined using the worst-case schedule only. This is because of monotonicity of dataflow graphs which are used during the analysis. This monotonicity tells us that producing tokens earlier can never lead to any later finish time of any actor in the graph.

When the end-to-end latency is determined using response times that include the queuing time, the burst of 3 events was included in the response times of both tasks τ_a and τ_b . It would then thus be added to the end-to-end latency, twice.

The fourth example is shown in Figure 5.10. This figure shows an example of a topology for which the temporal analysis suffers from inaccuracies if the

Figure 5.10: Example of inaccuracy in the temporal analysis if the correlation between c_{ac} and c_{bc} is lost.

correlation between streams is lost. The example has three tasks and a periodic source. Task τ_a has a response time between 0 and 5 time units and task τ_b has a constant response time equal to 5 time units. With our analysis flow we conclude that task τ_c has a minimum start time equal to its maximum start time. The enabling jitter of task τ_c is thus equal to 0. Methods which can not correlate streams use the maximum of the arrival jitters on the different incoming edges as the enabling jitter [HHJ⁺05]. In this case they would conclude that the enabling jitter of task τ_c is equal to 5 which is inaccurate.

5.9 CASE-STUDY

The temporal analysis flow of this chapter is demonstrated using two case-studies.

In the first case-study we analyze the fine frequency tracking loop of a radio processing applications. This fine frequency tracking loop contains a cyclic data dependency in the form of a feedback loop as well as a cyclic resource dependency. Furthermore, this case-study compares the in this chapter used generic enabling characterization with an enabling jitter based characterization.

The second case-study contains a Frequency Modulation (FM) receiver and a Digital Audio Broadcasting (DAB) receiver which share resource with the periods of the sources of the two receivers being different. We demonstrate that by using the analysis flow of this chapter, the difference in periods can be modeled accurately. The shared resource is scheduled using a starvation-free run-time scheduler. Therefore, this case-study also shows that the temporal analysis results of systems with starvation-free run-time schedulers can be improved with the temporal analysis flow of this chapter.

5.9.1 FINE FREQUENCY TRACKING LOOP

Radio processing applications often have feedback loops such as time and frequency synchronization loops, automatic gain control loops and loops for re-

Figure 5.11: Task graph of a radio processing application with a fine frequency tracking loop.

encoding and adaptive channel equalization. These feedback loops add cyclic constraints which complicates the analysis of such applications.

Figure 5.11 contains the task graph of an example of a radio processing application with a fine frequency tracking loop. The feedback loop is formed by the demodulation task and the mixer task. The demodulation task determines the amount of frequency offset and the mixer task compensates for this offset. The tokens in the feedback loop behave as delays which means that more tokens in this loop result in a reduced adaptation speed of the tracking loop which is undesirable. To illustrate the effect of multi-rate behavior we choose a configurable block size of k items that are processed at once by the mixer task. The ADC source task also produces the same amount of items at once. The period of the ADC source also depends on the block size and is equal to $k \cdot 16$ time units.

The radio processing application is executed on a MPSoC. The processors of this MPSoC use Static Priority Preemptive schedulers to schedule the tasks. The ADC source does have dedicated hardware support in the MPSoC. The red tasks (mixer and demodulation) are mapped to one processor of the MPSoC and the blue tasks (deinterleaving and frame-formatting) are mapped to another processor. The channel decoding task is mapped to a dedicated hardware accelerator. The demodulation, deinterleaving and frame-formatting all process at a granularity of one item. The channel decoding task uses redundancy in the bit stream to correct errors caused by a noisy channel. For example the Viterbi algorithm can be used to decode the input stream. For this case-study the redundancy in the bit stream has a ratio of two to one which means that the channel decoding task produces one corrected item per two consumed items.

In this case-study we focus on the most interesting part of this application, namely, the fine frequency tracking loop. The mixer task and the demodulation task share one processor and the demodulation task is configured with the highest priority. The mixer task has the lowest priority. Due to the priorities of the tasks, this part of the radio processing application has a cyclic resource dependency, i.e.,

the mixer task delays the demodulation task which in turn preempts the mixer task. Because of the feedback loop, this part of the application thus has both a cyclic resource as well as a cyclic data dependency.

Figure 5.12(a) shows the SDF model of the part of the application on which we focus. The ADC source is modeled with actor v_s and the mixer and demodulation tasks are modeled with actor components $(v_{a,0}, v_{a,1})$ and $(v_{b,0}, v_{b,1})$ respectively. The BCET and the WCET of the demodulation task are equal to $k \cdot 2$ and $k \cdot 4$ time units respectively and the BCET and WCET of the mixer task are equal to 4 and 6 time units respectively. The source is assumed to produce instantaneously. Figure 5.12(a) shows the firing durations belonging to the worst-case temporal model of the application.

The equivalent HSDF model of the SDF model is shown in Figure 5.12(b) for which the block size, k , is chosen to be equal to 3. The repetition factors of the SDF actor component v_s , v_a and v_b are then equal to 1, 1 and 3 respectively. Therefore, v_b is modeled with 3 HSDF actor components and the other two SDF actor components are both modeled with one HSDF actor component. The period of the source actor, v_s , is equal to $16 \cdot 3 = 48$ time units. The periods of v_a and v_b are thus equal to 48 and 16 time units respectively.

After three iterations of the analysis flow, the enabling characterizations have converged and the analysis flow ends. The conclusion is that the number of tokens in the feedback loop, 6 when k equals 3, are sufficient to meet the throughput, i.e., the ADC source can execute once per 48 time units. If less tokens are chosen in the feedback loop, the accuracy of the frequency compensation is better. However, from the analysis flow we conclude that with less than 6 tokens, the throughput cannot be met. The response time of the mixer task is after the first iteration of the flow equal to 30 time units, after the second iteration 42 time units and it converges in the third iteration to 48 time units.

Each iteration of the analysis flow determines the schedules of the HSDF model. Figure 5.12(d) shows these schedules of actor $v_{b,1}$ in the last iteration of the analysis flow when k is equal to 3. With the schedules, the enabling characterization is determined using Equations (5.30) and (5.1). This enabling characterization, $\hat{\eta}_b$, is shown in Figure 5.12(d) for task τ_b (the demodulation task) with the red solid line. Note that this enabling characterization is multi-dimensional periodic which cannot be expressed with a period and jitter characterization accurately. The blue dashed line in Figure 5.12(d) illustrates the jitter based enabling characterization from Section 5.5.5, denoted by $\hat{\eta}_b^j$. The drawn characterization is obtained by using the jitter based enabling characterization in all of the iterations of the analysis flow. As is shown in Figure 5.12(d), the interference is higher according to this jitter based enabling characterization which results in larger response times.

Table 5.1 illustrates this in more detail. It contains the different worst-case response times of the mixer task given different block sizes k . The second column contains the response times obtained by using the proposed enabling characterization. The third column shows the converged response times when an enabling

(a) Simplified SDF model

(b) Equivalent HSDF model for $k = 3$ (c) Dataflow schedule of actor v_b for $k = 3$ (d) Maximum number of enablings of actor v_b for $k = 3$

Figure 5.12: Simplified SDF model and corresponding HSDF model of a radio processing application with a fine frequency tracking loop. The analysis results are shown for actor v_b .

Table 5.1: Comparison between the worst-case response times of the mixer task obtained by the proposed enabling characterization and the jitter based characterization.

k	$\hat{R}_{mixer}$	$\hat{R}_{mixer}^j$	difference
1	16	16	+ 0.0%
2	32	44	+ 37.5%
3	48	78	+ 62.5%
4	64	106	+ 65.6%
8	128	230	+ 79.7%
16	256	472	+ 84.4%
32	512	956	+ 86.7%

characterization based on enabling jitters is used and the fourth shows the difference between the two. When the block size is equal to 1, the results obtained with the enabling jitter characterization are equal to the results obtained with the proposed enabling jitter. However, for larger block sizes the difference rapidly increases. For a block size of 32 items, the worst-case response time obtained with the enabling jitter characterization is 86.7% larger than the response time that is calculated by using the proposed enabling characterization.

Over-approximating the worst-case response times by using an enabling jitter based characterization can be even less desirable because it can also have an effect on the temporal results of the other tasks in the application. The tasks later in the pipeline than the mixer task, for example the deinterleaving task, will observe a larger difference between the best-case and the worst-case schedule when the worst-case response time of the mixer task is larger. With a larger difference between the best-case and the worst-case schedule, the interference bounds will also increase. This can again lead to larger response times of the tasks that share a processor with such a task.

5.9.2 FM AND DAB RECEIVER

Figure 5.13 shows the task graphs of two applications, a FM receiver and a DAB receiver. Both applications share one DSP core which executes both demodulation stages. This DSP uses a round-robin scheduler to schedule the tasks at run-time. For simplicity, the DAB demodulation task and the Data Out task are always scheduled together. For clarity we only present the demodulation stage of the applications.

The sample rates of the Analog-to-Digital Converters (ADCs) differ in both applications because the bandwidth requirements are different (sample rate of 2048kHz for DAB and 320kHz for FM). Furthermore, the demodulation task in

Figure 5.13: Example of an FM and a DAB application sharing one DSP.

the DAB application performs an FFT of size 2048 which means that the DAB demodulation task uses blocks of 2048 samples while the FM demodulation uses much smaller blocks of 8 samples.

These differences in the sample rates and the block sizes result in a different enabling rate of the two demodulation tasks. The FM demodulation task gets enabled with a period of $\frac{\text{blocksize}}{320\text{kHz}} = 25\mu\text{s}$. The DAB demodulation task is enabled with a period of 1.246ms which includes the symbol period of $\frac{\text{blocksize}}{2048\text{kHz}} = 1\text{ms}$ plus the guard interval of 0.246ms for DAB Transmission Mode I [ETS06]. The execution times of the tasks are assumed to be constant. The FM demodulation takes 15μs and DAB demodulation and Data Out task together take 450μs.

We now want to verify whether both applications meet their throughput constraint and how much buffering is required between the ADCs and the demodulation tasks. Traditional methods for analyzing systems with round-robin schedulers use the fact that for one execution of a task, the task has to wait for at most one execution of all the other tasks. However because no traffic characterization is taken into account, they assume that this interference of other tasks happens during all executions. In that case, one would conclude that the FM demodulation can only execute once per 465μs which means that it can not meet its throughput constraint.

The fix for this problem that is currently used, is executing the FM demodulation task 50 times in a row. 50 Executions of the FM demodulation task are grouped together which means that the DAB demodulation task interferes at most once per 50 executions. The 50 executions take 750μs and have a period of 1250μs. With this grouping, the traditional analysis method concludes that both the applications can meet their throughput. The response time of the FM demodulation task is equal to 1.2ms and also the response time of the DAB demodulation and Data Out task together is equal to 1.2ms. The FM demodulation can start after the corresponding ADC has produced $50 \cdot 8$ samples, 1.25ms after

the source starts. Using the method of Section 5.7 we conclude that a buffer size of $2 \cdot 50$ times the blocksize is sufficient for the buffer between the FM ADC and the FM demodulation task ($2 \cdot 50 \cdot 8$ samples). The DAB demodulation starts after the ADC has produced 2048 samples, 1ms after the sources starts. Two times the blocksize ($2 \cdot 2048$ samples) is thus sufficient for the buffer between the DAB ADC and the DAB demodulation task.

The problem can be solved more accurately and elegantly with the analysis method that is described in this chapter. With this analysis method it is not needed to group the executions of the FM demodulation task together because the actual enablings are taken into account. The analysis methods described in Section 5.2 conclude that the throughput constraints can be met. The worst-case response times of both the demodulation tasks are equal to $465\mu\text{s}$ which is much less than the response times of the grouping solution (61.25% less). The buffer sizes can be determined using the method presented in Section 5.7.1. The FM demodulation task can start $25\mu\text{s}$ after the source starts (8 samples) and using Equation 5.50 we conclude that the size of the buffer between the FM ADC and FM demodulation needs to be larger than $\frac{25+465-0}{25}$ times the blocksize. A buffer size of 20 times the blocksize ($20 \cdot 8$ samples) is thus sufficient for this buffer. This is a reduction of 80% compared to the solution of the previous paragraph. The buffer between the DAB ADC and the DAB demodulation task needs to be larger than $\frac{1000+465-0}{1246}$ times the blocksize. A size of 2 times the blocksize ($2 \cdot 2048$ samples) is thus sufficient for this buffer. The size of this buffer is equal to the buffer size that was determined in the previous paragraph.

5.10 CONCLUSION

In this chapter we introduced a temporal analysis method which can be used to analyze systems with non-starvation-free schedulers using dataflow analysis techniques for SDF graphs. Single processor as well as multiprocessor systems can benefit from the methods presented in this chapter. Because the actual data dependencies between tasks are used, a higher accuracy of the analysis results can be obtained than is possible when tasks are assumed to be independent.

The presented analysis techniques do not make assumptions on the graph topology and do therefore allow cyclic graph topologies. Also the combination between cyclic resource dependencies and cyclic data dependencies is supported which is unique. Next to that, the correlation between different event streams is preserved which can lead to accuracy improvements compared to related analysis approaches. Furthermore, the analysis method allows to take buffer capacity constraints into account during the analysis and can be used to determine sufficient buffer capacities afterwards.

Upper and lower bound schedules are used to determine enabling characterizations. The enabling characterization allows more modeling freedom than characterizations based on periods and enabling jitters. This is illustrated in the case-study. Also a linearized variant of the analysis flow is presented. This linear

variant does use the enabling jitter of tasks and an upper bound on their response times to enable an analysis flow that has a polynomial time-complexity for HSDF graphs. This conservative analysis method can therefore be used to explore the design space faster. Next to that, it can be used to minimize buffer capacities.

We expect that the presented throughput analysis approach based on enabling characterizations can be generalized, enabling the accurate analysis of systems with non-starvation-free schedulers in combination with more expressive data-flow models, such as CSDF [BEP96] and VRDF [WBS08a] models.

Compositional Temporal Analysis Model

ABSTRACT – *The incremental design and analysis of parallel hard real-time stream processing applications is hampered by the lack of an intuitive compositional temporal analysis model that supports arbitrary cyclic dependencies between tasks.*

This chapter introduces a compositional temporal analysis model for hard real-time systems, called the Compositional Temporal Analysis (CTA) model. In the CTA model, arbitrary cyclic dependencies can be specified. The CTA model also supports hierarchical composition and incremental design of timed components. Hiding the internal structure of components without changing the temporal properties is also possible. Furthermore, we show in this chapter that the CTA model can be used as an abstraction of timed dataflow models and that efficient analysis algorithms exist for the CTA model.

The previous chapters discussed abstractions for aperiodic multiprocessor systems based on dataflow analysis models. These abstractions focus mainly on accurate and broadly applicable methods for the temporal analysis of stream processing applications. Methods that give insight in the temporal behavior of these applications are very important. However, the design and verification of such stream processing applications is currently hampered by the lack of suitable compositional temporal analysis models. As a result, an incremental design style to reduce the complexity of the design and the analysis of these systems is hard to apply. Ideally, such an incremental design style would allow the grouping of

The material in this chapter is based on work presented in [HGWB12].

Figure 6.1: Composition of SDF actors.

components into subsystems that are characterized in isolation without loss of accuracy.

A limitation of dataflow models is that they are not compositional in general. A model is compositional if the properties of a composition of components can be deduced from the properties of the individual components without knowing their internal hierarchy. In the SDF model for example, composition is not always possible [JSL95, TBG⁺13] because deadlock freedom and token rate consistency of an SDF graph can only be checked if the SDF graph contains no hierarchy.

Figure 6.1, taken from [TBG⁺13], illustrates that the SDF model is not compositional. In Figure 6.1(a) an SDF graph is shown that is deadlock free because there are always sufficient tokens in one of the queues to fire one of the actors. However, when actors A and B are composed into an actor P , an issue with defining the rate at which actor P transfers tokens arises. Using consistent rates, which guarantee that there is no infinite accumulation of tokens on the edges, gives the SDF graph shown in Figure 6.1(b). However, this graph deadlocks because the numbers of initial tokens are insufficient. Even initially, no actor is enabled.

In this chapter we introduce the CTA model. We show that an abstraction can be made from an SDF graph to a CTA model in which hierarchical composition of components can be performed and incremental design is supported. It is also shown that the CTA model can model latency constraints and strictly periodic sources and sinks. The CTA model also supports arbitrary cyclic dependencies between components.

The outline of this chapter is as follows. The basic idea behind the CTA model is presented in Section 6.1. Section 6.2 describes the CTA component model in detail. Composition of CTA components is discussed in Section 6.3. The use of the CTA model is illustrated in Section 6.4. Related work is discussed in Section 6.5 and the conclusions are presented in Section 6.6.

6.1 BASIC IDEA

In this section we provide an informal introduction to the CTA model which is formalized in subsequent sections.

Figure 6.2: Example of a CTA component.

Components in the CTA model consist of ports and directed connections between ports. Ports transfer data at a current rate r which is bounded by a maximum rate $\hat{r}$. Connections introduce a total delay of Δ which depends on the current transfer rate of the connection. This delay specifies the time it takes for data to go through the connection. When ports are connected, their transfer rates are coupled by a fixed ratio which is specified by the connection. An example of such a component in the CTA model can be found in Figure 6.2. The example contains a CTA component A that consists of three ports which are connected by two connections. With the proposed CTA component description, we can conservatively model the periodic temporal behavior of dataflow graphs.

In Chapter 2 it has been shown that dataflow graphs can conservatively model the temporal behavior of streaming applications. Thanks to the monotonicity property of dataflow graphs, earlier production times, can not lead to worse temporal results. Therefore, these production times can be chosen conservatively. Furthermore, we call a dataflow graph to be temporally conservative to a task graph when every data item arrives earlier in the buffer than the corresponding token arrives in the queue of the dataflow graph.

We propose the CTA model as an additional level of abstraction in which the arrival of data is modeled temporally conservative (pessimistic) to the arrival of tokens in the dataflow graph by bounding a possible schedule of the dataflow graph with linear bounds. The CTA model is then temporally conservative to an application if the dataflow graph is also temporally conservative to the application. It is guaranteed that in the application data arrives earlier than is assumed during the temporal analysis of the CTA model. Furthermore, compared to temporal analysis of dataflow graphs, the CTA model adds composition and hiding possibilities.

Figure 6.3(a) contains a dataflow graph and Figure 6.3(c) shows the CTA model that corresponds to this graph. An actor in the dataflow graph is translated to a component where each incoming or outgoing queue of the actor becomes a port. Inside the component, all ports corresponding to an incoming queue are connected to all ports corresponding to an outgoing queue. The queues between actors in the dataflow graphs correspond with connections between components in the CTA model. In this example, the delays of the internal connections of the CTA components correspond with the firing durations of the corresponding actors. Initial tokens on a queue correspond with a negative, rate dependent, delay on the corresponding connection. This negative delay Δ represents data

Figure 6.3: Dataflow graph and the corresponding CTA model with the schedules used for temporal analysis.

that can initially be used, i.e. data can be used Δ time units before data is produced on the connection. For illustration we have chosen a transfer rate of $\frac{1}{4}$ token per time unit in Figure 6.3.

Figures 6.3(b) and 6.3(d) show that the delays of the CTA model in Figure 6.3(c), model the temporal behavior of the dataflow graph in Figure 6.3(a) conservatively. Figure 6.3(b) contains periodic schedules of actors E and F . The circles mark the consumption times of tokens and the squares mark the production times of tokens. On a queue, tokens can only be consumed after they are produced, which means that all the consumptions (circles) of F need to take place at a later time stamp than the corresponding productions (squares) of E . Purely for illustration purposes we have added time between these production and consumption times while they could occur at the same time.

The periodic schedules as shown in Figure 6.3(b) can be bounded by linear bounds which are illustrated with dashed and solid lines. Figure 6.3(d) contains only these linear bounds. The bound on the consumptions (dashed line) assumes earlier (or equal) consumption times of tokens and the production bound (solid line) assumes later (or equal) production times of tokens. We can use these linear bounds in the analysis because they assume later production times of data which leads to a conservative temporal analysis result (see Chapter 2). The CTA model of a dataflow graph is based on these linear bounds.

In the CTA model, periodic event sequences are used to express constraints. Such a periodic event sequence is an event sequence as used in Chapter 2 in which

the distance between each event is equal. These periodic event sequences are specified using an offset and a distance between events. The delays in the CTA model shift such a periodic event sequence over the time axis and thus change the offset. The delays in the CTA model are therefore chosen to be equal to the horizontal difference between the linear consumption and production bound on the schedules of the corresponding dataflow actors. This represents the maximum time between the consumption of data on one port and the production on the other port. The delay on the connection from F' to E' specifies the amount of initially available data, which can be computed with the number of initial tokens on the corresponding queue in the dataflow model. The time it takes to produce x tokens on a queue is in fact equal to x divided by the transfer rate of data on the connection. Actor E can transfer one token per four time units, i.e. the transfer rate is maximally $\frac{1}{4}$, and thus, component E' can start consuming events $\frac{-2}{1/4}$ time before F' starts producing events.

Connections in the CTA model can change the transfer rate with a fixed ratio. This can be seen as increasing or decreasing the distance between events of the periodic event sequence with a fixed amount of time.

Composition and hiding Figure 6.4(a) shows two CTA components, A and B . The ports of the components are drawn as small circles and are named p_x . The maximum rates of the ports are shown in the table below the components and are denoted by $\hat{r}$. When composing components, connections are added between the ports of the components. Arrows in this figure denote connections and the numbers written next to these arrows represent the delay introduced on the connection. The current rates of the ports connected by connections are coupled with a fixed transfer rate ratio which means that the maximum current rates of ports are adapted to the slowest port in the chain of connected ports. This transfer rate ratio is not shown in the figure, but can be found by dividing the maximum transfer rates ($\hat{r}$) from the table in Figure 6.4(a). Between p_1 and p_3 the transfer rate ratio is equal to $\frac{\hat{r}(p_3)}{\hat{r}(p_1)} = 2$ which means that the transfer rate is doubled. The transfer rate ratios of the other connections in Figure 6.4(a) are equal to 1.

Figure 6.4(b) shows the composition of components A and B into a component C where the connections between p_2 and p_4 and between p_3 and p_5 are added in the composition. The maximum current rates that are possible given the transfer rate ratios of the connections are shown in the table denoted by $\bar{r}$. As shown in this figure, the composition of components is again a component.

The ports that are connected in the composition can also be hidden without changing the characteristics of the component. This can be done by iteratively removing an internal port and creating a connection for each pair of ports which had a connection via this port. The delays of these new connections can be found by adding the delays of the original two connections together. The result of hiding all the internal ports of the composition of Figure 6.4(b) is component D

Figure 6.4: Components, composition and hiding in the CTA model. The tables below the models contain the visible ports, their maximum rates $\hat{r}$ and their maximum current rates $\bar{r}$.

which is shown in Figure 6.4(c). The delays of the resulting connections are equal to the sum of the delays of the original connections.

Consistency Not all compositions are possible. A composition must be consistent, which means that it must be able to meet the constraints imposed on the resulting component. Adding connections can have the result that a port is connected by multiple connections which means that there can be a conflict in the ratios enforced by the different connections. This type of consistency is similar to the consistency check for SDF graphs and it indicates accumulation of data at a lower level of abstraction.

It must be enforced that data becomes available in time. The delay of a connection specifies the time it takes for data to go through the connection. There can be cyclic connections between ports. This means that if the total time it takes for data to travel through such a cycle of connections is positive, data arrives too late. Because delays can be rate dependent, we can calculate maximum transfer rates for which all cycles have a negative cumulative delay. If such rates can

not be found, the composition is inconsistent. If a CTA model corresponds to an SDF graph, inconsistency of a composition usually indicates deadlock in that SDF graph.

6.2 COMPONENT MODEL

A component in the CTA model is defined as a tuple $V = (P, \hat{r}, C, \lambda, \phi, \epsilon)$. The ports of the component are specified by P . Each port has a strictly positive maximum transfer rate which is specified by $\hat{r} : P \rightarrow \mathbb{R}^+$. We use $r(p) \leq \hat{r}(p)$ as the current rate at which port p transfer data.

The set of connections between ports of the component is defined by $C \subseteq P \times P$. A connection $(p, q) \in C$ is directed from port p to port q and we use c_{pq} as a shorthand for (p, q) . For each connection in the component, a specification of the delay introduced on the connection is given by ϕ and ϵ , where $\epsilon : C \rightarrow \mathbb{R}$ specifies a constant delay on a connection and $\phi : C \rightarrow \mathbb{R}$ a rate dependent delay.

The transfer rates of ports of a connection (p, q) are coupled with a fixed ratio. This ratio is specified by $\lambda : C \rightarrow \mathbb{R}^+$. The current rate of port q of a connection (p, q) is coupled to the current rate of port p : $r(q) = \lambda(c_{pq}) \cdot r(p)$.

We define the transfer rate on a connection (p, q) to be equal to the transfer rate of the sending port: $r_c(c_{pq}) = r(p)$. The time that data is delayed over a connection (p, q) depends on the transfer rate of the connection and is equal to: $\Delta(c_{pq}) = \epsilon(c_{pq}) + \frac{\phi(c_{pq})}{r_c(c_{pq})}$. The total delay of a connection can be negative to specify that when the corresponding application starts, data is available at the connection.

We introduce periodic event sequences as the unit of computation in the CTA model. Such a periodic event sequence is an event sequence as used in Chapter 2 in which the distance between each event is equal. A periodic event sequence can be specified as a tuple (o, ℓ) . With o the offset of the event sequence and ℓ the distance between events. The time at which event n occurs in an event sequence (o, ℓ) is then equal to $\tau(n) = o + n \cdot \ell$. The rate r of such a periodic event sequence is equal to $\frac{1}{\ell}$.

The semantics of ports in the CTA model can be formalized using such periodic event sequences. A port p can produce event n on its outgoing connections at the moment that event n is available at all of its incoming connections. The moment at which the first event can be produced is called the start time $s(p)$ of port p . This moment is larger than or equal to the maximum of the offsets of the incoming periodic event sequences. The maximum rate at which port p can produce data from that moment on is equal to the minimum of the rates of the incoming periodic event sequences.

A connection in the CTA model can also be formalized using periodic event sequences. A connection is always directed from one port to another. Thus a connection (p, q) receives one periodic event sequence from port p and produces a periodic event sequence on port q . The semantics of the connection can be

Figure 6.5: Periodic event sequences of a connection.

formalized as a transformation of the parameters of the incoming periodic event sequence. Consider the situation illustrated in Figure 6.5. Given a periodic event sequence (o_p, ℓ_p) at port p the periodic event sequence on port q is constrained by connection (p, q) . The event sequence at port q is then equal to (o_q, ℓ_q) with $o_q \geq o_p + \epsilon(c_{pq}) + \ell_p \cdot \phi(c_{pq})$ and $\ell_q = \frac{1}{\lambda(c_{pq})} \cdot \ell_p$.

6.3 COMPOSITION

In this chapter we use the following definition for compositionality which is taken from [Ost95]:

Definition 6.1:

A system is called compositional, if the properties of a complex system can be deduced from the specifications of its component modules, without any further information about the exact internal structure of these modules.

In this section such a composition function is defined for the CTA model. The composition of components is again a component and the properties of this new component can be deduced from the individual components of the composition and the added connections between the components. Only consistent compositions are allowed as will be defined in subsequent sections. Furthermore, the connections between the external ports of a component, including their properties suffices for making a valid composition.

In Section 6.3.1 we describe the composition function itself. In Section 6.3.2 the consistency of a composition is discussed. Associativity of the composition function of the CTA model is presented in Section 6.3.3. The last section, Section 6.3.4, defines a function which can hide ports from the specification of a component without losing any information on the resulting ports of the component.

6.3.1 SPECIFICATION

Composing two components in the CTA model adds connections between the two components. Consider two components $A = (P_A, \hat{r}_A, C_A, \lambda_A, \phi_A, \epsilon_A)$ and $B = (P_B, \hat{r}_B, C_B, \lambda_B, \phi_B, \epsilon_B)$ where P_A and P_B are disjoint sets of ports. The result of the composition $D = A \oplus B$ is a new component which is specified by A, B and $\oplus$. The compose function $\oplus$ adds connections $C_\oplus$ between components A and B with $C_\oplus \subseteq (P_A \cup P_B) \times (P_A \cup P_B)$. The ratio between the rates on the added connections is specified by the function $\lambda_\oplus : C_\oplus \rightarrow \mathbb{R}$, the rate dependent delay introduced

on the new connections by the function $\phi_{\oplus} : C_{\oplus} \rightarrow \mathbb{R}$ and the constant delay by $\epsilon_{\oplus} : C_{\oplus} \rightarrow \mathbb{R}$.

The result of the composition $D = A \oplus B$, is formalized as the component specification $D = (P_D, \hat{r}_D, C_D, \lambda_D, \phi_D, \epsilon_D)$. The ports of D are equal to the union of the ports of A and B , i.e. $P_D = P_A \cup P_B$. The maximum rates $\hat{r}_D$ are defined using the maximum transfer rates of A and B :

$$\hat{r}_D(p) = \hat{r}_x(p) \quad \text{for all } p \in P_x, \text{ with } x \in \{A, B\}$$

The set of connections between the ports of component D is specified by $C_D = C_A \cup C_B \cup C_{\oplus}$. For each of these connections the rate ratio λ_D , the rate dependent delay ϕ_D , and the constant delay ϵ_D , are specified as follows:

$$\lambda_D(c) = \lambda_x(c) \quad \text{for all } c \in C_x, \text{ with } x \in \{A, B, \oplus\}$$

$$\phi_D(c) = \phi_x(c) \quad \text{for all } c \in C_x, \text{ with } x \in \{A, B, \oplus\}$$

$$\epsilon_D(c) = \epsilon_x(c) \quad \text{for all } c \in C_x, \text{ with } x \in \{A, B, \oplus\}$$

6.3.2 CONSISTENCY

A CTA component needs to meet a certain number of constraints. Rates of ports are coupled by connections between ports and the constraints imposed on the start times of ports, needs to be met. Because a port can be connected by multiple in- and/or outgoing connections, multiple constraints must be met for a port. We call a CTA component consistent if all the constraints for all the ports can be met.

Composing two consistent components can lead to inconsistencies because cyclic constraints that can not be met can be created between the two components. This section presents a method to verify the consistency of a component. This method can obviously also be used to check if a composition (which itself is also a component) is consistent. Inconsistent compositions and components are not allowed in the CTA model because they have ports on which it is not possible to transfer any data or connections on which data is accumulated.

A CTA component $V = (P, \hat{r}, C, \lambda, \phi, \epsilon)$ describes the following set of constraints: The transfer rates of ports connected by connections are coupled:

$$\forall c_{ij} \in C : r(j) = \lambda(c_{ij}) \cdot r(i)$$

Furthermore, the start time $s(p)$ of a port p needs to be larger or equal than the offsets of all the periodic event sequences on the incoming connections. The offset on an incoming connection can be specified using the start time of the other port of the connection and the delay of the connection. The start time constraint for all the ports can thus be enforced with:

$$\forall c_{ij} \in C : s(j) \geq s(i) + \epsilon(c_{ij}) + \frac{\phi(c_{ij})}{r(i)}$$

Next to that we have that the transfer rate of ports is larger than 0 and smaller or equal than its maximum rate:

$$\forall p \in P : 0 < r(p) \leq \hat{r}(p)$$

We now define a component to be consistent if a solution exist for the LP program, defined in Algorithm 5, in which we have substituted $r(p)$ by $1/\bar{\ell}(p)$. Note that LP programs can be solved in polynomial time and thus checking the consistency of a component has a polynomial time-complexity.

Algorithm 5 : Consistency

$$\text{Minimize } \sum_{p \in P} \bar{\ell}(p)$$

Subject to

$$\forall c_{ij} \in C : \bar{\ell}(j) = \frac{1}{\lambda(c_{ij})} \cdot \bar{\ell}(i) \quad (6.1)$$

$$\forall c_{ij} \in C : s(j) \geq s(i) + \epsilon(c_{ij}) + \phi(c_{ij}) \cdot \bar{\ell}(i) \quad (6.2)$$

$$\forall p \in P : \bar{\ell}(p) \geq \frac{1}{\hat{r}(p)} \quad (6.3)$$

Algorithm 5 not only checks consistency but also calculates for each port the minimum event distance. This corresponds with calculating for each port p the maximum possible transfer rate $\bar{r}(p) = 1/\bar{\ell}(p)$ for which the component is consistent. These maximum possible transfer rates can be useful if one is interested in the temporal properties of the component.

The set of constraints imposed by the composition of components A and B is equal to the union of the set of constraints that describes A , the set of constraints that describes B , and the set of constraints that describes the connections between A and B .

The solution of LP programs is exact in the sense that there is no smaller solution for the LP for which the constraints are satisfied. Furthermore, we show that for each $p \in P$ the optimal value is unique, i.e., there can be only one value for $\bar{\ell}(p)$ for which the solution of the LP is optimal. This is shown as follows: a CTA component can be seen as a directed graph made by the ports and the connections of the component. This directed graph is split in k weakly connected subgraphs. Such a weakly connected subgraph can be seen as a set of ports and connections between these ports such that there exists a path between each pair of ports of the subgraph. The subgraph is called weakly connected because the paths between the ports which make the subgraph weakly connected do not need to follow the directions of the connections. Note that there exist no connections between the different weakly connected subgraphs.

Because each pair of ports in a weakly connected subgraphs are connected, the distances between the events of all the ports in a weakly connected subgraph

are coupled, i.e. $\bar{\ell}(q) = \xi \cdot \bar{\ell}(p)$ with ξ a constant and ports q and p belonging to the same weakly connected subgraph. We call the ports of a subgraph i , P_i , and use $|P_i|$ for the number of ports of subgraph i . Because of the coupled event distances, we can write the sum of minimum distances between events of each port in subgraph i as follows:

$$\begin{aligned} \sum_{p \in P_i} \bar{\ell}(p) &= \bar{\ell}(p_0) + \xi_1 \cdot \bar{\ell}(p_0) + \dots + \xi_{|P_i|-1} \cdot \bar{\ell}(p_0) \\ &= \bar{\ell}(p_0) \cdot (1 + \xi_1 + \dots + \xi_{|P_i|-1}) \end{aligned} \quad (6.4)$$

Minimizing $\sum_{p \in P_i}$ is thus equivalent to minimizing $\bar{\ell}(p_0)$ for which one unique and exact minimum solution exists. There can thus be only one assignment to the individual values for $\bar{\ell}(p)$ that leads to the optimal solution for the minimum distances between events of a subgraph. Because the different weakly connected subgraphs are not connected, the optimal solution for a subgraph can not influence the minimal distances of other subgraphs, i.e., smaller minimum event distances for one weakly connected subgraph can not result in larger distances between events of another subgraphs. The minimum event distances calculated with Algorithm 5 are thus optimal and unique for each port $p \in P$.

6.3.3 ASSOCIATIVITY

This section shows that the composition operation is associative. This means that the resulting CTA model, after performing multiple compositions, does not depend on the order in which these compositions take place. This allows for incremental design because the consistency of compositions can be checked even if not all components are fully specified. The consistency of a subsystem can thus be checked separately from the complete system because composing the complete system does not depend on the order in which the different compositions are done.

Consider two compositions of three components: $A \oplus_0 (B \oplus_1 C)$ and $(A \oplus_2 B) \oplus_3 C$. We require that the set of added connections and the corresponding functions, i.e., λ , ϵ and ϕ , of $\oplus_0 \cup \oplus_1$ are equal to the connections and functions added by $\oplus_2 \cup \oplus_3$. We use the notation $\oplus_0 \cup \oplus_1 = \oplus_2 \cup \oplus_3$ for this.

With this requirement we prove the associativity of the composition operation for the CTA model with the following proposition:

Proposition 6.1:

If $\oplus_0 \cup \oplus_1 = \oplus_2 \cup \oplus_3$ then also $A \oplus_0 (B \oplus_1 C) = (A \oplus_2 B) \oplus_3 C$.

Proof. With composition, the ports, their connections and the delays on the connections of the components are not changed, see Section 6.3.1. Because also the added connections are equal for both compositions the resulting components are indeed equal.

Figure 6.6: Example of a CTA component where port y is hidden.

As discussed in Section 6.3.2, composition of components is equal to taking the union of the constraints of the components with the added constraints for the connections between the components. Union of sets is an associative operation so the constraints imposed by both the compositions are also equal. Because the constraints are equal also the possible solutions are equal. As shown in Section 6.3.2 solving the consistency algorithm always results in the exact and unique solution. No intermediate approximations are done which means that the composition operation is associative. $\square$

6.3.4 HIDING

Ports of a component that do not need to be connected from outside the component can be hidden from the component description. Hiding removes the port while maintaining the same constraints between the remaining ports of the component. It is therefore an exact operation in the sense that it does not change the temporal properties of the remaining ports. The transformation of Figure 6.4(b) into Figure 6.4(c) shows an example of hiding.

Often, when applying the compose function $\oplus$ on two components A and B , the ports that become connected by the added connections $C_{\oplus}$, do not need to be visible to the outside anymore. These ports can be hidden from the interface description of a component with the method presented in this section. This leads to a smaller description of the component. Next to that, it enables hierarchy and it enables the creation of valid black-box components for which only the external ports together with the connections between these external ports are known.

The idea of hiding a port p is that all the indirect constraints between ports which follow from connections from and to p are replaced by direct constraints. This is done by adding connections from all the ports with a connection to p to all the ports with a connection from p . We illustrate this with the example shown in Figure 6.6. Port y is hidden from the component description and the indirect constraints imposed by port y need to be redistributed. This is done by removing the two connections (x, y) and (y, z) and adding a new connection (x, z) . The characterization of this new connection is as follows.

We have that $r(y) = \lambda(c_{xy}) \cdot r(x)$ and $r(z) = \lambda(c_{yz}) \cdot r(y)$ and thus we have $r(z) = \lambda(c_{xy}) \cdot \lambda(c_{yz}) \cdot r(x)$. When we choose $\lambda(c_{xz}) = \lambda(c_{xy}) \cdot \lambda(c_{yz})$ we have $r(z) = \lambda(c_{xz}) \cdot r(x)$.

Next to that, the total delay of the two connections is equal to:

$$\Delta(c_{xz}) = \Delta(c_{xy}) + \Delta(c_{yz}) = \epsilon(c_{xy}) + \frac{\phi(c_{xy})}{r(x)} + \epsilon(c_{yz}) + \frac{\phi(c_{yz})}{r(y)}.$$

Because $r(y) = \lambda(c_{xy}) \cdot r(x)$ we have that $\frac{\phi(c_{yz})}{r(y)} = \frac{\phi(c_{yz})}{\lambda(c_{xy}) \cdot r(x)}$ and thus:

$$\Delta(c_{xz}) = \epsilon(c_{xz}) + \frac{\phi(c_{xz})}{r(x)} \text{ with } \epsilon(c_{xz}) = \epsilon(c_{xy}) + \epsilon(c_{yz}) \text{ and } \phi(c_{xz}) = \phi(c_{xy}) + \frac{\phi(c_{yz})}{\lambda(c_{xy})}.$$

We can generalize this approach to the following method in which we hide a port $p \in P$ from a component $V = (P, \hat{r}, C, \lambda, \phi, \epsilon)$ such that a new component $V' = (P', \hat{r}', C', \lambda', \phi', \epsilon')$ is created. We have $P' = P \setminus \{p\}$ and $\hat{r}'(p) = \hat{r}(p)$ for every port $p \in P'$

We first add the following direct connections to bypass the indirect connections via p :

$$\begin{aligned} C' &= C \cup C_n \quad \text{with} \\ C_n &= \{(i, j) \mid (i, p) \in C \wedge (p, j) \in C\} \end{aligned}$$

The values for $\lambda'(c)$, $\epsilon'(c)$ and $\phi'(c)$ for connections $c \in C$ are equal to $\lambda(c)$, $\epsilon(c)$ and $\phi(c)$ respectively. The values for connections $c \in C_n$ are as follows:

$$\begin{aligned} \lambda'(c_{ij}) &= \lambda(c_{ip}) \cdot \lambda(c_{pj}) & \text{with } c_{ip}, c_{pj} \in C \\ \epsilon'(c_{ij}) &= \epsilon(c_{ip}) + \epsilon(c_{pj}) & \text{with } c_{ip}, c_{pj} \in C \\ \phi'(c_{ij}) &= \phi(c_{ip}) + \frac{\phi(c_{pj})}{\lambda(c_{ip})} & \text{with } c_{ip}, c_{pj} \in C \end{aligned}$$

A number of constraints imposed by port p remain. Self connections from port p back to itself need to be redirected and also the maximum rate constraint of port p needs to be moved. We first calculate the maximum and minimum rate that is enforced by the constraints on port p and then move these maximum and minimum rate constraints to the ports which originally had a connection to port p .

A self connection of port p imposes a maximum rate constraint when it has a positive constant delay. In that case it also does have a negative delay otherwise the CTA component would not be consistent. The maximum rate that is enforced by such a connection is equal to $\frac{-\phi}{\epsilon}$. For example if a self connection has a constant delay of $\epsilon = 3$ and a variable delay of $\phi = -5$ we have: $3 + \frac{-5}{r} \leq 0$ and thus $r \leq \frac{5}{3}$ with r the transfer rate of the port.

The maximum rate imposed by self connections and the maximum rate specification is thus equal to:

$$\begin{aligned} r_{\max}(p) &= \min \left(\hat{r}_p, \min_{c \in C_{\max}} \left(\frac{-\phi(c)}{\epsilon(c)} \right) \right) \\ &\text{with } C_{\max} = \{c \mid c \in C \wedge c = (p, p) \wedge \epsilon(c) > 0\} \end{aligned} \quad (6.5)$$

When the constant delay of a self connection of port p is negative, a minimum rate constraint can be specified. Consider a self connection with a constant delay of $\epsilon = -3$ and a variable delay of $\phi = 5$. The rate constraint imposed by this connection is: $-3 + \frac{5}{r} \leq 0$ and thus $r \geq \frac{5}{3}$ with r again the transfer rate of the port.

The minimum rate imposed by self connections can be determined similarly as the maximum rate constraint:

$$r_{min}(p) = \max_{c \in C_{min}} \left(\frac{\phi(c)}{-\epsilon(c)} \right) \quad (6.6)$$

$$\text{with } C_{min} = \{c \mid c \in C \wedge c = (p, p) \wedge \epsilon(c) < 0\}$$

Now we add these constraints to each of the ports which originally had a connection to port p . We add one self connection to model the maximum transfer rate constraint and if the minimum rate constraint is defined, we add another self connection to model this minimum transfer rate.

We first add the self connection to C' on a port i to model the maximum transfer rate. This port i has in the original CTA model a connection c_{ip} to port p . This new self connection has a transfer ratio of $\lambda = 1$, a constant delay of $\epsilon = 1$ and a variable delay of $\phi = -r_{max}(p)$.

If a minimum rate constraint on port p was defined, we add a self connection to C' on every port i for which a connection c_{ip} existed in the original CTA component. This self connection has a transfer ratio of $\lambda = 1$, a constant delay of $\epsilon = -1$ and a variable delay of $\phi = r_{min}(p)$.

Now all the constraint on and via port p are redirected. As a last step, the connections to and from port p can safely be removed from C' .

The presented method for hiding a port only redistributes constraints between the remaining ports and does not add or remove constraints. Hiding a port thus does not influence the consistency of the component and also the temporal properties of the component do not change.

6.4 PRACTICAL APPLICATIONS OF THE CTA MODEL

An analysis model becomes useful if it can be derived from a different level of abstraction. In this section we illustrate some of the applications of the CTA model and give some examples of how the CTA model can be used as an extra level of abstraction.

In Chapter 2 it has been shown that different types of dataflow models can be used to model the temporal behavior of real-time applications. In Section 6.4.1 we give an example of how an SDF graph can be temporally analyzed with a conservative CTA model. This conservative CTA modeling is enabled by an extension of research performed on the linearized analysis of SDF graphs.

There are difficulties with expressing periodic sources and sinks in dataflow graphs. Section 6.4.2 shows how to include such periodic sources and sinks in

Figure 6.7: Translation of an SDF actor to a CTA component.

the analysis of CTA models. Furthermore, we illustrate in Section 6.4.3 how the CTA model can be used to calculate the sizes of buffers of applications. Adding latency constraints to dataflow graphs is in general also difficult. In Section 6.4.4 we show that latency constraints can be analyzed with the CTA model.

This section is concluded with a case-study in which the CTA model is used to model the temporal behavior of a car-radio application.

6.4.1 CTA ABSTRACTION OF SDF GRAPHS

This section shows that the CTA model can be used as an abstraction of SDF graphs. Exact analysis of SDF graphs often uses a transformation from the SDF graph to a corresponding HSDF graph. This transformation has a worst-case exponential blowup in the number of nodes and connections [JSL95]. By using linear bounds to conservatively bound schedules of the SDF graph, the transformation to an HSDF becomes redundant. The abstraction from an SDF graph to a CTA model uses these linear bounds and thus results in an analysis method in which the transformation to an HSDF graph is superfluous. The analysis algorithms defined for the CTA model have a polynomial computational complexity which means that using the CTA model as an abstraction for an SDF graph, the SDF graph can be conservatively analyzed in polynomial time.

Given that a periodic schedule exists between such linear bounds, and given that the self-timed execution of SDF graphs has a monotonic temporal behavior, it can be concluded that tokens are produced earlier than in the periodic schedule, see Chapter 2. Using linear bounds only leads to conservative analysis results because the bound on the production of tokens assumes later production times than the periodic schedule.

Figure 6.7 shows the abstraction of an SDF actor to a CTA component. Every incoming and every outgoing edge of the SDF actor corresponds to one port of the CTA component. Figure 6.7(a) illustrates the case for one incoming and one outgoing edge of the actor. The corresponding CTA component in Figure 6.7(b) has two ports. Every firing of the actor in Figure 6.7(a) takes ρ time and every firing γ tokens are consumed from the incoming edge and π tokens produced on the outgoing edge. The actor has a self-edge with one token to denote that its firings may not overlap which means that it can fire maximally once every ρ

Figure 6.8: Linear bounds on a schedule for the actor shown in Figure 6.7(a) with $\gamma = 3$, $\pi = 2$ and $\rho = 3$.

time units. The maximum transfer rates of the actor are thus equal to $\frac{\gamma}{\rho}$ for the incoming edge and $\frac{\pi}{\rho}$ for the outgoing edge. These transfer rates are used as the maximum rates of the corresponding ports in the CTA component. Note that if an actor does not have a self-edge, its maximum transfer rate would be equal to infinity. This can still be analyzed using a CTA component that does not have constraints on the maximum transfer rates of its ports.

The connections of the CTA component are as follows. For every port of the CTA component that corresponds to an incoming edge of the SDF actor, connections are added to every port that corresponds to an outgoing edge of the SDF actor. For example, if the SDF actor has 2 incoming edges and 3 outgoing edges, then the corresponding CTA component will consist of five ports and six connections between these ports.

Figure 6.7(b) shows the typical characterization of a connection in the created CTA component. The transfer rate ratio of the connection is equal to the number of tokens produced on the corresponding outgoing edge divided by the number of tokens consumed on the corresponding incoming edge. For the CTA component illustrated in Figure 6.7(b) this ratio is equal to $\frac{\pi}{\gamma}$.

For the calculation of the delay on the connection, we use the discussed linear bounds. Figure 6.8 shows a periodic schedule for the actor of Figure 6.7(a). The vertical axis shows the cumulative token transfer and the horizontal axis the elapsed time. The consumptions of tokens is visualized with circles and every firing, $\gamma = 3$, tokens are consumed at the beginning of that firing. The productions of tokens are visualized with squares. The number of produced tokens in every firing equals $\pi = 2$ and the tokens are produced at the end of the firing. The duration of a firing is $\rho = 3$.

We have drawn the schedule with a consumption rate of $r_c = \frac{3}{6}$ and a production rate of $r_p = \frac{2}{6}$. The start time of each firing f is defined as: $s(f) = \frac{f \cdot \gamma}{r_c}$. The tokens of firing f are consumed at time $s(f)$ and produced at time $s(f) + \rho$.

We start numbering tokens with token number 0. The token with the maximum token number that is consumed at time $s(f)$ is thus token $(f + 1) \cdot \gamma - 1$. The token with the lowest token number that is produced at time $s(f) + \rho$ is token $f \cdot \pi$.

An upper bound on the consumption of tokens can be defined with the formula $\hat{\alpha}_c = r_c \cdot t + (\gamma - 1)$ and a lower bound on the production of tokens can be defined as $\check{\alpha}_p = r_p \cdot (t - \rho)$. This is illustrated in Figure 6.8.

The delay of a connection can then be seen as the horizontal difference between the production bound, $\check{\alpha}_p$, and the consumption bound, $\hat{\alpha}_c$, on the x-axis ($\check{\alpha}_p = 0$ and $\hat{\alpha}_c = 0$). This corresponds to the difference in start times and is, as shown in Figure 6.8, equal to $\rho - \frac{1-\gamma}{r_c}$. The constant delay of the connection in the CTA component is thus ρ and the rate dependent delay of the connection is $\gamma - 1$.

These delays can also be derived algebraically using the knowledge about the consumption and production quanta of the SDF actor and using information about the periodic schedule of the firings. We use the notation of Chapter 2 for event sequences. We call the event sequence on the incoming queue of the SDF actor of Figure 6.7(a), sequence c , and the event sequence on the outgoing queue, sequence p . The event sequence on the incoming connection of the corresponding CTA component in Figure 6.7(b) is called c' and the event sequence on the outgoing connection p' .

Using the delays derived with the linear bounds of Figure 6.8, we have $p'(0) = c'(0) + \rho + \frac{\gamma-1}{r_{c'}}$ and $p'(i) = p'(0) + \frac{i}{r_p}$. We show that these delays can also be derived algebraically by conservatively approximating the periodic schedule of the SDF actor. With this algebraic derivation we also show that the SDF actor refines the corresponding CTA model, i.e., $\forall i \in \mathbb{N} : c\langle i \rangle \sqsubseteq c'\langle i \rangle \implies p\langle i \rangle \sqsubseteq p'\langle i \rangle$.

We first formalize the periodic schedule of the SDF actor and its enabling based on its firing rule. With this schedule and this enabling we derive conservative bounds on the token production times.

Given a periodic schedule of the SDF actor we have that firing j of the SDF actor starts $\frac{\gamma}{r_c}$ time units after firing $j - 1$ of the SDF actor. With r_c the rate of the event sequence on the incoming queue. This is equal to what we used in the delay derivation using the linear bounds.

Given the firing rule of SDF actors, we know that firing f of the SDF actor of Figure 6.8 can start when token $(f + 1) \cdot \gamma - 1$ can be consumed from the incoming queue of Figure 6.8. Equivalently we know that token i is consumed from the incoming queue in firing $\left\lfloor \frac{i}{\gamma} \right\rfloor$. Given the periodic schedule of the SDF actor it holds that token i is consumed at time stamp $c(i) = c(0) + \left\lfloor \frac{i}{\gamma} \right\rfloor \cdot \frac{\gamma}{r_c}$.

Furthermore, token i is produced at the end of firing $\left\lfloor \frac{i}{\pi} \right\rfloor$ of the SDF actor. This firing is finished ρ time units after it is started. Combining the information about consumed and produced tokens we know that token i is produced at the time stamp at which token $\left(\left\lfloor \frac{i}{\pi} \right\rfloor + 1\right) \cdot \gamma - 1$ is consumed plus the firing duration of the SDF actor, ρ :

$$p(i) = c \left(\left(\left\lfloor \frac{i}{\pi} \right\rfloor + 1 \right) \cdot \gamma - 1 \right) + \rho$$

Using the information about the periodic schedule we have:

$$\begin{aligned} &= c(0) + \frac{\left\lfloor \frac{(\lfloor \frac{i}{\pi} \rfloor + 1) \cdot \gamma - 1}{\gamma} \right\rfloor \cdot \gamma}{r_c} + \rho \\ &= c(0) + \frac{\left\lfloor \left\lfloor \frac{i}{\pi} \right\rfloor + 1 - \frac{1}{\gamma} \right\rfloor \cdot \gamma}{r_c} + \rho \end{aligned}$$

We now use twice that $\forall a \in \mathbb{R} : \lfloor a \rfloor \leq a$:

$$\begin{aligned} &\leq c(0) + \frac{\left\lfloor \frac{i}{\pi} \right\rfloor \cdot \gamma + \gamma - 1}{r_c} + \rho \\ &\leq c(0) + \frac{\gamma - 1}{r_c} + \frac{i \cdot \frac{\gamma}{\pi}}{r_c} + \rho \end{aligned}$$

Using $r_p = \lambda(c_{cp}) \cdot r_c = \frac{\pi}{\gamma} \cdot r_c$ we obtain:

$$\begin{aligned} &= c(0) + \frac{\gamma - 1}{r_c} + \frac{i}{r_p} + \rho \\ &\leq c'(0) + \frac{\gamma - 1}{r_c} + \frac{i}{r_p} + \rho \\ &= p'(i) \end{aligned} \tag{6.7}$$

Given $c(0) \leq c'(0)$ we thus have $\forall i \in \mathbb{N} : p(i) \leq p'(i)$. The SDF model thus refines the corresponding CTA model: $\forall i \in \mathbb{N} : c\langle i \rangle \sqsubseteq c'\langle i \rangle \implies p\langle i \rangle \sqsubseteq p'\langle i \rangle$.

An edge in an SDF graph is formed by connecting an outgoing edge from an actor to an incoming edge of an actor. Such an edge can be abstracted in a CTA model with a connection from the port corresponding to the outgoing edge to the port corresponding to the incoming edge. The transfer rate ratio on such a connection is equal to 1 and the delay of the connection can be calculated by using the number of initial tokens on the corresponding edge. Initial tokens allow the consuming actor to start consuming tokens before the producing actor produces tokens. For δ initial tokens and a transfer rate r on the edge, the consuming actor can start $\frac{\delta}{r}$ time before the first token is produced. This can be modeled in the CTA model with a delay of $-\frac{\delta}{r}$. The value $-\delta$ can thus be used as the rate dependent delay ϕ of the connection in the CTA model.

Example 6.1:

Figure 6.9 shows an SDF graph and the corresponding CTA model. Given the seven initial tokens of the SDF graph we can transform the SDF graph in an equivalent HSDF graph, see Chapter 2. On this HSDF graph it can be computed with an MCM algorithm [Das04, SB00] that actor v_a can fire on average twice every seven time

Figure 6.9: SDF to CTA example.

units. Because actor v_a consumes and produces 3 tokens per firing the average transfer rate is equal to $3 \cdot \frac{2}{7} = \frac{6}{7}$ tokens/time unit. Furthermore, all the rates in the dataflow are equal and are called r . With Algorithm 5 we can calculate that $r = \frac{4}{5}$ is the maximum transfer rate that ensures consistency. The calculated maximum transfer rate using the abstraction to the CTA model is thus slightly less accurate than the exact calculated transfer rate of the SDF model. This inaccuracy is the result of using conservative linear bounds.

6.4.2 PERIODIC SOURCES AND SINKS

Periodic sources are elements in an application that deliver data at a fixed transfer rate, they can not be delayed. Similarly, periodic sinks are elements that require data with a fixed transfer rate and can not be delayed too. Normal components in the CTA model are characterized with maximum transfer rates in contrast to what periodic sources and sinks require. If a periodic source or sink is composed with other components, the rate of the ports connected to this source or sink must adapt their transfer to this fixed rate to ensure that the components can keep up with the source or sink. To enforce this fixed rate, extra constraints can be added for periodic sources and sinks.

Sources and sinks can be expressed in the CTA model by modeling it as a normal component with a maximum transfer rate for each port equal to its fixed transfer rate. This fixed transfer rate can then be enforced in the consistency algorithm, as defined in Algorithm 5, by adding an extra constraint that states that for each port p of the source or sink, ℓ_p is equal to $\frac{1}{r_s}$ with r_s the fixed transfer rate of the source or sink. This is equivalent to adding two connection from and to each port p of the source or sink. By choosing the delay on these two connections correctly, the fixed transfer rate can be enforced. The rate ratio of both these connections is equal to 1, the constant delay is equal to 1 and -1 respectively and the variable delay is equal to $-r_s$ and r_s respectively.

The rate constraints imposed by these two connections are then as follows. The first connection enforces: $1 + \frac{-r_s}{r} \leq 0$, which is equivalent to:

$$r \leq r_s$$

The second connection enforces: $-1 + \frac{r_s}{r} \leq 0$, which is equivalent to:

$$r_s \leq r$$

Both connection together model thus the constraint:

$$r = r_s$$

6.4.3 BUFFER SIZING

The CTA model can be used to calculate the required sizes of buffers in an application. In this section we show how this can be done if the CTA model is used as an abstraction for an SDF graph.

A buffer with δ locations can be modeled in an SDF graph with two oppositely directed edges. One edge modeling the flow of empty locations and one modeling the flow of filled locations. A token corresponds to a location in the buffer so the sum of the number of tokens on the two edges always needs to be less or equal to δ tokens, to take the size of the buffer conservatively into account. The two edges in the middle of Figure 6.9(a) model for example a buffer with 7, initially empty, locations.

Typically, real-time stream processing applications have a throughput constraint. For example because they need to process values from a periodic source. This throughput constraint imposed by for example a periodic source can be included in the CTA model with the method discussed in Section 6.4.2. The throughput which the application can meet, depends, among other things, on the sizes of the buffers. Therefore, correct sizes of the buffers need to be calculated to ensure that the throughput constraint can be met.

As we have seen in Section 6.4.1, the variable delay ϕ of a connection that corresponds to an edge in an SDF graph is equal to $-\delta$ with δ the number of tokens on the edge. If the size of the buffer is not fixed, this number of tokens δ is also variable. We can use the CTA model of an SDF graph to find sufficient values for δ such that the throughput constraint can be met. Finding the smallest buffer sizes for which the throughput constraint can be met is equivalent to finding the maximum possible variable delays given the constraints.

For simplicity we assume a fully connected SDF graph which means that the corresponding CTA model is also fully connected. Therefore, adding one source or sink in the model immediately leads to fixed transfer rates of all the ports in the CTA model. This allows us to define an algorithm which calculates the values for the variable delays for which the transfer rates can be met.

Because all the ports of the CTA model are connected, we can assume that there is one port which defines the throughput constraint. We call this port p_τ with the throughput constraint $\frac{1}{\tau}$ which can be enforced by enforcing that $\bar{\ell}(p_\tau)$ is equal to τ . Next to that, we introduce a set of connections C_v which contains all the connections for which we need to compute the variable delay. Because

Algorithm 6 Buffer sizing

$$\text{Maximize } \sum_{c \in C_v} \phi(c)$$

Subject to

$$\forall c_{ij} \in C : \bar{\ell}(j) = \frac{1}{\lambda(c_{ij})} \cdot \bar{\ell}(i) \quad (6.8)$$

$$\forall c_{ij} \in C : s(j) \geq s(i) + \epsilon(c_{ij}) + \phi(c_{ij}) \cdot \bar{\ell}(i) \quad (6.9)$$

$$\forall p \in P : \bar{\ell}(p) \geq \frac{1}{\hat{r}(p)} \quad (6.10)$$

$$\forall c \in C_v : \phi(c) \leq 0 \quad (6.11)$$

$$\bar{\ell}(p_r) = \tau \quad (6.12)$$

the numbers of tokens on the edges corresponding to these connections can only be positive we have a constraint on the variable delay on these edges: $\phi(c) \leq 0$.

Sufficient variable delays can now be computed with Algorithm 6. Algorithm 6 can be solved with an LP solver because the given constraints can be simplified using the fact that all the $\bar{\ell}(p)$ variables are constants. Because $\bar{\ell}(p)$ is a constant, Equation 6.9 also forms a linear constraint. Furthermore, because all the $\bar{\ell}(p)$ variables are constant, Equations 6.8, 6.10 and 6.12 can be checked locally and can be removed from the algorithm.

The algorithm finds assignments to the variable delays on connections such that the throughput constraint of the corresponding application can be met. With these variable delays, sufficient numbers of tokens can be computed. The variable delay ϕ_c of a connection c is equal to $-\delta$ with δ the number of tokens on the corresponding edge in the dataflow graph. The number of tokens on an edge is thus equal to $-\phi_c$ with c the connection corresponding to the edge.

However, tokens in SDF graphs are integers while the solution of an LP algorithm is in general a set of real values. To solve this issue we can make use of the monotonicity property of SDF graphs as discussed in Chapter 2. This property tells us that increasing the number of tokens on an edge cannot lead to worse temporal results. We therefore know that the throughput constraint of the application can still be met even if we increase the numbers of tokens in the graph. We choose the number of tokens on an edge corresponding to a connection c equal to $\lceil -\phi_c \rceil$.

Example 6.2:

For the SDF graph shown in Figure 6.10(a) we want to compute the number of tokens δ such that a throughput requirement of $r = \frac{1}{2}$ tokens/time unit can be met. We first generate the conservative CTA model shown in Figure 6.10(b) with the method presented in Section 6.4.1. We now use Algorithm 6 on this CTA model to compute ϕ such that $\ell = \frac{1}{r} = 2$ can be achieved. The algorithm tells us that $\phi = -2.5$ is

Figure 6.10: Buffer sizing example for an SDF graph with a conservative CTA model.

Figure 6.11: A CTA model with latency constrained connections.

the largest value that can achieve a transfer rate of $\frac{1}{2}$. Thus with $\delta = \lceil 2.5 \rceil = 3$ the dataflow graph should be able to meet its throughput constraint. Applying an MCM [Rei68] algorithm on the equivalent HSDF graph indeed tells us that this is the case.

6.4.4 LATENCY CONSTRAINTS

In the CTA model it is also possible to take latency constraints into account. Two components that have latency constraints, should provide ports on which the latency constraint can be set. These ports should internally be correctly connected to the other ports of the component to enforce that these ports also adhere to the latency constraints. Adding a latency constraint can then be done by adding a connection between two such latency constraint ports of the components.

Figure 6.11 shows how this can be done in the CTA model. In the model, the ports p_0 , p_1 , p_2 and p_3 are added to specify a maximum and a minimum latency constraint. The connection (p_0, p_1) specifies a maximum latency constraint of L_{max} time units between port o and port i and the connection (p_2, p_3) specifies a minimum latency constraint of L_{min} time units between o and i .

For the maximum latency constraint, internal connections (p_1, i) and (o, p_0) are added such that together with (p_0, p_1) a delay constraint is created on ports i and o . The connections (p_1, i) and (o, p_0) do not contain any delay. The start

Figure 6.12: Block diagram of an audio echo cancellation application.

time constraint on ports enforces that data arrives at port i , maximally L_{max} time units before it arrives at o :

$$s(i) \geq s(o) - L_{max}$$

Or equivalent:

$$s(o) \leq s(i) + L_{max}$$

The minimal latency constraint, imposed by the path through the connections (i, p_2) , (p_2, p_3) and (p_3, o) , specifies a minimal delay between the ports i and o . The connections (i, p_2) and (p_3, o) do again not contain any delay. The start time constraint on ports ensures that data arrives at least L_{min} time units later at port o than it arrives at port i :

$$s(o) \geq s(i) + L_{min}$$

The other connections between components A and B now need to have delays such that the maximum and minimum latency constraints can be met.

6.4.5 CASE-STUDY

In this section we use the CTA model to analyze the temporal behavior of a car-radio application. The application is the audio echo cancellation application which is also used in one of the case studies of Chapter 4. In Chapter 4 the application is modeled using an SDF model. We show that with the CTA model this application and its constraints can be modeled more accurately than with SDF graphs. The application can also be analyzed, and thus designed, incrementally.

Figure 6.12 shows the block diagram of this application. A phone call can be handled using a Bluetooth (BT) device simultaneous with playing music at a lower volume. To prevent the howling effect and to cancel the sound from the speaker, audio echo cancellation is used. This ensures that only the speech of the

Figure 6.13: SDF model of the AEC task together with its adjacent tasks.

user is sent via the *BT* device. The latency between the microphone and *BT* may be at most $26ms$.

Similar as in Chapter 4, we focus in this case-study on the Audio Echo Cancellation (*AEC*) task and its adjacent tasks. The *OUT* and *ADC* task execute periodically at a frequency of exactly $8kHz$. The adjacent Sample Rate Conversion (*SRC*) task transforms a $44.1kHz$ stream to a $8kHz$ stream. Consistency is ensured when the *AEC* task can achieve a transfer rate higher or equal to $8kHz$.

Figure 6.13 contains an SDF graph of the *AEC* task together with its adjacent tasks. The self-edges of all actors are omitted for clarity. The *AEC* actor processes blocks of 80 samples to reduce the synchronization and scheduler overhead. The firing durations of the *SRC*, *ADC* and *OUT* actors are $\frac{1}{8}ms$ such that they can fire at a rate of $8kHz$. The firing duration of the *AEC* actor is defined in [WBJ07] as $9.091ms$ which means that its maximum transfer rate is $\frac{80 \text{ samples}}{9.091ms} > 8kHz$. However, as discussed in Chapter 4, the *AEC* task contains a 48 taps filter which causes the first 48 samples to be used only for filling the taps. Therefore, the *AEC* task has an additional algorithmic delay of $\frac{48 \text{ samples}}{8kHz} = 6ms$. An actor in the SDF graph can only model this delay inaccurately, which would lead to the conclusion that the maximum transfer rate of *AEC* is less than $8kHz$. Another option would be to not model the algorithmic delay and to subtract it from the latency constraint as is done in Chapter 4. However, this should be done carefully because then the analysis model is only conservative when the latency constraint is taken into account. Also the actual worst-case temporal behavior of the tasks is then not directly included in the analysis model.

The SDF graph can be analyzed using a CTA model that can be found with the technique presented in Section 6.4.1. Because the CTA model decouples the delay from the transfer rate of a component, we can model the additional algorithmic delay more accurately. The CTA model in which the *AEC* task is modeled with the algorithmic delay is shown in Figure 6.14. Each port in the CTA model has a transfer rate of $8kHz$, except the port denoted by $44.1kHz$ which has a transfer rate of $44.1kHz$. The delays of the blue connections inside *SRC*, *ADC* and *OUT* are all equal to $\frac{1}{8}ms$. The orange connection denoted by b in *SRC* has a delay that is larger than $\frac{1}{8}ms$ and we assume it to be equal to $\frac{2}{8}ms$. Furthermore, the green

Figure 6.14: CTA model corresponding to the SDF model of Figure 6.13.

Figure 6.15: The resulting CTA model after the *SRC* component is hidden from the CTA model of Figure 6.14.

connections inside *AEC* have a delay equal to $9.091 + \frac{79}{8} ms$. The red connections which are denoted by *a* include the algorithmic delay and have a delay equal to $15.091 + \frac{79}{8} ms$. We also modeled the maximum latency constraint between the microphone and *BT* with the connection denoted by -26 . This latency constraint can be directly specified in the CTA. As is shown in Chapter 4, in the SDF model this latency can only be modeled indirectly by using for example tokens. These tokens together with the rate of the *ADC*, then model the latency constraint. However, this is only possible when the latency constraint is a multiple of the period of the *ADC*.

The presented CTA model is consistent if the 158 initial tokens from [WBJ07] are used for δ_0 , δ_1 and δ_2 . With these initial tokens, the delays on the corresponding three connections in the CTA model are equal to $\frac{-158}{8} ms$ with which all the

delay constraints can be satisfied. The maximum delay between the microphone and *BT* is equal to $15.091 + \frac{81}{8}ms$ which is less than $26ms$.

Hiding can be applied to the ports of the *SRC* component. This will result in the CTA model shown in Figure 6.15. A new component *AEC'* is formed which now also does sample rate conversion on one of its inputs. The delay of the *SRC* is moved inside the *AEC'*. This results in the change of the delay of two internal connections compared to the delays of *AEC*. These connection are colored orange and purple and are denoted by *c* and *d* respectively. The orange connection has a delay equal to $15.091 + \frac{81}{8}ms$ and the purple connection has a delay equal to $9.091 + \frac{81}{8}$. Hiding the ports of the *SRC* component would also add a cyclic constraint from the port denoted by $44.1kHz$ back to itself. Given the chosen initial tokens, the delay on this connection would always be negative and is therefore left out of the figure. From the obtained model we conclude that there is no loss of accuracy when applying hiding because the end-to-end delays do not change.

6.5 RELATED WORK

A compositional analysis method for real-time systems is presented in [TWS06]. It defines adaptive interfaces of components to enable the validation of system constraints and introduces properties like refinement and independent implementability. The methods presented in [LPT09, PLT11] extend this analysis method with a uniform interface, based on arrival curves. This allows the composition of different types of modeling and analysis methods. However, the use of communication buffers with a finite capacity result in cyclic dependencies. Such cyclic dependencies result in an exponential worst-case computational complexity of the analysis algorithms while the presented analysis algorithms for the CTA model have a polynomial computational complexity. Furthermore, buffer sizing for cyclic task graphs is not addressed.

A formal algebra for the analysis of temporal properties is presented in [HM06]. It defines an algebra for composing components based on the description of their interface. These interfaces are characterized similarly as the interfaces of our components, with an arrival rate function for ports and a latency specification for tasks. It supports both incremental design and independent refinement of components but does not support cyclic dependencies between tasks.

To support hierarchical composition of actors in untimed SDF graphs, non-monolithic profiles are introduced in [TBG⁺13]. These profiles consist of SDF graphs extended with shared FIFOs. The focus of [TBG⁺13] is on the functional behavior of applications whereas the CTA model focuses on the compositional analysis of the temporal behavior of applications.

An analysis method which uses transfer rates of SDF actors to calculate schedules is introduced in [WBS06]. This method is extended in [WBS07a] for CSDF [BEP96] graphs. The CTA model also uses transfer rates and can be seen

as a generalization of these works because it supports compositionality. Furthermore, the inclusion of the effects of run-time scheduling in dataflow graphs has been presented in [WBS09]. This run-time scheduling is restricted to the use of starvation free schedulers.

In [GHB14a] the CTA model is used to hierarchically analyze the temporal behavior of applications with data-dependent execution behavior.

6.6 CONCLUSION

In this chapter we have introduced a compositional temporal analysis model in which components are characterized using ports with maximum transfer rates and connections with delays. This CTA model can be used to analyze applications with arbitrary cyclic dependencies, periodic sources and sinks and can take latency constraints into account. Furthermore, the analysis algorithms defined for the CTA model have a polynomial time-complexity.

We also provided an abstraction from an SDF graph to a conservative CTA model on which the analysis can be performed. With the CTA model more pessimistic results are usually obtained than when using SDF graphs. However, unlike the SDF model, the CTA model has compositional analysis properties.

We have also shown that the analysis with the CTA model supports independent implementability which helps to reduce the complexity when designing and developing stream processing applications. We furthermore presented a method for hiding the internal connections of a composition. This results in components which are only specified by their external ports with maximum transfer rates and connections between these ports containing delays.

The practical applicability of the CTA model has been illustrated with an audio echo cancellation application.

Conclusion

This thesis introduced new abstractions and improved existing abstractions for real-time stream processing applications executed on aperiodic multiprocessor systems. Compared to state of the art, results are obtained in both broadening the applicability of abstractions as well as improving the accuracy of existing abstractions.

The dynamic behavior of stream processing applications requires flexible multiprocessor systems. Tasks of the stream processing applications are therefore executed data-driven on run-time scheduled resources. Furthermore, the schedules of the tasks are aperiodic to allow for the varying and potentially data-dependent execution rates of tasks.

Because of the nature of stream processing applications, the existence of strict performance constraints is apparent. Violating these performance constraints leads to a major reduction of the quality of service of the applications. Therefore, analysis methods are used to ensure that the temporal constraints of applications are met.

These analysis methods rely on accurate abstractions of the applications to provide useful estimates of the temporal behavior of applications. Successively applying such abstractions allows to provide guarantees on the temporal behavior of the analyzed stream processing applications. However, current abstractions have limitations and do therefore not always suffice.

The temporal behavior of stream processing applications can intuitively and accurately be modeled using timed dataflow models. These dataflow models can be used to verify temporal constraints and to optimize the applications. However, current abstractions based on dataflow models have a limited scope and a limited accuracy. As a result, the dynamism that is supported by the used multiprocessor systems can not be analyzed properly. Furthermore, hierarchical composition and incremental design is not supported and next to that, the current class of supported run-time schedulers is limited.

In this thesis we introduced new abstractions for stream processing applications based on dataflow analysis techniques which alleviate these shortcomings. The introduced abstractions broaden the scope of dataflow analysis techniques, improve their accuracy and provide an additional level of abstractions which is compositional.

We continue this chapter with a summary of the results of the different chapters of this thesis. We then discuss the actual contributions in more detail in Section 7.2 and conclude this chapter with an indication of possible future research directions in Section 7.3.

7.1 SUMMARY

In Chapter 1 we have shown that aperiodic multiprocessor systems in combination with data-driven scheduling have big advantages over other types of multiprocessor systems. The realization of such an aperiodic multiprocessor system requires less constraints than a periodic multiprocessor system. The use of data-driven scheduling allows to cope with for example varying execution times, starting and stopping of applications and dynamic behavior of tasks. Methods to provide guarantees on the temporal behavior of these systems also exist. The most applicable analysis method for stream processing applications is the dataflow analysis method. Abstractions using dataflow models are often intuitive and allow to model a wide range of applications. For example cyclic applications are typically supported.

Abstractions are only meaningful if the conclusions which can be drawn from an abstraction also hold for the actual realization. In Chapter 2 an abstraction/refinement relation is introduced which can be used to define abstractions on the temporal behavior of stream processing applications. The refinement relation can be used hierarchically to obtain (easier) analyzable models. With respect to alternative refinement relations, a coupling of the functional behavior is added. This coupling of the functional behavior ensures that the temporal results obtained by using abstractions, can be correlated to the actual temporal behavior of the realization. Furthermore, we introduced a subset of dataflow models for which the refinement can be applied more easily because the functional behavior is by definition coupled correctly to the realization. For this subset the functional behavior of an application is orthogonal to its temporal behavior.

Where most dataflow analysis techniques focus on pipeline parallelism, few cover also data parallelism. As we showed in Chapter 3 the combination of pipeline and data parallelism is promising. The introduced method allows to model data parallelism accurately without having to replicate dataflow actors. The number of tokens on the self-edge of an actor is used to model the amount of replication of the corresponding task. A new transformation of an SDF model to an equivalent HSDF model is introduced to show that the used parameterized dataflow model is both conservative and accurate. With the presented dataflow model it is possible to determine data and pipeline parallelism and the required

buffer sizes simultaneously which makes the trade-off between them explicit. The analysis method is enabled by a new type of circular buffer with which the communication between tasks is implemented. The introduced circular buffer has support for multiple (data parallel) producers and multiple (data parallel) consumers. Furthermore, the buffer can be implemented such that the amount of synchronization operations are not dependent on the amount of data parallelism that is used.

In Chapter 4, a dataflow modeling technique is presented which exploits knowledge about the variation of execution times of tasks. The (σ, ρ) workload characterization is introduced to enable this dataflow analysis technique. This workload characterization forms an upper bound on the cumulative execution time of consecutive executions of a task. We showed that the (σ, ρ) characterization can be used to model the temporal behavior of tasks conservatively, by using an actor component consisting of two actors. We furthermore showed that the (σ, ρ) characterization can be combined with the temporal analysis of tasks that are scheduled by budget run-time schedulers. Also the combination of cyclic behavior of tasks with the (σ, ρ) characterization is supported. Next to that, a technique is presented which can derive a (σ, ρ) workload characterization from temporal information on a finite number of consecutive executions. The (σ, ρ) characterization makes explicit how much additional buffer locations are required to compensate for a maximum deviation from the average execution time of a task.

The scope of existing dataflow analysis techniques is limited to starvation-free schedulers. In Chapter 5 a new dataflow analysis technique is presented which broadens this scope to non-starvation-free schedulers. The method defines a dataflow analysis based alternative to existing analysis methods such as SymTA/S and MPA-RTC. The presented analysis techniques do not make assumptions on the graph topology and do therefore support cyclic graph topologies. The analysis method from Chapter 5 is the only method for data-driven multiprocessor systems which supports the combination of cyclic resource dependencies and cyclic data dependencies. Also, the analysis method can be used to achieve more accurate analysis results than other dataflow analysis techniques when multiprocessor systems with starvation-free schedulers are analyzed. Because of the support of cyclic graph topologies, buffer capacity constraints can be taken into account during the analysis and sufficient buffer sizes can be determined after the analysis. Finally, a linearized variant of the analysis flow is presented. This linear variant has a polynomial time-complexity and can therefore be used to explore the design space faster. Next to that, it allows to define a buffer sizing algorithm which can minimize buffer sizes by making the trade-off between the enabling jitter of tasks and the buffer sizes.

The last limitation of dataflow models that is addressed in this thesis concerns the compositionality of dataflow modeling techniques. Dataflow models are not compositional in general. The properties of compositions of components can in general only be deduced from the properties of the individual components, when their exact internal hierarchy is known. The lack of such a compositionality characteristic hampers the design and analysis of real-time applications. In Chapter 6

we have introduced the CTA model. This CTA model has strong hierarchical compositionality characteristics and can be used to model and analyze applications with arbitrary cyclic dependencies between components, periodic sources and sinks and latency constraints. The CTA model supports incremental design and independent implementability which helps to simplify the design and development of stream processing applications. Furthermore, the internal connections of a component can be hidden without worsening its temporal properties. Hiding can be used to obtain components of which only the external properties need to be known to a designer. We also provided an abstraction from an SDF graph to a conservative CTA model. With the obtained CTA model more pessimistic results are usually obtained than when using SDF graphs. However, unlike the SDF model, the CTA model has compositional temporal analysis properties.

7.2 CONTRIBUTIONS

The main contributions of this thesis are:

- *A new refinement relation to enable the abstractions as defined in this thesis.* In contrast to other refinement relations, also a conformance relation between the functional behavior of different abstractions is specified. This added conformance relation of the functional behavior ensures that events on the different abstractions levels can be correlated. Furthermore, it makes explicit under what conditions refinement is allowed. This new refinement relation can be used as a formal base for existing abstractions using dataflow models and it is also used as the formal base of the abstractions presented in this thesis. (Chapter 2)
- *A new analysis method to model and determine data parallelism.* This analysis method can be used to determine the required amount of data parallelism and make the trade-off between pipeline and data parallelism. Furthermore, a new type of buffer is introduced which enables the introduced modeling and corresponding implementation of data parallelism. This buffer is shared by the data parallel replications and thereby enables the combination of pipeline parallelism with data parallelism. (Chapter 3)
- *The (σ, ρ) workload characterization and corresponding analysis method.* The (σ, ρ) workload characterization contains information about the cumulative execution time of consecutive executions of a task and is a generalization of the WCET workload characterization. This new workload characterization can be used to combine knowledge about varying execution times of a task with existing dataflow analysis techniques. (Chapter 4)
- *A new analysis method for the analysis of systems with non-starvation-free run-time schedulers.* This new analysis method broadens the scope of dataflow analysis techniques to support non-starvation-free schedulers

such as static priority (non-)pre-emptive schedulers. Such non-starvation-free run-time schedulers complicate the analysis because the enablings of interfering tasks need to be known. The introduced analysis method for systems with non-starvation-free schedulers is the first analysis method that supports the combination of cyclic data dependencies and cyclic resource dependencies of applications that are executed data-driven. The introduced analysis flow can also be used to achieve an accuracy improvement over existing dataflow analysis methods when starvation-free schedulers are used. (Chapter 5)

- *The Compositional Temporal Analysis (CTA) model.* The CTA model has strong compositional properties regarding the temporal behavior of applications and can therefore be used to support incremental design of applications. The CTA model can serve as a higher abstraction level on top of dataflow models and allows to model latency and throughput constraints accurately. (Chapter 6)

7.3 POSSIBLE FUTURE DIRECTIONS

The new abstractions defined in this thesis can be extended in numerous directions. Possible future research directions are:

- The introduced analysis for systems with non-starvation-free schedulers opens a new field of research with a lot of opportunities to achieve more accurate analysis results than is possible with competitive analysis approaches. The accuracy as well as the applicability of the approach can be further improved. Furthermore, the combination with existing dataflow analysis methods is still limited. Optimizations regarding the response times of tasks as for example discussed in [RRE03] are possible. But also methods which exploit the properties of the used dataflow models to a larger extent have great potential [WHGB14, WHGB15]. We also expect that the used enabling characterization enables the analysis of systems with non-starvation-free schedulers in combination with more expressive dataflow models, such as CSDF [BEP96] and VRDF [WBS08a].
- The (σ, ρ) workload characterization is used to model variation of execution times of tasks. Another possible application of the (σ, ρ) workload characterization that is not yet explored can lie in the context of robustness. The (σ, ρ) workload characterization can be used to model and analyze the case in which the, for example measured, WCET of a task was optimistic. The dataflow model of the (σ, ρ) characterization can be used to determine the required buffer sizes which allow to recover from rare overshoots of the execution time of a task.
- The introduced refinement relation includes a relation with the functional behavior of the actual realization. Currently, this relation between the dif-

ferent levels of abstractions is fairly strict. An interesting direction would be to broaden and relax this coupling of the functional behavior without loosing the required coupling of the temporal behavior. Furthermore, it is not yet explored to what extent the refinement relation holds in the domain of non-starvation-free schedulers.

- The data parallelism approach is an interesting new step in determining parallelism of applications. In the future it would be very interesting if the results of the trade-off between pipeline and data parallelism can be combined with the mapping of tasks to processors. This mapping is currently seen as input to the analysis methods but can potentially influence the results significantly. Furthermore, currently only the combination with budget based run-time schedulers is supported. Combining data parallelism with other types of run-time schedulers is an open issue.
- The CTA model is in this thesis used as a model that can serve as an abstraction on top of SDF models. It would be interesting to know if the CTA model can also be used as an abstraction of applications with data-dependent and or conditional behavior. The combination with the VRDF [WBS08a, WBS08b] and or VPDF [WBS10] model is a potentially interesting direction. An extension of the CTA model might be required to support varying execution rates of tasks. A first attempt to use the CTA model for the hierarchical analysis of data-dependent applications can be found in [Geu15].

We believe that the in this thesis used dataflow analysis techniques have a good intuitive and strong analytical potential for the temporal analysis of aperiodic multiprocessor systems. We hope that the introduced abstractions emphasize the potential of dataflow analysis techniques. Furthermore, we hope that the newly identified research directions inspire researchers to work in the domain of real-time analysis of aperiodic multiprocessor systems and make use of these dataflow analysis techniques.

List of Acronyms

ADC Analog-to-Digital Converter	151
AEC Audio Echo Cancellation	107
BCET Best-Case Execution Time	119
BDF Boolean Dataflow	19
CPO complete partial order	25
CSDF Cyclo-Static Dataflow	4
CTA Compositional Temporal Analysis	155
DAB Digital Audio Broadcasting	147
DSP Digital Signal Processor	106
EDF Earliest Deadline First	8
FCFS First-Come First-Served	115
FIFO First-In First-Out	15
FM Frequency Modulation	147
HSDF Homogeneous Synchronous Dataflow	4
ILP Integer Linear Programming	61
KPN Kahn Process Network	15
LP Linear Programming	129
MCM Maximum Cycle Mean	19
MCR Maximum Cycle Ratio	54
MPA Modular Performance Analysis	7
MPA-RTC MPA with Real-Time Calculus	7
MPSoC Multiprocessor System-on-Chip	82

PBS Priority-Based Budget Scheduler 7

PEDF Partitioned Earliest Deadline First 4

SADF Scenario-Aware Data Flow 8

SDF Synchronous Dataflow 4

SPNP Static Priority Non-Preemptive 7

SPP Static Priority Preemptive 7

SymTA/S Symbolic Timing Analysis for Systems 7

TDM Time-Division Multiplexing 7

VPDF variable-Rate Phased Dataflow 19

VRDF Variable-Rate Dataflow 8

WCET Worst-Case Execution Time 2

List of Symbols

Notation	Description	Page
$\mathbb{N}$	Set of non-negative natural numbers	14
$\mathbb{N}^+$	Set of positive natural numbers	14
$\mathbb{Z}$	Set of integer numbers	14
$\mathbb{R}$	Set of real numbers	14
$\mathbb{R}^+$	Set of positive real numbers	14
$\mathbb{B}$	The boolean domain	14
$\lfloor a \rfloor$	The floor of a , the largest integer $b \in \mathbb{Z}$ for which $b \leq a$ holds	14
$\lceil a \rceil$	The ceil of a , the smallest integer $b \in \mathbb{Z}$ for which $b \geq a$ holds	14
G	Dataflow graph	17
V	Set of actors	17
E	Set of edges	17
v_a	Actor a	17
$\rho_a(i)$	Firing duration of firing i of actor v_a . We omit the firing number when the firing duration is constant.	17
e_{ij}	Edge from actor v_i to v_j	17
$\pi(e_{ij})$	Token production on edge e_{ij}	18
$\gamma(e_{ij})$	Token consumption on edge e_{ij}	18
$\delta(e_{ij})$	Number of initial tokens on edge e_{ij}	17
δ_{ij}	Shorthand notation for $\delta(e_{ij})$	17
q_a	Repetition factor of actor v_a . Specifies the relative firing frequencies	21
κ_a	Number of phases in the cyclo-static period of CSDF actor v_a .	18

	Notation	Description	Page
192	(σ, ρ)	Two-parameter workload that forms an upperbound on the cumulative execution time of a task	86
	τ_a	Task a	15
	B_a	Best-case execution time of task τ_a	119
	C_a	Worst-case execution time of task τ_a	78
	J_a	Enabling jitter of task τ_a	120
	P_a	Period of task τ_a	118
	$\hat{R}_a$	Maximum response time of task τ_a	121
	$\tilde{R}_a$	Linearized maximum response time of task τ_a	139
	$\check{R}_a$	Minimum response time of task τ_a	120
	$w_a(q)$	Busy period of q consecutive executions of task τ_a	120
	$\hat{\eta}_a(\Delta t)$	Maximum number of enablings of task τ_a in each interval Δt	120
	$\check{\theta}_a(n)$	Minimum time distance between n consecutive enablings of task τ_a	120
	$\psi(c_{ab})$	Number of containers that task τ_a produces on connection c_{ab}	23
	$\chi(c_{ab})$	Number of containers that task τ_b consumes on connection c_{ab}	23
	S_a	Budget of task τ_a	56
	Q_a	Interval in which budget is guaranteed for task τ_a	56
	P	Set of ports of a CTA component	161
	$\hat{r}(p)$	Maximum transfer rate of port p	159
	C	Connections a CTA component	161
	(i, j)	Connection from port i to port j	161
	c_{ij}	Connection from port i to port j , shorthand notation for (i, j)	161
	$\lambda(c)$	Rate ratio of connection c	161
	$\epsilon(c)$	Constant delay of connection c	161
	$\phi(c)$	Variable delay of connection c	161
	$\Delta(c)$	Total delay of connection c	157
	$\oplus$	Composition operator of the CTA model. $A \oplus B$ is the composition of components A and B	162

Bibliography

- [ABD⁺95] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell, and A.J. Wellings. Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.
- [ABR⁺93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.
- [AHG07] P. Alexander, D. Haley, and A. Grant. Outdoor Mobile Broadband Access with 802.11. *Communications Magazine*, 45(11):108–114, 2007.
- [BB01] B. Bhattacharya and S.S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. *IEEE Trans. on Signal Processing*, 49(10):2408–2421, 2001.
- [BBJS08] T. Bijlsma, M.J.G. Bekooij, P.G. Jansen, and G.J.M. Smit. Communication Between Nested Loop Programs via Circular Buffers in an Embedded Multiprocessor System. In *Proc. of the Int’l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 33–42, 2008.
- [BBS11] T. Bijlsma, M.J.G. Bekooij, and G.J.M. Smit. Circular Buffers with Multiple Overlapping Windows for Cyclic Task Graphs. In *Trans. on High-Performance Embedded Architectures and Compilers (HiPEAC): Volume 5, Issue 3*, 2011.
- [BCOQ92] F. Baccelli, G. Cohen, G. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: an Algebra for Discrete Event Systems*. Wiley, 1992.
- [BCP02] G. Bernat, A. Colin, and S.M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. of the Int’l Real-Time Systems Symp. (RTSS)*, pages 279–288, 2002.
- [BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996.

- [BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [BHM⁺05] M.J.G. Bekooij, R.J.H. Hoes, O. Moreira, P. Poplavko, M. Pastnak, B. Mesman, J.J.D. Mol, S. Stuijk, S.V. Gheorghita, and J.L. van Meerbergen. Dataflow Analysis for Real-Time Embedded Multiprocessor System Design. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3, pages 81–108. Springer, 2005.
- [Bij11] T. Bijlsma. *Automatic Parallelization of Nested Loop Programs for Non-Manifest Real-Time Stream Processing Applications*. PhD thesis, University of Twente, Enschede, The Netherlands, Enschede, the Netherlands, July 2011.
- [BL13] D. Bui and E.A. Lee. StreaMorph: A Case for Synthesizing Energy-efficient Adaptive Programs Using High-level Abstractions. In *Proc. of the Int’l Conf. on Embedded Software (EMSOFT)*. IEEE, 2013.
- [BMKD12] B. Bodin, A. Munier-Kordon, and B.D. De Dinechin. K-Periodic Schedules for Evaluating the Maximum Throughput of a Synchronous Dataflow Graph. In *Proc. of the Int’l Conf. on Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 152–159. IEEE, 2012.
- [BMKHB13] M. Benazouz, A. Munier-Kordon, T. Hujsa, and B. Bodin. Liveness Evaluation of a Cyclo-Static DataFlow Graph. In *Proc. of the Design Automation Conf. (DAC)*, page 3. ACM, 2013.
- [BMP⁺04] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastnak, and J. Van Meerbergen. Predictable Embedded Multiprocessor System Design. In *Proc. of the Int’l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 77–91. Springer, 2004.
- [BS91] F. Boussinot and R. De Simone. The Esterel Language. *Proc. of the IEEE*, 79(9):1293–1304, 1991.
- [BS11] M. Bamakhrama and T. Stefanov. Hard-Real-Time Scheduling of Data-Dependent Tasks in Embedded Streaming Applications. In *Proc. of the Int’l Conf. on Embedded Software (EMSOFT)*, pages 195–204. ACM, 2011.
- [BT97] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.
- [BT01] J.-Y. Le Boudec and P. Thiran. *Network Calculus: a Theory of Deterministic Queuing Systems for the Internet*, volume 2050. Springer, 2001.
- [BT13] A. Bouakaz and J.-P. Talpin. Buffer Minimization in Earliest-Deadline First Scheduling of Dataflow Graphs. In *Proc. of the Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 133–142. ACM, 2013.

- [BTV12] A. Bouakaz, J.-P. Talpin, and J. Vitek. Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications. In *Proc of the Int'l Conf. on Application of Concurrency to System Design (ACSD)*, pages 183–192. IEEE, 2012.
- [Buc93] J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.
- [But11] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.
- [CKT03] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, pages 190–195. IEEE, 2003.
- [Cru91a] R.L. Cruz. A Calculus for Network Delay. I. Network Elements in Isolation. *IEEE Trans. on Information Theory*, 37(1):114–131, 1991.
- [Cru91b] R.L. Cruz. A Calculus for Network Delay. II. Network Analysis. *IEEE Trans. on Information Theory*, 37(1):132–141, 1991.
- [Das04] A. Dasdan. Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms. *Trans. on Design Automation of Electronic Systems (TODAES)*, 9(4):385–418, 2004.
- [DBBL07] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised. *Real-Time Systems*, 35(3):239–272, 2007.
- [DBC⁺07] K. Denolf, M. Bekooij, J. Cockx, Verkest D., and H. Corporaal. Exploiting the Expressiveness of Cyclo-Static Dataflow to Model Multimedia Implementations. *EURASIP Journal on Advances in Signal Processing*, 2007.
- [DP02] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.
- [EBLP94] M. Engels, G. Bilson, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow: Model and Implementation. In *Proc. of the Asilomar Conf. on Signals, Systems and Computers*, volume 1, pages 503–507, 1994.
- [ETS06] ETSI. Digital Audio Broadcasting (DAB) to Mobile, Portable and Fixed Receivers. ETS 300 401 v1.4.1, 2006-06. <http://www.etsi.org>.
- [FAS⁺14] R.M.W. Frijns, S. Adyanthaya, S. Stuijk, J.P.M. Voeten, M.C.W. Geilen, R.R.H. Schifferers, and H. Corporaal. Timing Analysis of First-Come First-Served Scheduled Interval-Timed Directed Acyclic Graphs. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, 2014.
- [FKH⁺08] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S.S. Bhattacharyya. A Generalized Static Sata Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 189–198. ACM, 2008.

- [GBS05] M.C.W. Geilen, T. Basten, and S. Stuijk. Minimising Buffer Requirements of Synchronous Dataflow Graphs with Model Checking. In *Proc. of the Design Automation Conf. (DAC)*, pages 819–824. ACM, 2005.
- [Geu15] S.J. Geuns. *Specification and Compilation of Real-Time Stream Processing Applications*. PhD thesis, University of Twente, Enschede, The Netherlands, May 2015. To be published.
- [GGB⁺06] 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 *Formal Methods in Computer Aided Design (FMCAD)*, pages 68–75. IEEE, 2006.
- [GGBM91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proc. of the IEEE*, 79(9):1321–1336, 1991.
- [GGBS08] A.H. Ghamarian, M.C.W. Geilen, T. Basten, and S. Stuijk. Parametric Throughput Analysis of Synchronous Data Flow Graphs. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, pages 116–121, 2008.
- [GGS⁺06] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput Analysis of Synchronous Data Flow Graphs. In *Proc of the Int’l Conf. on Application of Concurrency to System Design (ACSD)*, pages 25–36. IEEE, 2006.
- [GHB13] S.J. Geuns, J.P.H.M. Hausmans, and M.J.G. Bekooij. Automatic Dataflow Model Extraction from Modal Real-Time Stream Processing Applications. In *Proc. of the Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 143–152. ACM, 2013.
- [GHB14a] S.J. Geuns, J.P.H.M. Hausmans, and M.J.G. Bekooij. Hierarchical Programming Language for Modal Multi-Rate Real-Time Stream Processing Applications. In *Int’l Conf. on Parallel Processing Workshops (ICPPW)*, pages 453–460, 2014.
- [GHB14b] S.J. Geuns, J.P.H.M. Hausmans, and M.J.G. Bekooij. Temporal Analysis Model Extraction for Optimizing Modal Multi-Rate Stream Processing Applications. In *Proc. of the Int’l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 21–30. ACM, 2014.
- [GHKB13] R. de Groote, P.K.F. Hölzenspies, J. Kuper, and H.J. Broersma. Back to Basics: Homogeneous Representations of Multi-Rate Synchronous Dataflow Graphs. In *Proc. of the Int’l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 35–46. IEEE, 2013.
- [GHKS14a] R. de Groote, P.K.F. Hölzenspies, J. Kuper, and G.J.M. Smit. Multi-Rate Equivalents of Cyclo-Static Synchronous Dataflow Graphs. In *Proc of the Int’l Conf. on Application of Concurrency to System Design (ACSD)*, pages 62–71. IEEE, 2014.

- [GHKS14b] R. de Groote, P.K.F. Hölzenspies, J. Kuper, and G.J.M. Smit. Single-Rate Approximations of Cyclo-Static Synchronous Dataflow Graphs. In *Proc. of the Int'l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 11–20. ACM, 2014.
- [GKBS12] R. de Groote, J. Kuper, H. Broersma, and G.J.M. Smit. Max-Plus Algebraic Throughput Analysis of Synchronous Dataflow Graphs. In *EUROMICRO Conf. on Software Engineering and Advanced Applications (SEAA)*, pages 29–38. IEEE, 2012.
- [GS10] M. Geilen and S. Stuijk. Worst-Case Performance Analysis of Synchronous Dataflow Scenarios. In *Proc. of the Int'l Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 125–134. ACM, 2010.
- [GTA06] M.I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162. ACM, 2006.
- [GTW11] M.C.W. Geilen, S. Tripakis, and M.H. Wiggers. The Earlier the Better: A Theory of Timed Actor Interfaces. In *Proc. of the Int'l Conf. on Hybrid Systems: Computation and Control (HSCC)*, April 2011.
- [HBC11] J.P.H.M. Hausmans, M.J.G. Bekooij, and H. Corporaal. Resynchronization of Cyclo-Static Dataflow Graphs. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, pages 1315–1320, 2011.
- [HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, 1991.
- [HGWB12] J.P.H.M. Hausmans, S.J. Geuns, M.H. Wiggers, and M.J.G. Bekooij. Compositional Temporal Analysis Model for Incremental Hard Real-Time System Design. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 185–194. ACM, 2012.
- [HGWB13a] J.P.H.M. Hausmans, S.J. Geuns, M.H. Wiggers, and M.J.G. Bekooij. Dataflow Analysis for Multiprocessor Systems with Non-Starvation-Free Schedulers. In *Proc. of the Int'l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 13–22. ACM, 2013.
- [HGWB13b] J.P.H.M. Hausmans, S.J. Geuns, M.H. Wiggers, and M.J.G. Bekooij. Two Parameter Workload Characterization for Improved Dataflow Analysis Accuracy. In *Proc. of the Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 117–126. IEEE, 2013.
- [HGWB14a] J.P.H.M. Hausmans, S.J. Geuns, M.H. Wiggers, and M.J.G. Bekooij. Temporal Analysis Flow Based on an Enabling Rate Characterization for Multi-Rate Applications Executed on MPSoCs with Non-Starvation-Free Schedulers. In *Proc. of the Int'l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 108–117. ACM, 2014.

- [HGWB14b] J.P.H.M. Hausmans, S.J. Geuns, M.H. Wiggers, and M.J.G. Bekooij. Unified Dataflow Model for the Analysis of Data and Pipeline Parallelism, and Buffer Sizing. In *Proc. of the Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 12–21. IEEE, 2014.
- [HHJ⁺05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - the SymTA/S Approach. In *IEE Proceedings Computers and Digital Techniques*, volume 152, pages 148–166, 2005.
- [HM06] T.A. Henzinger and S. Matic. An Interface Algebra for Real-Time Components. In *Proc. of the Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 253–266. IEEE Computer Society, 2006.
- [HT07] W. Haid and L. Thiele. Complex Task Activation Schemes in System Level Performance Analysis. In *Proc. of the Int'l Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 173–178. IEEE, 2007.
- [HWM⁺08] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Applying Dataflow Analysis to Dimension Buffers for Guaranteed Performance in Networks on Chip. In *Proc. of the Int'l Symp. on Networks-on-Chip (NOCS)*, pages 211–212. IEEE Computer Society, 2008.
- [HWM⁺09] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis. *Computers & Digital Techniques*, 3(5):398–412, 2009.
- [HZS01] X.S. Hu, T. Zhou, and E.H.M. Sha. Estimating Probabilistic Timing Performance for Real-Time Embedded Systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(6):833–844, 2001.
- [Jer05] M. Jersak. *Compositional Performance Analysis for Complex Embedded Applications*. PhD thesis, University of Braunschweig-Institute of Technology, 2005.
- [JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.
- [JPTY08] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. Cyclic Dependencies in Modular Performance Analysis. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 179–188. ACM, 2008.
- [JSL95] Pino J.L., Bhattacharyya S.S., and E.A. Lee. A Hierarchical Multiprocessor Scheduling Framework For Synchronous Dataflow Graphs. Technical Report UCB/ERL M95/36, University of California, Berkeley, 1995.
- [Kah74] G. Kahn. The semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress*, volume 74, pages 471–475, 1974.
- [KB03] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proc. of the IEEE*, 91(1):112–126, 2003.

- [KJL13] A.E. Kiasari, A. Jantsch, and Z. Lu. Mathematical Formalisms for Performance Evaluation of Networks-on-Chip. *ACM Computing Surveys*, 45(3):38:1–38:41, 2013.
- [KM77] G. Kahn and D.B. MacQueen. Coroutines and Networks of Parallel Processes. In *Proc. of the IFIP Congress*, volume 77, pages 993–998, 1977.
- [KM08] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 114–124. ACM, 2008.
- [Kop11] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.
- [Lee91] E. Lee. Consistency in Dataflow Graphs. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):223–235, 1991.
- [Leh90] J.P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proc. of the Int’l Real-Time Systems Symp. (RTSS)*, volume 90, pages 201–209, 1990.
- [LL73] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.
- [LM87] E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.
- [LMC12] A. Lele, O. Moreira, and P. Cuijpers. A New Data Flow Analysis Model for TDM. In *Proc. of the Int’l Conf. on Embedded Software (EMSOFT)*, pages 237–246. ACM, 2012.
- [LP95] E. Lee and T. Parks. Dataflow Process Networks. *Proc. of the IEEE*, 83(5):773–801, 1995.
- [LPT09] K. Lampka, S. Perathoner, and L. Thiele. Analytic Real-Time Analysis and Timed Automata: A Hybrid Method for Analyzing Embedded Real-Time Systems. In *Proc. of the Int’l Conf. on Embedded Software (EMSOFT)*, pages 107–116. ACM, 2009.
- [LSV98] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.
- [MB07] O. Moreira and M.J.G. Bekooij. Self-Timed Scheduling Analysis for Real-Time Applications. *EURASIP Journal on Advances in Signal Processing*, 2007.
- [MBBM07] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. Practical and Accurate Throughput Analysis with the Cyclo Static Dataflow Model. In *Proc. of the Int’l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 238–245. IEEE, 2007.

- [MBBM08] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. Cache Aware Mapping of Streaming Applications on a Multiprocessor System-on-Chip. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, pages 300–305. IEEE, 2008.
- [MBGS10] O. Moreira, T. Basten, M. Geilen, and S. Stuijk. Buffer Sizing for Rate-Optimal Single-Rate Data-Flow Scheduling Revisited. *IEEE Trans. on Computers*, 59(2):188–201, 2010.
- [MKT04] A. Maxiaguine, S. Künzli, and L. Thiele. Workload Characterization Model for Tasks with Variable Execution Demand. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, volume 2, pages 1040–1045. IEEE Computer Society, 2004.
- [MNS10] S. Meijer, H. Nikolov, and T. Stefanov. Combining Process Splitting and Merging Transformations for Polyhedral Process Networks. In *Proc. of the Symp. on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 97–106. IEEE, 2010.
- [MVB07] O. Moreira, F. Valente, and M.J.G. Bekooij. Scheduling Multiple Independent Hard-Real-Time Jobs on a Heterogeneous Multiprocessor. In *Proc. of the Int’l Conf. on Embedded Software (EMSOFT)*, pages 57–66, 2007.
- [NE12] M. Negrean and R. Ernst. Response-Time Analysis for Non-Preemptive Scheduling in Multi-Core Systems with Shared Resources. In *Int’l Symp. on Industrial Embedded Systems (SIES)*, pages 191–200. IEEE, 2012.
- [NG93] Q. Ning and G.R. Gao. A Novel Framework of Register Allocation for Software Pipelining. In *Proc. of the Symp. on Principles of Programming Languages (POPL)*, pages 29–42. ACM, 1993.
- [Ope07] OpenMP ARB. OpenMP Application Program Interface, 2013-07. <http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>.
- [Ost95] J.S. Ostroff. Abstraction and composition of discrete real-time systems. *Proc. of CASE*, 95:370–380, 1995.
- [PBB⁺03] 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 *Proc. of the Int’l Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 63–72. ACM, 2003.
- [PLT11] S. Perathoner, K. Lampka, and L. Thiele. Composing Heterogeneous Components for System-wide Performance Analysis. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, 2011.
- [PPL95] T.M. Parks, J.L. Pino, and E.A. Lee. A Comparison of Synchronous and Cyclo-Static Dataflow. In *Proc. of the Asilomar Conf. on Signals, Systems and Computers*, volume 1, pages 204–210. IEEE, 1995.

- [PWT⁺07] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M.G. Harbour. Influence of Different System Abstractions on the Performance Analysis of Distributed Real-Time Systems. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 193–202. ACM, 2007.
- [QHE12] S. Quinton, M. Hanke, and R. Ernst. Formal Analysis of Sporadic Overload in Real-Time Systems. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, pages 515–520. IEEE, 2012.
- [Rei68] R. Reiter. Scheduling Parallel Computations. *Journal of the ACM (JACM)*, 15(4):590–599, 1968.
- [RRE03] K. Richter, R. Racu, and R. Ernst. Scheduling Analysis Integration for Heterogeneous Multiprocessor SoC. In *Proc. of the Int'l Real-Time Systems Symp. (RTSS)*, pages 236–245. IEEE, 2003.
- [SAA⁺04] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A.K. Mok. Real time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.
- [SB00] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.
- [SB09] J. Staschulat and M. Bekooij. Dataflow Models for Shared Memory Access Latency Analysis. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 275–284. ACM, 2009.
- [SBW09] M. Steine, M. Bekooij, and M. Wiggers. A Priority-Based Budget Scheduler with Conservative Dataflow Model. In *Proc. of the Euromicro Conf. on Digital System Design, Architectures, Methods and Tools (DSD)*, pages 37–44. IEEE, 2009.
- [SDI⁺08] S. Stein, J. Diemer, M. Ivers, S. Schliecker, and R. Ernst. On the Convergence of the SymTA/S Analysis. *TU Braunschweig, Braunschweig, Germany*, 2008.
- [SE09] S. Schliecker and R. Ernst. A Recursive Approach to End-To-End Path Latency Computation in Heterogeneous Multiprocessor Systems. In *Proc. of the Int'l Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 433–442. ACM, 2009.
- [SGB06] S. Stuijk, M. Geilen, and T. Basten. Exploring Trade-offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs. In *Proc. of the Design Automation Conf. (DAC)*, pages 899–904. ACM, 2006.
- [SGB08] S. Stuijk, M. Geilen, and T. Basten. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Trans. on Computers*, 57(10):1331–1345, 2008.
- [SKD02] T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances. In *Proc. of the Int'l Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 7–12. ACM, 2002.

- [SRIE08] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems. In *Proc. of the Int'l Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 185–190. ACM, 2008.
- [SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems*, 1(1):27–60, 1989.
- [SV98] D. Stiliadis and A. Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *IEEE/ACM Trans. on Networking (ToN)*, 6(5):611–624, 1998.
- [SYBT13] L. Schor, H. Yang, L. Bacivarov, and L. Thiele. Expandable Process Networks to Efficiently Specify and Explore Task, Data, and Pipeline Parallelism. In *Proc. of the Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2013.
- [TBG⁺13] S. Tripakis, D. Bui, M.C.W. Geilen, B. Rodiers, and E.A. Lee. Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. *ACM Trans. on Embedded Computing Systems (TECS)*, 12(3):83, 2013.
- [TBW94] K. Tindell, A. Burns, and A. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2):133–151, 1994.
- [TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *Proc. IEEE Int'l Symp. on Circuits and Systems*, volume 4, pages 101–104. IEEE, 2000.
- [TDS⁺95] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu. Probabilistic Performance Guarantee for Real-Time Tasks With Varying Computation Times. In *Proc. of the Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 164–173. IEEE, 1995.
- [TGB⁺06] 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 *Proc. of the Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 185–194. IEEE, 2006.
- [TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the Int'l Conf. on Compiler Construction (CC)*, pages 179–196. Springer, 2002.
- [TLHL11] S. Tripakis, B. Lickly, T.A. Henzinger, and E.A. Lee. A Theory of Synchronous Relational Interfaces. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 33(4):14, 2011.
- [TS09] L. Thiele and N. Stoimenov. Modular Performance Analysis of Cyclic Dataflow Graphs. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 127–136. ACM, 2009.

- [TWS06] L. Thiele, E. Wandeler, and N. Stoimenov. Real-Time Interfaces for Composing Real-Time Systems. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 34–43. ACM, 2006.
- [WBJ06] M.H. Wiggers, M.J.G. Bekooij, P.G. Jansen, and G.J.M. Smit. Efficient Computation of Buffer Capacities for Multi-Rate Real-Time Systems with Back-Pressure. In *Proc. of the Int'l Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 10–15. ACM, 2006.
- [WBJ07] M.H. Wiggers, M.J.G. Bekooij, P.G. Jansen, and G.J.M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Real-Time Systems with Back-Pressure. In *Proc. of the Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 281–292. IEEE Computer Society, 2007.
- [WBS07a] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Proc. of the Design Automation Conf. (DAC)*, page 663. ACM, 2007.
- [WBS07b] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Modelling Run-Time Arbitration by Latency-Rate Servers in Dataflow Graphs. In *Proc. of the Int'l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 11–22. ACM, 2007.
- [WBS08a] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Buffer Capacity Computation for Throughput Constrained Streaming Applications with Data-Dependent Inter-Task Communication. In *Proc. of the Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 183–194. IEEE Computer Society, 2008.
- [WBS08b] 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 *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, pages 640–645. IEEE, 2008.
- [WBS09] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Monotonicity and Run-Time Scheduling. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 177–186. ACM, 2009.
- [WBS10] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Buffer Capacity Computation for Throughput-Constrained Modal Task Graphs. *ACM Trans. on Embedded Computing Systems (TECS)*, 10(2):17, 2010.
- [WHGB14] P.S. Wilmanns, J.P.H.M. Hausmans, S.J. Geuns, and M.J.G. Bekooij. Accuracy Improvement of Dataflow Analysis for Cyclic Stream Processing Applications Scheduled by Static Priority Preemptive Schedulers. In *Proc. of the Euromicro Conf. on Digital System Design, Architectures, Methods and Tools (DSD)*, pages 623–630. IEEE, 2014.
- [WHGB15] P.S. Wilmanns, J.P.H.M. Hausmans, S.J. Geuns, and M.J.G. Bekooij. Buffer Sizing to Reduce Interference and Increase Throughput of Real-Time Stream Processing Applications. In *Int'l Symp. on Real-Time Computing (ISORC)*. IEEE, 2015. To appear.

- [Wig09] M. H. Wiggers. *Aperiodic Multiprocessor Scheduling for Real-Time Stream Processing Applications*. PhD thesis, University of Twente, Enschede, The Netherlands, June 2009.
- [WTVL06] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieveise. System Architecture Evaluation Using Modular Performance Analysis: a Case Study. *Int'l Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006.

List of Publications

- [JH:1] Joost P.H.M. Hausmans, Stefan J. Geuns, Maarten H. Wiggers, and Marco J.G. Bekooij. Unified Dataflow Model for the Analysis of Data and Pipeline Parallelism, and Buffer Sizing. In *Proc. of the Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 12–21. IEEE, 2014.
- [JH:2] Joost P.H.M. Hausmans, Stefan J. Geuns, Maarten H. Wiggers, and Marco J.G. Bekooij. Temporal Analysis Flow Based on an Enabling Rate Characterization for Multi-Rate Applications Executed on MPSoC with Non-Starvation-Free Schedulers. In *Proc. of the Int'l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 108–117. ACM, 2014.
- [JH:3] Joost P.H.M. Hausmans, Stefan J. Geuns, Maarten H. Wiggers, and Marco J.G. Bekooij. Dataflow Analysis for Multiprocessor Systems with Non-Starvation-Free Schedulers. In *Proc. of the Int'l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 13–22. ACM, 2013.
- [JH:4] Joost P.H.M. Hausmans, Stefan J. Geuns, Maarten H. Wiggers, and Marco J.G. Bekooij. Two Parameter Workload Characterization for Improved Dataflow Analysis Accuracy. In *Proc. of the Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 117–126. IEEE, 2013.
- [JH:5] Joost P.H.M. Hausmans, Stefan J. Geuns, Maarten H. Wiggers, and Marco J.G. Bekooij. Compositional Temporal Analysis Model for Incremental Hard Real-Time System Design. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 185–194. ACM, 2012.
- [JH:6] Joost P.H.M. Hausmans, Marco J.G. Bekooij, and Henk Corporaal. Resynchronization of Cyclo-Static Dataflow Graphs. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, pages 1315–1320. IEEE, 2011.
- [JH:7] Philip S. Wilmanns, Joost P.H.M. Hausmans, Stefan J. Geuns, and Marco J.G. Bekooij. Buffer Sizing to Reduce Interference and Increase Throughput of Real-Time Stream Processing Applications. In *Int'l Symp. on Real-Time Computing (ISORC)*. IEEE, 2015. To appear.

- [JH:8] Philip S. Wilmanns, Joost P.H.M. Hausmans, Stefan J. Geuns, and Marco J.G. Bekooij. Accuracy Improvement of Dataflow Analysis for Cyclic Stream Processing Applications Scheduled by Static Priority Preemptive Schedulers. In *Proc. of the Euromicro Conf. on Digital System Design, Architectures, Methods and Tools (DSD)*, pages 623–630. IEEE, 2014.
- [JH:9] Stefan J. Geuns, Joost P.H.M. Hausmans, and Marco J.G. Bekooij. Hierarchical Programming Language for Modal Multi-Rate Real-Time Stream Processing Applications. In *Int'l Conf. on Parallel Processing Workshops (ICPPW)*, pages 453–460, 2014.
- [JH:10] Stefan J. Geuns, Joost P.H.M. Hausmans, and Marco J.G. Bekooij. Temporal Analysis Model Extraction for Optimizing Modal Multi-Rate Stream Processing Applications. In *Proc. of the Int'l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 21–30. ACM, 2014.
- [JH:11] Stefan J. Geuns, Joost P.H.M. Hausmans, and Marco J.G. Bekooij. Automatic Dataflow Model Extraction from Modal Real-Time Stream Processing Applications. In *Proc. of the Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 143–152. ACM, 2013.
- [JH:12] Stefan J. Geuns, Joost P.H.M. Hausmans, and Marco J.G. Bekooij. Sequential Specification of Time-Aware Stream Processing Applications. *ACM Trans. on Embedded Computing Systems (TECS)*, 12(1s):35, 2013.
- [JH:13] Stefan J. Geuns, Joost P.H.M. Hausmans, and Marco J.G. Bekooij. Sequential Specification of Time-Aware Stream Processing Applications (Extended abstract). In *Proc. of the Symp. on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 56–56. IEEE, 2012.
- [JH:14] Stefan J. Geuns, Joost P.H.M. Hausmans, and Marco J.G. Bekooij. Mapping of Modal Applications given Throughput and Latency Constraints. In *Proc. of the Int'l Conf. on Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 372–379. IEEE, 2011.
- [JH:15] André M. van Renssen, Stefan J. Geuns, Joost P.H.M. Hausmans, Wouter Poncin, and Reinder J. Bril. On Utilization Bounds for a Periodic Resource under Rate Monotonic Scheduling. *Proc. Work-in-Progress (WiP) of the Euromicro Conf. on Real-Time Systems (ECRTS)*, 2009.

ISBN 978-90-365-3853-4

9 789036 538534

STELLINGEN

behorende bij het proefschrift

Abstractions for Aperiodic Multiprocessor Scheduling *of Real-Time Stream Processing Applications*

door Joost P.H.M. Hausmans,
te verdedigen op vrijdag 24 april 2015

- 1 — Bestaande dataflow verfijningstheorieën hebben belangrijke tekortkomingen waardoor ze in de praktijk niet toepasbaar zijn. *(Hoofdstuk 2)*
- 2 — Door gebruik te maken van aperiodieke schedules is het mogelijk om real-time garanties te geven zonder dat er gebruik wordt gemaakt van alleen de maximale uitvoeringstijden van taken. *(Hoofdstuk 4)*
- 3 — In een typisch embedded multiprocessor systeem wordt momenteel gebruik gemaakt van non-starvation-free schedulers terwijl alles erop wijst dat dit type scheduler de analyse aanzienlijk compliceert en maar weinig voordeel oplevert. *(Hoofdstuk 1 en 5)*
- 4 — Het is mogelijk om periodieke abstracties met krachtige compositionele en wiskundige eigenschappen te maken van aperiodieke multiprocessor schedules. *(Hoofdstuk 6)*
- 5 — Wetenschap vereist het maken van abstracties.
- 6 — De intuïtie van een oplossing is minstens zo belangrijk als de formele uitwerking ervan.
- 7 — Controverse stimuleert wetenschappelijke vooruitgang.
- 8 — Iets moeilijks eenvoudiger maken is gewoonlijk veel moeilijker dan iets eenvoudigs moeilijker maken.
- 9 — Creativiteit kan niet geleerd worden, alleen gestimuleerd.
- 10 — Het belang van muziek, cultuur en kunst kan niet gekwantificeerd worden.

PROPOSITIONS

accompanying the thesis

Abstractions for Aperiodic Multiprocessor Scheduling *of Real-Time Stream Processing Applications*

by Joost P.H.M. Hausmans,
to be defended on Friday 24th April, 2015

- 1 — Existing dataflow refinement theories have important shortcomings that make them inapplicable in practice. *(Chapter 2)*
- 2 — By making use of aperiodic schedules it is possible to give real-time guarantees without having to use only the worst-case execution times of tasks. *(Chapter 4)*
- 3 — A typical embedded multiprocessor system currently uses non-starvation-free schedulers while there are indications that this type of scheduler complicates the analysis significantly and the advantages are limited. *(Chapter 1 and 5)*
- 4 — It is possible to make periodic abstractions with powerful compositional and mathematical features from aperiodic multiprocessor schedules. *(Chapter 6)*
- 5 — Science requires the use of abstractions.
- 6 — The intuition of a solution is at least as important as its formal derivation.
- 7 — Controversy stimulates scientific progress.
- 8 — Simplifying something difficult is in general much harder than complicating something simple.
- 9 — Creativity can not be taught, only stimulated.
- 10 — The importance of music, culture and art can not be quantified.