

Accurate analysis of real-time stream processing applications

using dataflow models and timed automata



Guus Kuiper

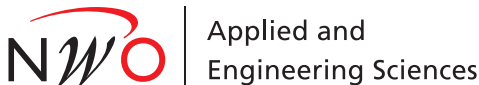
Members of the graduation committee:

Prof. dr. ir. M. J. G. Bekooij	University of Twente (promotor)
Prof. dr. ir. G. J. M. Smit	University of Twente
Dr. A. K. I. Remke	University of Twente
Prof. dr. ing. J. Castrillon	Technische Universität Dresden
Prof. dr. ir. T. Basten	Eindhoven University of Technology
Dr. ir. E. de Groot	University of Twente (special expert)
Prof. dr. J. N. Kok	University of Twente (chairman and secretary)

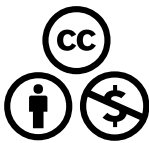
UNIVERSITY OF TWENTE.

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

IDS Ph.D. Thesis Series No. 18-463
Institute on Digital Society
PO Box 217, 7500 AE Enschede, The Netherlands



This work is part of the research program 'Integrated Design Approach for Safety-Critical Real-Time Automotive Systems' with project number 12698, which is financed by NXP and the Netherlands Organization for Scientific Research (NWO). Project leader : Prof. dr. ir. M.J.G. Bekooij



Copyright © 2019 Guus Kuiper, Enschede, The Netherlands. This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/4.0/deed.en_US.

This thesis was typeset using \LaTeX , TikZ, and Kile. This thesis was printed by Gildeprint Drukkerijen, The Netherlands.

ISBN 978-90-365-4541-9
ISSN 2589-4730; IDS Ph.D. Thesis Series No. 18-463
DOI 10.3990/1.9789036545419

ACCURATE ANALYSIS OF REAL-TIME STREAM
PROCESSING APPLICATIONS

USING DATAFLOW MODELS AND TIMED AUTOMATA

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. T. T. M. Palstra,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 8 maart 2019 om 16.45 uur

door

Guus Kuiper

geboren op 29 januari 1988
te Doetinchem

Dit proefschrift is goedgekeurd door:

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

ABSTRACT

Autonomous cars are an example of a safety-critical Cyber-Physical System (CPS). In such a CPS, there is a very complicated interaction between the continuous-time physical environment and the discrete-time embedded control system of the car using its sensors and actuators. The sensors periodically sample the physical properties of the environment and produce streams of data samples. The actuators consume data from streams and expect to receive this data periodically from the digital control system. In this thesis, we focus on the modem system in autonomous cars, which is used for vehicle-to-vehicle communication. Such a safety-critical embedded system must be analyzed in order to verify whether temporal constraints will be satisfied. During the design of such a system, bounds must therefore be derived for the interval of time it takes for a real-time system to produce an output as a response to an input event.

Nowadays, these embedded systems are often implemented on multiple processors. The software that is running on such a system is split into tasks, that are connected by First-In-First-Out (FIFO) communication buffers. Some of these tasks execute conditionally, depending on the value of the incoming data. The incoming data can put the application in a certain *mode*, in which only a subset of tasks actually executes. Run-time schedulers determine when a task is allowed to execute on a processor.

The available analysis methods for this type of systems have severe shortcomings. The analysis results of these methods are insufficiently accurate. Furthermore, the run-time of the analysis methods is often unacceptably high. Moreover, most of these methods do not support applications containing cyclic dependencies and dynamic applications containing conditionally executed tasks.

The analysis of applications containing cyclic dependencies naturally fits to dataflow analysis approaches. However, for dynamic applications where tasks are scheduled using budget schedulers, the accuracy of dataflow analysis can be low. We therefore introduce *locks* that cause additional sequence constraints in these dynamic applications. These locks prevent interference between tasks in different modes, which can reduce the latency for dynamic applications.

To enable the analysis for non-starvation-free schedulers, we also introduce *barriers*. These non-starvation-free schedulers are a broader class than budget schedulers. The barriers ensure that all inputs of a mode are available before any tasks in that mode can start. The combination of locks and barriers allows for compositional analysis of modes. This means that each mode can be analyzed in isolation, even if these modes belong to a different level in the hierarchy of an application. As a

result, existing analysis methods for non-dynamic applications can be used for each mode in isolation.

vi

To obtain more accurate analysis results, we introduce a *transformation* from dataflow graphs into timed automata. Using these timed automata, accurate analysis can be performed by taking the correlation of firing durations into account of consecutive firings of the same task, but also of different tasks. Current dataflow analysis techniques abstract from this specific type of correlation by means of over-approximation, whereas we can take this correlation into account with timed automata, without over-approximation. Moreover, we enable the analysis of systems with out-of-order communication by annotating tokens with a token index order, instead of only using the token arrival order. Out-of-order communication can be a result of multiple executions of the same tasks in parallel. By consuming tokens in index order, the functional behavior of the graph is orthogonal to the arrival order of the tokens.

In order to accurately analyze the consequences of pre-emptive task scheduling on the temporal behavior of an application, we make use of model checking of timed automata. In this model we include correlation of the executions of different tasks, sharing the same resource. However, modeling task scheduling directly in a timed automata model can make the scheduling problem undecidable. We address this issue by either an over-approximation in the timed automata model, or by allowing the model checker to over-approximate reachability. The run-time of the analysis method is improved by combining model checking of timed automata with dataflow analysis. Moreover, latency is minimized by using this *hybrid analysis/optimization* method, where we introduce additional *sequence constraints* to limit the interference between tasks.

SAMENVATTING

Autonome auto's zijn een voorbeeld van een veiligheidskritisch Cyber-Physical System (CPS). In zo een CPS is er een zeer gecompliceerde interactie tussen de continue tijd van de fysieke omgeving van het systeem en de discrete tijd van het embedded controlesysteem in een auto. Sensoren samplen de fysieke eigenschappen van de omgeving periodiek en produceren hier data stromen van. Actuatoren lezen data uit stromen, waarbij ze deze data periodiek verwachten te ontvangen van het digitale besturingssysteem. In dit proefschrift concentreren we ons op het modern systeem binnen autonome auto's, die wordt gebruikt voor voertuig-naar-voertuigcommunicatie. Een dergelijk veiligheidskritisch embedded systeem moet worden geanalyseerd om na te gaan of aan eisen wat betreft timing wordt voldaan. Tijdens het ontwerp van een dergelijk systeem moeten daarom grenzen worden afgeleid voor het tijdsinterval dat een real-time systeem nodig heeft om een reactie te produceren op een gebeurtenis.

Tegenwoordig worden deze embedded systemen vaak geïmplementeerd op meerdere processoren. De software die op een dergelijk systeem wordt uitgevoerd, is opgesplitst in taken die zijn verbonden door First-In-First-Out (FIFO) communicatiebuffers. Sommige van deze taken worden conditioneel uitgevoerd, afhankelijk van de waarde van de binnenkomende data. De inkomende gegevens kunnen de applicatie in een bepaalde *modus* plaatsen, waarin slechts een deel van de taken daadwerkelijk wordt uitgevoerd. *Run-time schedulers* bepalen wanneer een taak mag worden uitgevoerd op een processor. Helaas hebben de beschikbare analysemethoden voor dit soort systemen ernstige tekortkomingen.

Een van de problemen van de analysemethoden is dat de analyseresultaten voor dit type systemen zijn onvoldoende nauwkeurig. Bovendien is de rekentijd die de analysemethoden nodig hebben om tot een resultaat te komen vaak onaanvaardbaar hoog. Bovendien ondersteunen de meeste van deze methoden geen applicaties die cyclische afhankelijkheden bevatten en dynamische applicaties met conditioneel uitgevoerde taken.

De analyse van applicaties die cyclische afhankelijkheden bevatten, past van nature bij dataflow-analysetechnieken. Voor dynamische applicaties waarbij taken worden gepland met behulp van *budget schedulers*, kan de nauwkeurigheid van analyse echter laag zijn. Daarom introduceren we *locks* die extra beperkingen in de executievolgorde van taken toevoegen in deze dynamische applicaties. Deze *locks* voorkomen interferentie tussen taken in verschillende modi, wat de latentie voor dynamische applicaties kan verminderen.

Om de analyse voor *non-starvation-free schedulers* mogelijk te maken, introduceren

we ook *barriers*. Deze schedulers zijn een bredere klasse dan budget gebaseerde schedulers. De *barriers* zorgen ervoor dat alle inkomende data van een modus beschikbaar is voordat taken binnen die modus worden uitgevoerd. De combinatie van locks en *barriers* maakt compositieanalyse van modi mogelijk. Dit betekent dat elke modus afzonderlijk kan worden geanalyseerd, zelfs als deze modi tot een ander niveau in de hiërarchie van een applicatie behoren. Echter, kunnen bestaande analysemethoden voor niet-dynamische applicaties voor elke modus afzonderlijk worden gebruikt.

Voor meer accurate analyseresultaten introduceren we een transformatie van dataflow-grafen naar *timed automata*. Met behulp van deze *timed automata* kan een nauwkeurige analyse worden uitgevoerd door de correlatie van vuurtijden mee te nemen, zowel bij opeenvolgende vuringen van dezelfde taak, als tussen verschillende taken. Huidige dataflow-analysetechnieken abstraheren van deze correlatie door middel van over-approximatie, terwijl we deze correlatie mee kunnen nemen met *timed automata*, zonder over-approximatie. Bovendien maken we de analyse mogelijk van systemen met niet-sequentiële communicatie door tokens met een *token-indexvolgorde* te annoteren, in plaats van alleen de token-aankomstvolgorde te gebruiken. Niet-sequentiële communicatie kan het resultaat zijn van meerdere uitvoeringen van dezelfde taken in parallel. Door tokens in indexvolgorde te consumeren, is het functionele gedrag van de grafiek orthogonaal ten opzichte van de aankomstvolgorde van de tokens.

Om de consequenties van onderbreekbare taakplanning op het temporele gedrag van een applicatie nauwkeurig te analyseren, maken we gebruik van modelcontrole van *timed automata*. Het rechtstreeks modelleren van taakplanning in een *timed automata*model kan het planningsprobleem onbeslisbaar maken. We pakken dit probleem aan door een overschatting in het *timed automata*model, of door de modelcontrole een overschatting te laten maken. De looptijd van de analysemethode is verbeterd door modelcontrole van getimede automaten te combineren met dataflow-analyse. Bovendien wordt de latentie geminimaliseerd door gebruik te maken van deze hybride analysemethode, waarbij we extra volgordebependingen introduceren om de interferentie tussen taken te verminderen.

DANKWOORD

Eindelijk is het dan zo ver; mijn promotie komt in zicht, en de laatste hand wordt gelegd aan dit boekwerk. Vier jaar lang heb ik aan mijn onderzoek morgen werken, met als eindresultaat dit boekwerk. Deze periode was heel anders dan het jaar wat ik nu bijna al binnen het bedrijfsleven aan de slag ben. Nu werk ik bij Demcon/Bond met een groot multidisciplinair team samen aan een groot doel. Bij een promotie ga je toch meer voor eigen resultaten, uitgedrukt in de vorm van publicaties. Mijn promotie was dus meer werk op een eilandje. Echter had ik dit niet kunnen doen zonder hulp of steun. Vandaar dat ik de nodige mensen wil bedanken.

Als eerste Marco, mijn promotor. Lang geleden begon ik mijn afstudeeropdracht onder jouw supervisie. Niet in de analysetechnieken, maar hardware / chip design. Ook hier ging het al over het kunnen geven van garanties, de “rotonde met stoplichten”, maar dan voor latency voor de communicatie binnen een chip. Hier heb je mij toch langzaam overtuigd van de analysewereld. Deze hemel waarin alle componenten zich ideaal gedragen. Terwijl er in de echte wereld onverwacht, vanalles mis kan gaan, en aannames die je maakt ineens niet meer op kunnen gaan. Jouw diepgaande technische kennis zorgde dat we elke discussie weer een stapje verder kwamen richting een paper. Daarnaast kan ik ook jouw uitgebreide feedback waarderen op het minst favoriete onderdeel van de promotie, het daadwerkelijk opschrijven van de onderzoeksresultaten. Dit is vooral knap gezien je maar part-time, één dag in de week, professor bent bij CAES naast je reguliere werk bij NXP. Wat voor type scheduler wordt er voor deze Marco resource gebuikt om je tijd zo te verdelen tussen de vele promovendie en afstudeerders? Parallellisatie zou dit scheduling probleem een stuk makkelijker, maar is in dit geval lastig te implementeren zolang het klonen van mensen nog niet echt een ding is.

Hiernaast zijn er meer mensen die zich inhoudelijk met mijn promotie bezig hebben gehouden. Philip, Joost, en Stefan waren de dataflow experts bij CAES die mij goed geholpen hebben met mijn papers, ook al zaten jullie ver weg in Eindhoven bij NXP, wat de samenwerking er niet makkelijker op maakte. Als we dan wel een keer bij elkaar zaten bij de SCOPES conferentie, hadden jullie veel feedback op mijn conferentiepresentatie. Dit commentaar heb ik toen een paar uur voor mijn presentatie nog meer net allemaal kunnen verwerken voordat ik de echte presentatie mocht houden. Daarnaast was Robert met zijn dataflow expertise wel altijd in de buurt, en had hij altijd een interessant dataflow probleem om de tijd mee te vullen. Ook de afstudeerders; Oscar, Viktorio, Daniel, Mark en Oguz, mag ik niet vergeten. Al waren jullie onderzoeken niet direct relevant voor mijn promotie, jullie mogen begeleiden was wel een leuke afwisseling op mijn eigen onderzoek.

Naast de inhoudelijk ondersteuning bracht mijn vakgroep, CAES, nog zoveel meer. Met Gerard aan het hoofd van de vakgroep was CAES een gezellige groep om deel van uit te maken, waar ik 5 jaar lang met veel plezier heb gezeten. Vooral de vrijdagmiddagborrels en koffiepauzes waren een goede manier om vernieuwde inspiratie op te doen. Vooral dankzij de inzet van Gerwin. De secretaresses, Marlous, Nicole en Thelma, ook bedankt voor de ondersteuning, vooral bij het regelen van de conferentiereizen. Ook Jochem verdient een bedankje voor de template voor dit boekje. Deze is verder ontwikkeld en beschikbaar gemaakt op Github voor de geïnteresseerden¹.

Wat misschien nog wel het belangrijkste is aan een promotie, is alles er omheen. Je hoofd leeg maken, wat bij mij goed ging door te gaan sporten. Dankzij de vele Messed Uppers heb ik de problemen van de promotie aan de kant gezet bij trainingen, wedstrijden, toernooien en feestjes. Net als bij een promotie zijn hier zo de up en downs (hebben we dit seizoen al een wedstrijden gewonnen in de Eredivisie?). De vakantie en etentjes met de vrienden uit Doetinchem, EJ, Hans, Martin, Niels en Tjerk, zorgden voor de nodige afleiding. Wat is er beter dan inspiratie op doen met jullie op een strand in Bali, of tussen de vogels in een mangrovebos in Senegal? Ook de nieuwe collega's bij DEMCON/Bond zorgen voor een leuke werksfeer zodat het ook niet zo erg is om 's avond de laatste loodjes van mijn proefschrift kan afronden.

Als laatste wil ik mijn familie nog bedanken. Pap, en mam, ook al snappen jullie totaal niet waar ik me de afgelopen 4 jaar mee bezig heb gehouden, toch waren jullie altijd geïnteresseerd in mijn onbegrijpbare verhalen. Luc, jij stond altijd klaar bij Gringo's om mij aan de drank te helpen, als ik dat weer eens nodig had. Donna, jij hebt vast wat van je overvloedige energie aan mij overgedragen om mijn promotie af te kunnen ronden. Nienke, zonder jou was ik vast niet zo ver gekomen met mijn promotie, had ik een saaie foto op de cover gehad, zaten er nog veeel meer spelfouten in dit boekje,

Guus

Enschede, januari 2019

¹<https://github.com/gkuiper/phdthesis-template>

CONTENTS

1	INTRODUCTION	1
1.1	Cyber-Physical Systems	2
1.2	Real-time systems	4
1.2.1	<i>Run-time scheduling</i>	6
1.2.2	<i>Analysis of run-time scheduling</i>	8
1.3	Problem statement	10
1.4	Contributions	11
1.5	Outline	12
2	BACKGROUND	15
2.1	Model checking	16
2.1.1	<i>Transition systems</i>	17
2.1.2	<i>Reachability analysis</i>	18
2.1.3	<i>Bisimulation</i>	19
2.1.4	<i>Timed automata</i>	21
2.2	Dataflow models	25
2.2.1	<i>Properties of dataflow models</i>	27
2.2.2	<i>More expressive dataflow models</i>	28
2.2.3	<i>Analysis</i>	30
2.3	Analysis models for concurrent systems	31
2.4	Summary	33
3	ENFORCING MUTUALLY EXCLUSIVE TASK EXECUTION IN MODAL APPLICATIONS	35
3.1	Related work	37
3.2	Basic idea	39
3.3	Types of mutual exclusivity	43
3.4	Response times TDMA	44
3.5	Real-time lock implementation	45
3.5.1	<i>Realization</i>	45
3.5.2	<i>Code Generation</i>	48

3.5.3	<i>Deadlock-freedom</i>	51
3.6	SVPDF model	52
3.7	Lock from sequential specification	53
3.8	Case study	55
3.9	Conclusion	58
4	COMPOSITIONAL ANALYSIS OF MODES AND FPP SCHEDULING	61
4.1	Related Work	63
4.2	Basic idea	64
4.3	Analysis flow	67
4.3.1	<i>Flattening of a hierarchical level</i>	69
4.4	Periodic source constraints	72
4.5	Response times	73
4.5.1	<i>Mutual exclusive execution using locks</i>	74
4.6	Response times larger than period	76
4.7	Case study	77
4.8	Conclusion	82
5	LATENCY ANALYSIS USING TIMED AUTOMATA	85
5.1	Related work	87
5.2	The HSDF ^a model	87
5.3	Max-plus Semantics of HSDF ^a	89
5.4	Extended timed automata	91
5.5	Timed automata model of HSDF ^a graphs	92
5.5.1	<i>Uppaal components</i>	92
5.5.2	<i>Dataflow edge model</i>	93
5.5.3	<i>Actor model</i>	95
5.5.4	<i>Complete automaton of an HSDF^a graph</i>	97
5.5.5	<i>Integer clock constraints</i>	98
5.6	Case study	98
5.7	Conclusion	101

6	HYBRID LATENCY ANALYSIS	105
6.1	Introduction	105
6.2	Related work	107
6.3	Basic idea	109
6.4	Timed automata	110
6.4.1	<i>FIFO buffer</i>	110
6.4.2	<i>Task template</i>	110
6.4.3	<i>Processor template</i>	112
6.4.4	<i>Verification</i>	113
6.5	Timed-dataflow	113
6.5.1	<i>Deadlock</i>	113
6.5.2	<i>Minimum guaranteed throughput</i>	114
6.5.3	<i>Approximative dataflow analysis</i>	114
6.6	Hybrid analysis	115
6.7	Sequence constraints	116
6.7.1	<i>Negative tokens</i>	119
6.7.2	<i>Redundant constraints</i>	119
6.8	Case study	121
6.9	Conclusion	125
7	CONCLUSION	127
7.1	Summary	128
7.2	Contributions	129
7.3	Recommendations for future work	131
	ACRONYMS	135
	SYMBOLS	137
	BIBLIOGRAPHY	139
	LIST OF PUBLICATIONS	147
	INDEX	149

INTRODUCTION

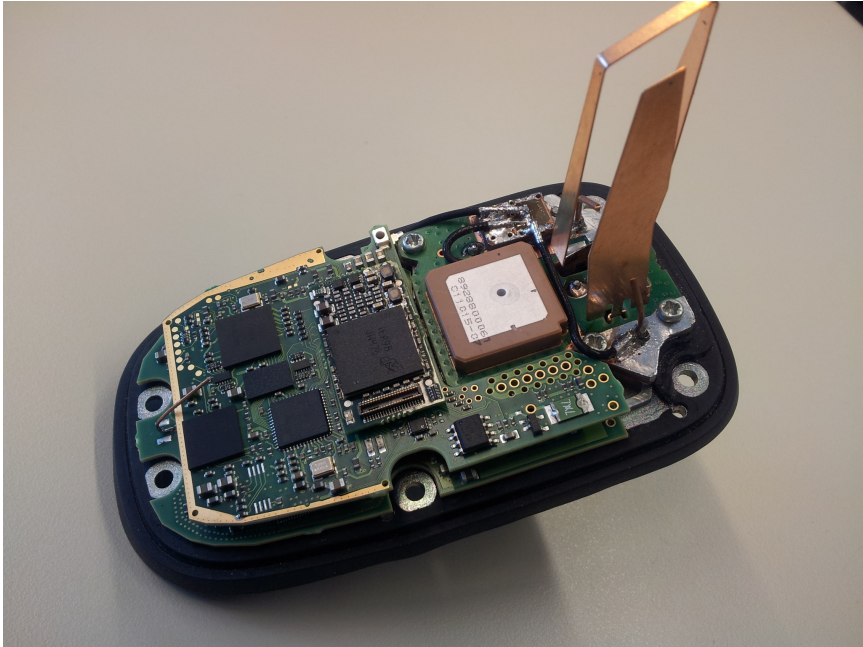
Cars have had no radical changes to their design for decades, until recently. Fossil fuels are running out quickly, which leads to the need to use renewable energy sources. As a result, electrical cars have started to emerge. The next major innovation in the automotive industry will likely be autonomous driving.


The trend to increase the level of autonomy of cars imposes great challenges to their design. There is a very complicated interaction between the physical environment, sensors observing the environment, the hardware and software processing these observations and taking actions, and the actuators of the car. The interaction between these components, in for example autonomous cars, results in a system that is known as a Cyber-Physical System (CPS).

An increasing amount of data is processed by cars in order to provide a certain level of autonomy. Sensors in cars produce this data by observing the environment. However, more intelligent autonomous cars also take advantage of communication with nearby vehicles. These cars are equipped with vehicle-to-vehicle communication systems, like the one shown in Figure 1.1. In this thesis we will focus on the modem subsystem of these vehicle-to-vehicle communication systems.

There are very strict demands on the temporal behavior of the modem system in autonomous cars, since these cars are safety-critical systems. Incorrect or too late decisions can have disastrous consequences. Using the vehicle-to-vehicle communication system, a car must respond quickly to a sudden braking action of a closely preceding car, to prevent collisions. During the design of such a modem subsystem, we want to provide guarantees about the worst-case temporal behavior of that subsystem. In this thesis we will therefore describe new models and analysis techniques that can be used to provide such temporal guarantees.

The modem application is computationally intensive, since it has to deal with rapidly changing channel conditions. Therefore, the modem is usually implemented on a



 Figure 1.1: A shark fin shaped modem system of WISI Automotive for vehicle-to-vehicle communication (Photo by Business Wire).

multiprocessor system. In this thesis we will analyze applications that are implemented on such multiprocessor systems.

The outline of this chapter is as follows. CPSs are described in more detail in Section 1.1. We then expand upon the digital processing part of these CPSs that must satisfy temporal constraints in Section 1.2. There, we also compare existing analysis approaches for these type of systems and their shortcomings. The main problem addressed in this thesis is formulated in Section 1.3. The contributions of this thesis towards improving the analysis methods are summarized in Section 1.4. Finally, the outline of this thesis is presented in Section 1.5.

1.1 CYBER-PHYSICAL SYSTEMS

The applications we are targeting in this thesis receive their incoming data from the environment, process it, and transmit the corresponding output back to the environment. The data is often processed on an *embedded system*, which is a computer system designed for a specific application. The interaction between the physical environment and the computational part in the form of an embedded system forms a CPS.

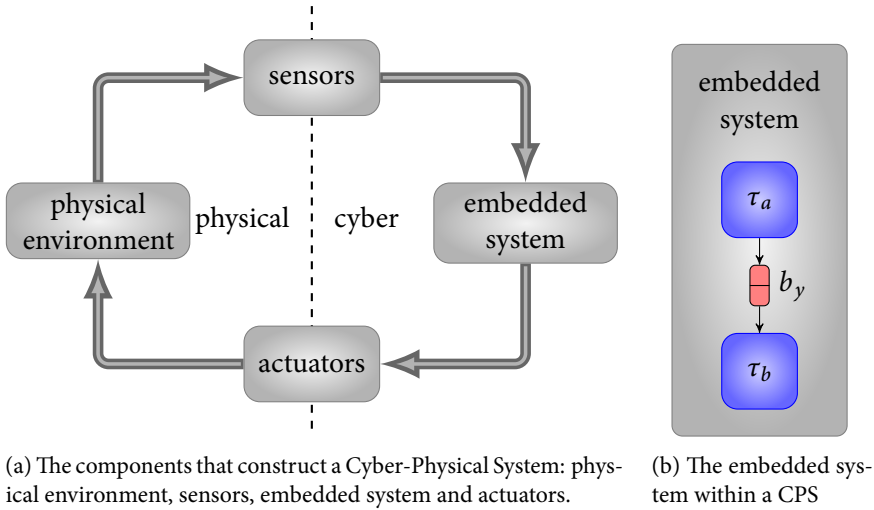


Figure 1.2: An overview of the components of a Cyber-Physical System (CPS) (a) and the embedded system inside a CPS (b).

A Cyber-Physical System is defined as:

A computing system that is tightly coupled with its physical environment, where algorithms running on the computing system interact with the physical environment, based on the inputs received from sensors measuring properties of the physical environment.

Such a CPS encompasses both the *continuous-time* domain of the physical environment, and a *discrete-time* domain in the embedded system. On the boundaries of these two domains, a conversion is made from continuous-time signals to discrete-time signals and vice versa. A *sensor* performs the task of converting a continuous time signal to a discrete signal. The continuous-time signals are sampled by a sensor *periodically*: a sample is taken from the continuous-time signals at equal intervals of time. An *actuator* receives a discrete-time signal and transforms it into a continuous-time signal. Hence, sensors and actuators perform the opposite conversion. Figure 1.2a shows how these components together form a CPS.

In this thesis we focus especially on modern applications for vehicle-to-vehicle communication. The wireless communication channel is the physical environment in this type of CPSs. The modem consists of both a receiver and transmitter, where the receiver decodes samples produced by an Analog-to-Digital Converter (ADC) and the transmitter encodes samples, which are sent to a Digital-to-Analog Converter (DAC).

We consider embedded systems that execute *stream processing applications*. This are applications that process endless streams of samples. Internally, a stream processing application can be subdivided into *tasks*, as shown in Figure 1.2b. Each task

performs a part of the stream processing application. The output of a task, which thus forms an intermediate result of the application, is typically communicated to other tasks using First-In-First-Out (FIFO) *buffers*. One task writes data into the buffer, whereas another task reads data from it. In case there is insufficient data in the buffer, the task that reads is blocked; i.e. it has to wait until data arrives before it can continue to execute. Moreover, these FIFO buffers have a finite size, such that a task that wants to write data into a buffer is blocked in case the buffer is already full. These finite-sized buffers therefore create cyclic dependencies in applications. The advantage of using FIFO buffers is that they allow for functionally deterministic exchange of data.

The applications we consider are either *static* or *dynamic*. In the static case, the rate at which tasks in the application execute is determined by the sample rate of the sensors. However, in the dynamic case, tasks can be *executed conditionally*. Tasks are said to execute conditionally if the values of the incoming data determine whether the tasks will actually execute or not. A task that does not execute, also does not produce data into its output buffers. The behavior of dynamic applications can therefore change depending on the physical environment. These dynamic applications are also called *modal applications*, because the execution of tasks can depend on previously received data, which puts the application in a particular *mode*. In this thesis we will consider both modal applications and static applications.

Although the use of FIFO buffers in applications itself can result in functionally deterministic behavior, this does not hold for the buffers at the boundaries of the application. Buffers that are filled by sensors, or read by actuators behave differently. These sensors and actuators execute time-triggered, instead of tasks that execute data-driven. These two execution schemes, time-triggered and data-driven execution, are discussed in more detail in the next section. If the throughput constraint that is imposed by these sensors and actuators is not fulfilled, buffer overflows or underflows can occur. A buffer overflow results in loss of samples. In case of a buffer underflow, the last sample is repeated. These buffer overflows and underflows lead to undesirable functionally non-deterministic, and thus unpredictable, behavior. The embedded system is therefore subjected to throughput constraints, which makes this system a real-time system. Real-time systems are discussed in more detail in the next section.

1.2 REAL-TIME SYSTEMS

The type of embedded systems we address in this thesis must satisfy strict temporal constraints. These systems are therefore real-time systems.

A real-time system is defined as:

A system of which the correctness of their results not only depends on the values as an outcome of a computation, but also on the moment in time at which these values are produced.

In such a real-time system, tasks are characterized by the time it takes to perform one execution, which is called the *execution time* of a task. An execution of a task may start when there is sufficient data for a task to read from all its input buffers. After a task finishes its execution, data is written to its output buffers. The execution time of a task is typically not a constant. Therefore, an upper bound on the execution time is specified. This upper bound is called the Worst-Case Execution Time (WCET). The Best-Case Execution Time (BCET) is a lower bound on the execution time of a task.

In this thesis we consider embedded systems with multiple processors such that sufficient computational performance is available to meet temporal requirements. Tasks can either execute on different processors *in parallel*, or some tasks can execute interleaved in time on the same processor *concurrently*. A *scheduler* determines the moment in time at which a task is executed on a processor. We consider systems where tasks are scheduled on a per-processor-basis, therefore there is a scheduler for each processor. The assignment of tasks to processors is performed at design-time, before the application is executed on a multiprocessor system.

Two task scheduling policies exist: time-triggered and data-driven. In time-triggered scheduling policies, the start of an execution of a task is performed based on a timer. For data-driven schedulers, scheduling is performed based on the arrival of data. In the latter case, the application is less sensitive to variations in the execution times. A sporadic longer execution of a task can in some cases be compensated by shorter consecutive executions to still meet the temporal constraints on the execution of the application. Such an aperiodic, data-driven, scheduling policy can exploit a tighter workload characterization of tasks than the WCET characterization [Hau15]. In this thesis we develop techniques that exploit these tighter workload characterizations.

Many real-time applications contain *cyclic dependencies*. These cyclic dependencies can be a result of feedback loops and the use of bounded FIFO buffers for inter-task communication. These cyclic dependencies can bound the number of concurrent executions of tasks on the cycle. Although these cyclic dependencies can result in better analysis results, they complicate the analysis of real-time systems.

Analysis of a real-time system is required in order to verify whether the system satisfies its temporal constraints. Therefore, we need to analyze systems with the combination of tasks with: cyclic dependencies, variable execution times, parallel execution of tasks, and the effect of run-time scheduling. A temporal constraint of the system can be in the form of a maximum *latency*. The *latency* is the time difference between the time at which a sample was produced by a sensor and the corresponding sample was consumed by an actuator.

Sensors often produce samples periodically, and therefore impose a *throughput* constraint on the real-time system. This means that the throughput of each individual task, as well as the combined throughput of all communicating tasks, should be verified for satisfying the throughput constraint. Analysis models for real-time

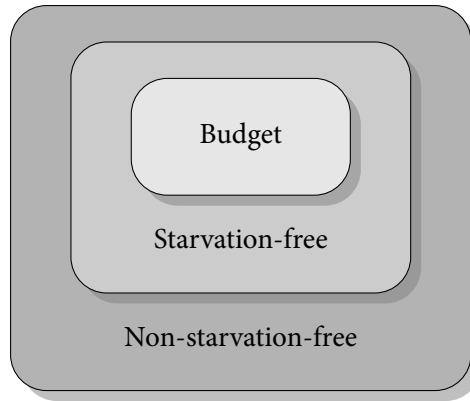


Figure 1.3: Classes of run-time schedulers.

systems that can express the combination of tasks with cyclic dependencies, variable execution times, and that include the effect of scheduling, are discussed in Chapter 2. In that chapter, we also address the analysis of these models.

1.2.1 RUN-TIME SCHEDULING

A *scheduler* performs arbitration for a *shared resource*, a processor in this case, using a certain scheduling policy. In this section, we introduce three classes of schedulers for real-time systems. After this introduction, we compare existing analysis approaches for these systems and the techniques behind these approaches in Subsection 1.2.2.

A task that is ready to start its execution on a processor can be delayed by the scheduler if other tasks also want to execute on the same processor. This delay caused by other tasks is called *interference*. The *response time* of a task is the time between the moment a task is ready to start an execution and the finish time of that execution. The response time thus consists of both the execution time of the task and the interference caused by other tasks.

In [WBS09], three classes of run-time schedulers for real-time systems are defined:

1. Budget schedulers
2. Starvation-free schedulers
3. Non-starvation-free schedulers

The relation between these three classes of schedulers is illustrated in Figure 1.3. Budget schedulers are the least general class of schedulers, whereas non-starvation-free schedulers are the most general class in Figure 1.3.

These classes of schedulers can be distinguished based on the amount of knowledge that is required about the tasks that are being scheduled, in order to bound the

interference between tasks. The three classes can be distinguished based on two properties:

1. Execution times of all tasks sharing the same resource
2. Rate of execution of all tasks sharing the same resource

The execution time of a task is the time the task needs exclusive access to the processor to finish an execution. The execution rate of a task is the number of executions of a task during a certain time window.

Budget schedulers are the most restrictive class of schedulers we consider. The advantage of schedulers in this class is that no knowledge about execution times and execution rates of other tasks is required to provide a bound on the interference between tasks. An example of a budget scheduler is the Time Division Multiple Access (TDMA) scheduler. A budget scheduler assigns a minimum time budget to each task within a maximum time interval, the *replenishment interval*. During each interval, every task can execute for the duration of their assigned time budget. The sum of the budgets of all tasks logically adds up to the entire replenishment interval. The maximum interference a task can experience can therefore be derived independently for every task, i.e. without knowledge of the execution time and execution rate of other tasks.

The starvation-free schedulers belong to a larger class of schedulers. For these schedulers, only knowledge about the execution time of other tasks is required to upper bound the interference. The Round-Robin (RR) scheduler is the best known scheduler of this class. For a RR scheduler, all tasks may execute once, in a predefined order, until they are finished, after which this sequence repeats. For this reason, all other tasks can execute at most once, before the task under consideration is executed. Therefore, knowledge of only the execution times is sufficient to determine the maximum interference.

The broadest class of schedulers is the class of non-starvation-free schedulers. Both the execution time and execution rate of other tasks are required to determine the interference between tasks. A well known scheduler inside this class is the Fixed Priority Pre-emptive (FPP) scheduler, which arbitrates tasks based on their priority. A task with a higher priority that becomes ready to execute, pre-empts a lower priority task, to quickly get access to the processor. The task with the higher priority causes interference on the lower priority tasks that are thereby delayed. Without knowledge about both the execution time and execution rate of all tasks with a higher priority, the worst-case situation is that a low priority task must wait indefinitely before its execution can start.

Out of these three classes of schedulers, systems with budget schedulers and starvation-free schedulers are relatively easy to analyze. However, in practice many real-time systems make use of non-starvation-free schedulers, although the use of these schedulers draatically complicates system analysis. System designers are often unaware of the consequence on the analysis of using non-starvation-free schedulers. In this thesis we focus on non-starvation-free schedulers.

Table 1.1: Comparison of analysis methods for systems with non-starvation-free schedulers.

Analysis method		Multiprocessor	Cyclic resource dependencies	Cyclic data dependencies	Buffer sizing	Dynamic applications	Run-time	Accuracy
MAST	[HGGM01]	+	+	-	-	-	+	-
SymTA/S	[HHJ ⁺ 05]	+	+	-	-	-	+	-
MPA-RTC	[TS09]	+	+	-	-	-	+	-
MPA-RTC	[JPTY08]	+	-	+	-	-	+	-
Dataflow	[KHB16a]	+	+	+	+	-	++	+-
Timed automata	[HV06]	+	+	-	-	-	--	+
TIMES	[AFM ⁺ 03]	-	+	+	-	-	--	+
Our approach	this thesis	+	+	+	+	+-	-	++

1.2.2 ANALYSIS OF RUN-TIME SCHEDULING

There are two main categories of analysis techniques for systems with non-starvation-free schedulers. The first computes a fix-point, and the second uses model checking, which is based on reachability analysis. In this section, we briefly introduce both categories of analysis techniques, and give a comparison of analysis approaches that utilize these techniques.

Many analysis approaches have been developed for the analysis of systems with non-starvation-free schedulers. Table 1.1 presents a number of these approaches and highlights system features that are supported (+) or not supported (-). The table also presents a rough estimate of their run-time and accuracy within the range of ‘very good’ (++) to ‘very bad’ (--). The run-time and accuracy of these approaches is heavily dependent on the system being analyzed, as is shown using a benchmarks for a number of approaches in [PWT⁺07].

Most of the fixed-point based analysis approaches are not able to analyze arbitrary task graphs. Especially cyclic data dependencies are problematic for MAST [HGGM01], SymTA/S [HHJ⁺05], and MPA-RTC [TS09]. MPA-RTC has been extended to support either cyclic data dependencies [TS09], or cyclic resource dependencies [JPTY08], but the combination of both is not yet supported.

Dataflow based analysis is suitable to analyze graphs that contain cyclic dependencies [Das04, SB09]. These dependencies can even be exploited to obtain tighter analysis results using buffer sizing techniques [WGHB15]. Schedulers cannot be modeled directly in dataflow models unless the schedule is static [DSB⁺13]. However, the

effect of scheduling can be included in dataflow models. This was first shown for the class of budget schedulers [WBS09, LMC12]. Recently, the class of supported schedulers has been extended with non-starvation-free schedulers [HWGB13, HGWB14, LMB⁺14]. The dataflow analysis approaches also compute fixed-points. We focus on dataflow analysis in this thesis, since it supports the most system features as shown in Table 1.1. Moreover, cyclic data dependencies analysis and buffer sizing techniques have been developed that improve the accuracy of the analysis results.

Timed automata based approaches [HV06, AFM⁺03] are based on model checking, which uses reachability analysis. TIMES [AFM⁺03] is an analysis tool that internally uses timed automata for the analysis of systems with non-starvation-free schedulers. However, it currently does not support systems with multiple processors. These timed automata based approaches produce accurate analysis results, but in general suffer from a large state space and therefore their run-time can be impractically large.

Before we state the differences between the two categories of analysis approaches, we need to explain fixed-point based analysis in more detail. A *fixed-point* is a value x of a function f for which $f(x) = x$. Loosely speaking, the least fixed-point of a function is the smallest value x for function f that is a fixed-point. Kleene's fixed-point theorem states that the unique least fixed-point of a Scott-continuous function can be computed by iteratively calling the function, starting at the least possible value. A function is Scott-continuous iff it preserves directed suprema. A Scott-continuous function is by definition also a *monotonic* function.

Functions can either be monotonic or non-monotonic. A function f is *monotonic* iff it is either entirely non-increasing, or entirely non-decreasing. As a results, there are two possible types of monotonic functions: for monotonically increasing functions, we have that $\forall_{i \geq j}: f(i) \geq f(j)$, and for monotonically decreasing functions, we have that $\forall_{i \geq j}: f(i) \leq f(j)$. A function that does not satisfy any of these two conditions is non-monotonic. However, an approximation of such a non-monotonic function can be a monotonic function.

For the analysis of systems that include schedulers, we are interested in the fixed-point of the function that calculates the interference a task can experience. Given a monotonic function for the maximum interference, the maximum interference is obtained by iterating the function starting with zero interference [TBW94]. This iterative computation of a fixed-point is often relatively fast. However, in general, the interference function is non-monotonic, and a monotonic over-approximation of the function needs to be constructed, before a fixed-point can be calculated.

Fixed-point based analysis approaches rely on abstracting from scheduling in order to create a monotonic model. Also non-determinism in an application, for example in the execution times of tasks, can be hidden using abstractions. This creates a *bounding abstraction* where only a worst-case and best-case bound of the system are analyzed [KB17]. These two bounds must include all possible behaviors of the system, since a monotonic over-approximation of the system is analyzed. As a results of these bounding abstractions, the analysis is computationally efficient.

However, this efficiency of the analysis approaches based on the computation of fixed-points, comes with a potential loss of accuracy.

On the other hand, timed automata [AD94] do allow analysis of non-monotonic behavior and the expression of non-determinism. As a result, non-starvation-free schedulers can be modeled in timed automata. More accurate analysis can therefore be performed by model checking of these timed automata. The exact analysis of systems with the combination of pre-emptive schedulers, dependencies between tasks and non-constant execution times, as described in more detail in Chapter 6, is an undecidable analysis problem. Therefore, timed automata based analysis approaches also need to perform approximations to prevent undecidability. Since the analysis is performed on a system with non-monotonic components, it is insufficient to only analyze the worst-case and best-case behavior of each component. Therefore, all behaviors of the system must be included in the inclusion abstraction of the system [KB17]. This can lead to a traversal of a very large state space. While model checking of timed automata can lead to more accurate analysis results than analysis approaches based on the iterative computation of a fixed-point, the large state space causes it to be computationally intensive.

The comparison presented in this section shows that the holy grail of analysis approaches, for systems with non-starvation-free schedulers, does not exist. Current analysis approaches are not able to accurately analyze applications that contain arbitrary cyclic dependencies. Moreover, dynamic applications are also not considered by these approaches. Both fixed-point based approaches as well as reachability analysis using model checking have their advantages and disadvantages. A purely timed automata based analysis approach can analyze systems with non-monotonic behavior and non-determinism with a high accuracy, at the cost of a long run-time. The abstractions required to efficiently compute fixed-points iteratively, lead to a low accuracy of the fixed-point based approaches. This low accuracy, however, can be improved by exploiting cyclic data dependencies using dataflow analysis techniques. In this thesis we address these issues by combining dataflow based analysis with model checking of timed automata.

1.3 PROBLEM STATEMENT

So far, we have introduced the type of systems that we address in this thesis: CPSs, of which we are interested in the real-time embedded system part. The implementation of the real-time system is targeted towards multiprocessor systems, where tasks are scheduled at run-time. Many analysis approaches exist for this type of systems, however, each approach has its downsides such as the restriction on the class of supported system features, a low accuracy or a very high run-time.

Therefore, the research problem that is addressed in this thesis is to:

Define techniques that improve the accuracy of the real-time analysis results, and also increase the class of real-time multiprocessor systems and applica-

tions that can be analyzed, while minimizing the run-time of the analysis algorithms.

Besides improvements of the analysis techniques, we also consider improvements to applications by adding locks and barriers that introduce so called *sequence constraints*. These sequence constraints limit the possible orders in which tasks can execute. As a result of these sequence constraints, the analysis can be simplified and the throughput, latency and accuracy can be improved.

This thesis describes dataflow analysis techniques for dynamic applications executed on multiprocessor systems with non-starvation-free schedulers. Furthermore, we show that a subclass of dataflow models can be modeled as timed automata. From the analysis results, it can be concluded that by applying model checking techniques, more accurate results can be obtained at the cost of a longer run-time. However, it also became apparent that it is impossible to guarantee exactness of the results for the considered analysis problem. Moreover, we will present an analysis technique that combines dataflow analysis and model checking that reduces the run-time without sacrificing accuracy. The work described in this thesis will also help to better understand the relation, similarities, and differences between model checking and dataflow based temporal analysis approaches.

1.4 CONTRIBUTIONS

The contributions presented in this thesis are:

- » Techniques to improve the throughput of modal applications on multiprocessor systems with budget scheduler by introducing a lock.
- » Techniques to enable the analysis of dynamic applications on multiprocessor systems with non-starvation free schedulers, by introducing a barrier.
- » Techniques to improve the analysis accuracy of static applications on systems without run-time schedulers by introducing a transformation of dataflow into timed-automata model.
- » Techniques to improve the analysis accuracy of static applications on systems with run-time schedulers by introducing a model of the run-time scheduler in timed automata. To achieve this, we encountered the issue that over-approximation must be applied to create a timed automata model. We observed long run-times especially when model checking methods were applied for buffer sizing.
- » Techniques to reduce the run-time of buffer sizing using model checking by proposing a combination of model checking with dataflow analysis to prune the search space.
- » Techniques to reduce the latency by proposing techniques to insert additional sequence constraints.



Summarizing; in this thesis we introduce real-time analysis techniques for static as well as dynamic applications. The analysis makes use of dataflow models and timed automata, and combines them to reduce the run-time of the analysis algorithms. Furthermore, additional sequence constraints are derived that improve the analysis results.

1.5 OUTLINE

The analysis models used in this thesis, dataflow and timed automata, are introduced in Chapter 2, together with our view on the relation between these models. Chapter 3 introduces a *lock* to allow efficient analysis and resource usage when budget schedulers are used. Chapter 4 extends the analysis to the more general class of non-starvation-free schedulers by introducing *barriers*. The combination of locks and barriers allows hierarchical modes to be analyzed in isolation.

The subsequent part of this thesis addresses temporal analysis where we only consider a single mode. In Chapter 5, we improve the accuracy of the analysis of applications modeled as dataflow graphs by transforming the graphs into timed automata, which are model checked. Chapter 6 introduces a *hybrid analysis* flow, which combines computationally efficient dataflow analysis with model checking of timed automata. Additional *sequence constraints* are introduced in applications in order to reduce the latency.

Finally, Chapter 7 presents the conclusions of this thesis and gives recommendations for future work.

BACKGROUND

Concurrent real-time applications running on multiprocessor systems are difficult to analyze. In 1965, Dijkstra identified one reason which is mutual exclusion [Dij65]. A classical example which illustrates issues caused by mutual exclusion, also by Dijkstra, is the dining philosophers problem. All philosophers sitting around a table should be able to alternate infinitely between thinking and eating, while sharing resources.

In order to analyze this interleaving of concurrent processes that together form an entire system, the system could be modeled in an analysis model. Analysis on such a model is used to analyze functional behavior such as deadlock, starvation and determinism. Next to the functional correctness, temporal behavior is crucial for real-time systems, and should also be verified using these analysis models.

Since we are interested in functionally correct systems, we restrict the systems we want to model to concurrent processes that communicate using FIFO buffers [Dij72]. This is opposed to systems that communicate using shared memory and requiring sort of locking, to guarantee mutual exclusive access to data structures, which prevents race conditions. Processes do not have any side-effects, except the interaction with buffers, and can only update their own state.

Throughout this thesis we use two timed analysis models: dataflow models and timed automata. The expressiveness of dataflow models is restricted, however, dataflow models can be analyzed efficiently. On the other hand, timed automata are very expressive, and many properties can be derived from them using model-checking. Since model-checking is based on state space exploration, and the state space is often large, the analysis of timed automata is less efficient and costs a large run-time.

In dataflow models, concurrent processes are specified as actors. Dependencies between these processes are represented by edges connecting actors. Together, these actors and edges form a directed graph. Cyclic dependencies in such a graph can

be used to bound the number of concurrent executions of actors on the cycle.

Each timed automaton executes sequential, however, multiple timed automata can execute concurrently. These timed automata communicate using so called hand-shaking actions. Communicating timed automata execute in parallel. A supervising scheduler can be modeled using timed automata, which grants access to a shared resource in a concurrent multiprocessor real-time system.

The two analysis models used in this thesis are discussed in more detail in this chapter: model checking of timed automata in Section 2.1 and dataflow models in Section 2.2. We use dataflow models throughout this entire thesis, whereas model checking of timed automata is relevant for Chapter 5 and Chapter 6. Furthermore, in Section 2.3 we compare the expressiveness of these two analysis models and place them in perspective to other related models.

2.1 MODEL CHECKING

Formal verification of safety-critical systems using *model checking* [CE81] is becoming increasingly popular. The verification process using model checking is fully automated by tools like UPPAAL [DILSo9], in contrast to providing proofs manually. Model checking usually requires the following steps: creation of a model of a system, simulation of the model, and finally formal verification of this model. A convenient representation of the model is in the form of graphs, as is the case for timed automata. Simulations of such a model can be used to compare the behavior of the model with the behavior of the system. If these differ, a revision of the model is required. The formal verification using the model consists of analyzing the state space of the model for certain properties specified in a temporal logic like Computation Tree Logic (CTL) [CE81]. When a property does not hold, a counterexample can be returned by the model checker [CGMZ95].

To understand model checking for real-time systems, we first consider the mathematical basis for model checking: transition systems. Then we discuss how to analyze these transition systems using reachability analysis. However, reachability analysis is not guaranteed to terminate for transition systems that include continuous time. Bisimulation is an equivalence relation between two transition systems; it can be used to prove that an abstraction with a finite state space exists of a system with an infinite size. The finite state space is the result of partitioning the continuous time of the state space into a finite number of regions, which together form a regions graph. Finally, timed automata are introduced based on these concepts. For timed automata a finite region graph can always be constructed, such that reachability analysis is guaranteed to terminate. We use timed automata later on in this thesis to analyze concurrent real-time multiprocessor systems.

2.1.1 TRANSITION SYSTEMS

Computations of discrete systems can often be described as states with transitions between them. Such a *transition system* can formally be described as a directed graph, $T = (S, Act, \rightarrow, S_0)$ where:

- » S is a set of states
- » Act is a set of actions
- » $\rightarrow \subseteq S \times Act \times S$ is a transition relation
- » $S_0 \subseteq S$ is a set of initial states

In the directed graph, the nodes are represented by the states and the edges are the transitions between different states.

A transition system starts in one of its initial states $s \in S_0$. In case there are multiple initial states, an initial state is selected non-deterministically from S_0 . From there, the state of the system evolves to subsequent states according to the possible transitions \rightarrow in the system. A current state s transitions to a next state s' following the transition $s \xrightarrow{\alpha} s'$. During the transition the action $\alpha \in Act$ is performed. Again, when there are multiple transitions possible from s , one of them is selected nondeterministically. This nondeterminism is crucial to describe concurrent systems [BKLo8].

The set S and Act of a transition system is not required to be finite. However, for the transition system T to be finite, the size of S and Act must be finite.

For a transition system we can define the *direct successors* and *predecessors* of a state $s \in S$ as $Post(s)$ and $Pre(s)$.

$$Post(s, \alpha) = \{s' \in S \mid s \xrightarrow{\alpha} s'\} \quad (2.1)$$

$$Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha) \quad (2.2)$$

$$Pre(s, \alpha) = \{s' \in S \mid s' \xrightarrow{\alpha} s\} \quad (2.3)$$

$$Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha) \quad (2.4)$$

This can be extended to sets of states $C \subseteq S$.

$$Post(C) = \bigcup_{s \in C} Post(s) \quad (2.5)$$

$$Pre(C) = \bigcup_{s \in C} Pre(s) \quad (2.6)$$

Using these definitions, a *terminal state* is defined as $Post(s) = \emptyset$. The set of terminal state S_f it holds that $Post(S_f) = \emptyset$. A transition system is deterministic if it holds $\forall s, \alpha : |Post(s, \alpha)| \leq 1$ and $|S_f| \leq 1$.

Algorithm 1: Forward reachability

```

1 Input:  $S, S_0, S_f$ 
2 Initialization:  $W_0 = S_0, i = 0;$ 
3 repeat
4   if  $W_i \cap S_f \neq \emptyset$  then
5     return  $S_f$  reachable
6    $W_{i+1} = Post(W_i) \cup W_i;$ 
7    $i = i + 1;$ 
8 until  $W_i = W_{i-1};$ 
9 return  $S_f$  not reachable

```

2.1.2 REACHABILITY ANALYSIS

Properties of a transition system can be verified by *reachability analysis*. An error condition of a system can be encoded as a certain state in a transition system. Analysis on the transition system is used to conclude whether that state is reachable, and thereby conclude that the error can occur in the system. More sophisticated system properties may be specified in logic such as CTL, however, this is outside the scope of this thesis, but is explained in [BKLo8]. We now focus on two simple reachability algorithms.

We discuss two reachability analysis algorithms that can be used to determine if a terminal state in S_f can be reached. One is the forward reachability algorithm whereas the other is the backward reachability algorithm. Both algorithms guarantee termination given finite transition systems. These reachability algorithms are modifications of standard graph traversal algorithms like Breadth-First Search (BFS) or Depth-First Search (DFS). These algorithms can further be extended to return a counter-example, showing the path to the terminal state in case it is reachable.

The *forward reachability* algorithm, see Algorithm 1, starts with the set of initial states S_0 which is copied in the set W_0 . This set is gradually extended during each iteration i of the algorithm with the successor states of that current set and stored in W_i using $Post(W_i)$, as defined in Equation 2.5. The algorithm terminates when no more states are added to W_i and it then returns that the terminal states cannot be reached. If however, W_i contains one of the terminal states the algorithm terminates and returns that the terminal states can be reached.

The *backward reachability* algorithm in Algorithm 2 is very similar. Since it operates backwards, the set W_0 is initialized with the set of terminal states S_f . This algorithm terminates when no more states are added to W_i using $Pre(W_i)$, as defined in Equation 2.6. A terminal state is reachable when there is a backwards path reaching one of the initial states in S_0 .

A potential issue of reachability analysis is that termination can only be guaran-

Algorithm 2: Backwards reachability

```

1 Input:  $S, S_0, S_f$ 
2 Initialization:  $W_0 = S_f, i = 0;$ 
3 repeat
4   if  $W_i \cap S_0 \neq \emptyset$  then
5      $\lfloor$  return  $S_f$  reachable
6      $W_{i+1} = Pre(W_i) \cup W_i;$ 
7      $i = i + 1;$ 
8 until  $W_i = W_{i-1};$ 
9 return  $S_f$  not reachable

```

teed in practice for finite transition system. However, in real-time systems, the number of states is infinite, because continuous time plays a crucial role. Encoding timestamps that are elements of \mathbb{R} for the events in the systems into states leads to an infinite number of states. Therefore, reachability analysis is in general not guaranteed to terminate.

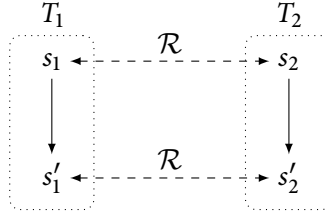

2.1.3 BISIMULATION

The problem of an infinite state space can in some cases be solved by grouping similar states together. A finite state space is obtained if an originally infinite state space can be grouped into a finite number of partitions. A *bisimulation* is a specific similarity relation between two transition systems [Mil80, Par81]. Two bisimilar systems can have the same model checking properties such as reachability. The goal is to construct a partitioning of a infinite state space into a bisimilar system that has a finite state space.

Formally, a bisimulation is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ between two transition systems $T_1 = (S_1, Act_1, \rightarrow, S_{0_1})$ and $T_2 = (S_2, Act_2, \rightarrow, S_{0_2})$ where:

- (1.) for each $s_1 \in S_{0_1}$, there exists $s_2 \in S_{0_2}$ such that $(s_1, s_2) \in \mathcal{R}$, and for each $s_2 \in S_{0_2}$ there exists $s_1 \in S_{0_1}$ such that $(s_1, s_2) \in \mathcal{R}$
- (2.) for each pair $(s_1, s_2) \in \mathcal{R}$
 - (2a.) if $s'_1 \in Post(s_1)$ then there exists $s'_2 \in Post(s_2)$ with $(s'_1, s'_2) \in \mathcal{R}$
 - (2b.) if $s'_2 \in Post(s_2)$ then there exists $s'_1 \in Post(s_1)$ with $(s'_1, s'_2) \in \mathcal{R}$

According to (1.) every initial state in S_{0_1} must be related to an initial state in S_{0_2} and the other way around. Moreover, the transitions in both systems must be related (2.). For a transition from $s_1 \in S_1$ to $s'_1 \in S_1$ there must also exist a matching transition from $s_2 \in S_2$ to $s'_2 \in S_2$ such that both s'_1 and s'_2 are related (2a.). Vice versa (2b.), a transition from $s_2 \in S_2$ must be matched by a transition from $s_1 \in S_1$. Figure 2.1 shows this relationship graphically. Multiple states in one transition system can be related to the same state in another transition system.


 Figure 2.1: Bisimulation relation for T_1 and T_2 .

A *quotient transition system* is a partitioning of a transition system T . Partitioning \mathcal{P} of S contains sets of states from S . Every state in S is included in exactly one partition in \mathcal{P} . The quotient transition system is defined as $Q = (\mathcal{P}, Act, \rightarrow, P_0)$ where:

- (1.) for all $P, P' \in \mathcal{P}$, $P \rightarrow P'$ iff $s \in P$ and $s' \in P'$ such that $s \rightarrow s'$
- (2.) $P_0 = \{P \in \mathcal{P} \mid P \cap S_0 \neq \emptyset\}$

A more strict condition is required for the partitions of Q to be a bisimulation of T , and thereby retain model checking properties like reachability.

$$\forall P, P' \in \mathcal{P}, \text{ either } P \cap Pre(P') = \emptyset \text{ or } P \cap Pre(P') = P \quad (2.7)$$

Equation 2.7 states that either none of the state in a partition $P \in \mathcal{P}$ can transition to a state in a partition $P' \in \mathcal{P}$, or all states in P can transition to P' . This particular partitioning groups bisimilar states together. A bisimilar quotient transition system can turn an originally infinite transition system into a finite transition system using the partitions where each partition can be seen as a state in the new transition system.

Figure 2.2a shows an example of a (finite) transition system of which we will derive a smaller bisimilar quotient transition system. The four states of T ; s_1, s_2, s_3 and s_4 , can be partitioned into $P_1 = \{s_1, s_2\}$ and $P_2 = \{s_3, s_4\}$ where $\mathcal{P} = \{P_1, P_2\}$. It can be verified that this \mathcal{P} satisfies Equation 2.7. According to this partitioning the transition system in Figure 2.2a is bisimilar to the transition system in Figure 2.2a, although it has only half as many states.

An algorithm that can create such a finite transition system is presented in Algorithm 3 [BKLo8]. First, it creates initial partitions for the initial states S_0 , terminal states S_f , and all remaining states. These partitions are refined until the algorithm terminates exactly when Equation 2.7 is satisfied. During each iteration a pair of sets of states is picked that do not satisfy the bisimulation condition Equation 2.7. These sets are replaced by a set that can transition to S_j , and another set that cannot transition to S_j . This process is repeated until all partitions satisfy the bisimulation condition. This algorithm is guaranteed to terminate for finite transition systems.

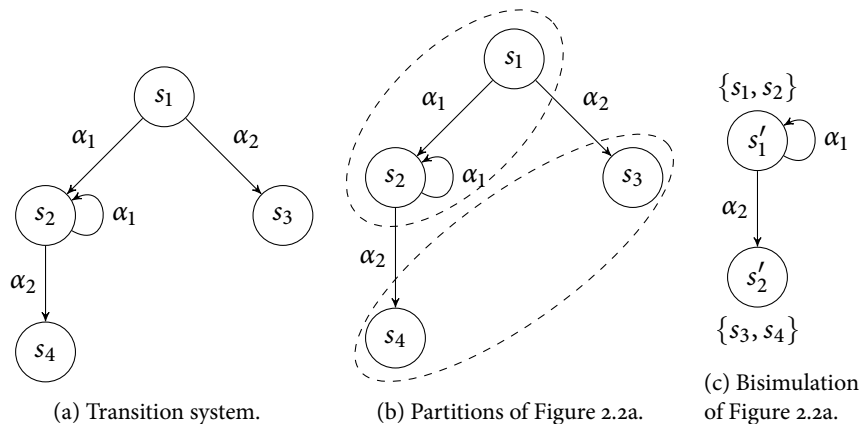


Figure 2.2: Example of a bisimulation.

Algorithm 3: Automatic derivation of a bisimulation of a finite transition system

```

1 Input:  $S, S_0, S_f$ 
2 Initialization:  $\mathcal{P} = \{S_0, S_f, S \setminus (S_0 \cup S_f)\}$ ;
3 while there exists
    $S_i, S_j \in \mathcal{P}$  such that  $S_i \cap \text{Pre}(S_j) \neq \emptyset$  and  $S_i \cap \text{Pre}(S_j) \neq S_i$  do
4    $S'_i = S_i \cap \text{Pre}(S_j)$ ;
5    $S''_i = S_i \setminus \text{Pre}(S_j)$ ;
6    $\mathcal{P} = (\mathcal{P} \setminus S_i) \cup \{S'_i, S''_i\}$ ;
7 return  $\mathcal{P}$ 

```

For infinite transition systems that include time, termination of the bisimulation algorithm is not guaranteed. Therefore, in the next section we look at timed automata, which are a subset of infinite transition systems for which a finite transition system does exist that is bisimilar to the original transition system.

2.1.4 TIMED AUTOMATA

For real-time systems, not only functional correctness of a system is of interest, but also the temporal behavior of such a system must be within a given specification. In this section we therefore discuss *timed automata* [Lew90, AD90, Dil89, AD94]. Timed automata can be seen as a compromise between expressiveness and analyzability, where the analysis of a timed automaton is always decidable due to restrictions regarding what can be described in the timed automaton. Inherently, including continuous time in a transition system leads to infinite transition systems. However, for timed automata, a finite transition system which is bisimilar to

the originally infinite transition systems can always be constructed.

The beneficial property of timed automata is that a finite transition system can always be constructed, which is a result of the way time can be expressed in a timed automaton. A timed automaton consists of a number of clocks, as real valued variables. All these clocks $x \in C$ progress time at the same rate $\dot{x} = 1$. The only other way the value of a clock can change is when a clock is reset to 0. Each clock can be reset independently. Clocks are therefore useful to measure the time difference between events. Furthermore, constraints on clocks g must follow a specific syntax:

$$g ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid g \wedge g \quad (2.8)$$

where $c \in \mathbb{N}$ and $x \in C$, $B(C)$ is the set of allowed clock constraints over C according to this syntax. It is crucial for timed automata that all comparisons of clocks are with constant integers. Constraints using rational constants could easily be supported by scaling and shifting these constraints to integers. Without the restrictions on the clock constraints that can be specified, timed automata would in general not be decidable.

A *timed automaton* \mathcal{A} is defined as $\mathcal{A} = (L, Act, C, \rightarrow, Inv, l^0)$ where

- » L is a finite set of locations
- » Act is a finite set of actions
- » C is a finite set of clocks
- » $\rightarrow \subseteq L \times Act \times B(C) \times 2^C \times L$ is a transition relation
- » $Inv : L \rightarrow B(C)$ is an invariant assignment function
- » $l^0 \subseteq L$ is a set of initial states

In this definition, 2^C is the power set of C , where the number of elements in this power set is $2^{|C|}$, where $|C|$ is the number of clocks in C . The transition relation consists of $l \xrightarrow{g:\alpha,D} l'$, where g is a binary clock guard, α an action in Act and $D \subseteq C$ a set of clocks that is reset to 0. A transition is therefore only possible when the clock constraint of the corresponding guard g is satisfied. A transition to another location is atomic and therefore does not advance clocks. A reset to 0 of a clock x only occurs when the clock $x \in D$. The invariant of a location $Inv(l)$ specifies a upper bound on the value of a clock in that location. This specific location must be left by a transition to another location within the specified bound.

From this definition of a timed automaton, a transition system that also supports time can be constructed. In such a *timed transition system*, the state is not only dependent on the location, but is a tuple (l, n) consisting of both a location $l \in L$ and value of all clocks n . The transition relation \rightarrow therefore allows two types of transitions. A transition in a timed transition system is either a discrete or delay transition according to:

- (1.) *discrete transition*: $(l, n) \xrightarrow{g:\alpha,D} (l', n)$ (including clock resets)

(2.) *delay transition*: $(l, n) \xrightarrow{d} (l, n + d)$ with delay $d \in \mathbb{R}$ ($n + d$ should satisfy $\text{Inv}(l)$)

where the discrete transition (1.) is similar to the one for timed automata. The delay transition (2.) should satisfy the invariant $\text{Inv}(l)$ of the current location l , since $n + d$ may not exceed the clock bound of an invariant.

The usage of the continuous time domain for the clocks immediately leads to an infinite state space. Moreover, the time delay d , can be infinitely small so there can be an infinite number of delay transitions within a bounded time interval. The transition system of timed automata appears to be infinite. Reachability algorithms for such a system are not guaranteed to terminate. However, based on the clock constraints in Equation 2.8, a limited number of clock regions can be constructed for a timed automaton. A *clock region* is a partitioning of the continuous time space for all clocks where the clock difference inside one region cannot be distinguished using clock constraints, as given in Equation 2.8.

As an example, consider a timed automaton consisting of two clocks x_1 and x_2 , a two locations l_0 and l_1 , and the invariants $\text{Inv}(l_0) = \{x \in \mathbb{R}^2 \mid x_1 \leq 3\}$ and $\text{Inv}(l_1) = \{x \in \mathbb{R}^2 \mid x_2 \leq 2\}$, as shown in Figure 2.3a. For each of the clocks the following clock regions can be identified:

$$x_1 : x_1 = 0, x_1 \in (0, 1), x_1 = 1, x_1 \in (1, 2), x_1 = 2, x_1 \in (2, 3), x_1 = 3, x_1 \in (3, \infty) \quad (2.9)$$

$$x_2 : x_2 = 0, x_2 \in (0, 1), x_2 = 1, x_2 \in (1, 2), x_2 = 2, x_2 \in (2, \infty) \quad (2.10)$$

A *region graph* is the entire set of all clock regions. The region graph for this system consists of the products of these sets of clock regions. The unbounded regions $x_1 \in (3, \infty)$ and $(2, \infty)$ might be reachable in case there is an invariant of a location that does not specify an upper bound for a particular clock. For this two-dimensional example, the region graph consists of a number of points, lines, triangle, and rectangles. These regions are for example:

$$\{x \in \mathbb{R}^2 \mid x_1 = 1 \wedge x_2 = 1\} \quad \text{point} \quad (2.11)$$

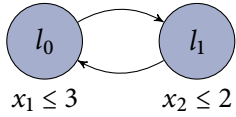
$$\{x \in \mathbb{R}^2 \mid x_1 \in (0, 1) \wedge x_2\} \quad \text{line} \quad (2.12)$$

$$\{x \in \mathbb{R}^2 \mid x_1 \in (0, 1) \wedge x_2 \in (0, 1) \wedge x_1 > x_2\} \quad \text{triangle} \quad (2.13)$$

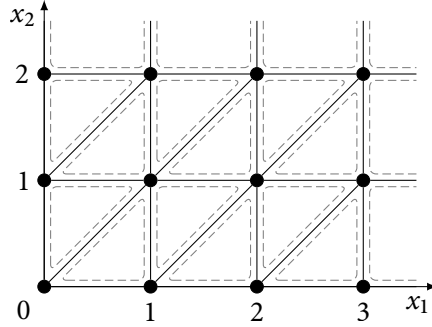
$$\{x \in \mathbb{R}^2 \mid x_1 \in (3, \infty) \wedge x_2 \in (0, 1)\} \quad \text{rectangle} \quad (2.14)$$

The region graph for the timed automaton shown in Figure 2.3a is graphically represented in Figure 2.3b.

Intuitively, there is a finite number of regions in such a region graph, and also a finite number of possible transitions between regions, which leads to a transition system that is finite. In fact, a region graph is proven to be a finite bisimulation of a timed automaton [Čer92]. Therefore, reachability analysis is decidable for timed automata.



(a) Timed automaton with two locations and invariants: $Inv(l_0) = \{x \in \mathbb{R}^2 \mid x_1 \leq 3\}$ and $Inv(l_1) = \{x \in \mathbb{R}^2 \mid x_2 \leq 2\}$



(b) Region graph of the timed automaton in Figure 2.3a

Figure 2.3: A timed automaton and corresponding region graph.

Although the reachability analysis for timed automata is decidable, the state space of timed automata can be huge. Bounds on the number of clock regions can be derived of a timed automaton. These are based on integer bounds that originate from an invariant, for example. The lower and upper bound on the number of regions for a timed automaton consisting of N clocks ($|C|$) and the largest constraint on each clock x_i of C_i , are as follows:

$$N! \prod_{i=1}^N C_i \quad (\text{lower bound}) \quad (2.15)$$

$$N! 2^N \prod_{i=1}^N (2C_i + 2) \quad (\text{upper bound}) \quad (2.16)$$

From this follows that the number of regions is finite, however, the upper bound on the number of regions is exponential in number of clocks. A more compact representation of clock regions is obtained by using so called *zones*. Zones are unions of regions and can efficiently be stored in difference bound matrices [Dil89]. Although more efficient representations of timed automata exist, the problem of model checking of timed automata is proven to be *PSPACE-complete* [ACD93].

Multiple timed automata are composed together to form one larger concurrent system. The parallel composition $TA_1 \parallel TA_2$ of timed automaton TA_1 and TA_2 uses *handshaking actions* H , where $H \subseteq Act_1 \cap Act_2$. We use these handshaking actions to communicate between both timed automata. Such a handshaking action is performed simultaneously across both timed automata, only if both corresponding transitions are possible, e.g. both satisfy their guards. Using these handshaking actions, concurrent multiprocessor real-time systems can be modeled using timed automata. However, the state space of the required timed automaton can be very large.

2.2 DATAFLOW MODELS

Dataflow models have been introduced as models where the concurrency in applications is made explicit. These dataflow models allow applications to be modeled and analyzed that are executed on multi-processor systems. In a dataflow model, individual computations are each represented by a so called *actor*. Multiple of these actors are combined to form a dataflow graph. Actors in such a graph communicate by sending data, in the form of a *token*, across *edges* connecting the actors. An actor can only start when sufficient tokens are accumulated on these edges. The actor then fires and consumes data. Since these actors fire solely dependent on incoming data, infinite streams of incoming data are therefore supported. This makes dataflow models suitable to model stream processing applications. Since applications modeled as dataflow models execute based on presence of data, these dataflow models are also called *data-driven*.

A *Homogeneous Synchronous Dataflow (HSDF)* graph is defined as $G = (V, E, \delta, \rho)$ where

- » V is a finite set of actors
- » $E \subseteq V \times V$ is a finite set of edges
- » $\delta : E \rightarrow \mathbb{N}_0$ assigns a number of initial tokens to each edge
- » $\rho : V \rightarrow \mathbb{R}_0$ assigns a firing duration to each actor

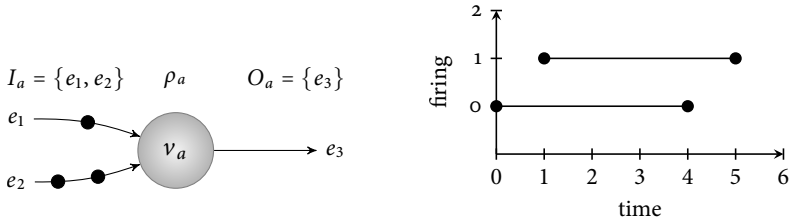
In an HSDF graph, a set of actors, V , is connected by a set of directed edges, E . An edge $(v_a, v_b) \in E$, where $v_a, v_b \in V$, in short e_{ab} , represents an unbounded queue. For such an edge e_{ab} , v_a produces tokens and v_b consumes tokens. Initially, these edges contain a number of initial tokens as specified by δ . An actor is *enabled* to fire if there are sufficient tokens on all its incoming edges. The set of incoming edges and outgoing edges, $I_a, O_a \subseteq E$, of v_a is defined as:

$$I_a = \{(v_i, v_j) \in E \mid v_j = v_a\} \quad (\text{incoming edges}) \quad (2.17)$$

$$O_a = \{(v_i, v_j) \in E \mid v_i = v_a\} \quad (\text{outgoing edges}) \quad (2.18)$$

The number of tokens required on the incoming edges of an actor to be enabled to fire is determined by the *firing rule* of the actor. For all Homogeneous Synchronous Dataflow (HSDF) actors, the firing rule states that each incoming edge in I_a should contain at least a single token before an actor is enabled. In a *self-timed execution* of a dataflow graph, all actors fire as soon as they are enabled. After an actor finishes its firing it atomically produces tokens on all outgoing edges in O_a , a single token per edge for HSDF actors. Multiple overlapping firings of an actor are also allowed, as long as there are sufficient tokens on its incoming edges to enable the different firings.

Time is introduced in dataflow models by assigning a *firing duration* ρ to each actor [SB09]. The firing duration is defined as the time between the start and finish



(a) Example of an actor in an HSDF graph. (b) Overlapping firings when a second token arrives on e_1 at time 1.

Figure 2.4: A dataflow actor of which the first two firings as shown.

of a firing of an actor. The firing duration of an actor is often bound within an interval such that $\{\rho_a \in \mathbb{R}_0 \mid \hat{\rho}_a \leq \rho_a \leq \hat{\rho}_a\}$.

An example of an HSDF actor is shown in Figure 2.4a. Overlapping firings of this actor will for example occur when a second token arrives on e_1 at time 1, as shown in Figure 2.4b, where $\rho_a = 4$.

A transition system, such as timed automata, can also be derived for dataflow models under the assumption that all firing durations are natural numbers, since only natural numbers can be expressed in clock constraints in transition systems. This gives an idea of the difference in expressiveness of dataflow models compared to timed automata, which is detailed further in Section 2.3. Concurrency is made explicit in such a transition system using a multiset of firings for each actor [GG⁺06]. This multiset contains the remaining firing durations for all firings that have started, but not yet finished. Each time, time progresses by a clock cycle, all these remaining firing durations are decremented. In the transition system for HSDF graphs, the state is a tuple (γ, τ) , where γ associates with each edge $e \in E$ the number of tokens present on that edge in that state, whereas τ keeps track of the time progress of actors. The time progress $\tau : V \rightarrow \mathbb{N}^{\mathbb{N}}$ associates with each actor $v_a \in V$ a multiset that represents the remaining firing duration for different concurrent firings of v_a . The initial state of the system consists of the initial tokens in the dataflow model, in combination with an empty multiset for each actor since no firing has started, such that the initial state equals $(\delta, \{v_a, \{\} \mid v_a \in V\})$.

The transition system supports overlapping firings of an actor, which is the case when a firing starts in between the start and finish of another firing of the same actor, as already shown in Figure 2.4b. Overlapping firings are supported since τ can keep track of multiple simultaneous firings of an actor. In case there are sufficient tokens on all incoming edges, for example ≥ 2 , and actor that fire data-driven, two concurrent firings will automatically start at the same point in time.

Three types of transitions are possible in this transition system of HSDF graphs: the start of a firing of an actor, its finish, and progression of time. The transition relation is therefore defined as $(\gamma, \tau) \xrightarrow{\beta} (\gamma', \tau')$, where there are three options for

β reflecting the three types of allowed transitions $\beta \in (V \times \{start, end\}) \cup \{clk\}$. The three transitions are defined for an actor v_a as:

- (1.) $(v_a, start)$ if $\forall e \in I_a: \gamma(e) \geq 1$ then
 - (1a.) $\forall e \in I_a: \gamma(e)' = \gamma(e) - 1$
 - (1b.) $\tau' = \tau[v_a \mapsto \tau(v_a) \uplus \{\rho(v_a)\}]$
- (2.) (v_a, end) if $0 \in \tau(v_a)$ then
 - (2a.) $\forall e \in O_a: \gamma(e)' = \gamma(e) + 1$
 - (2b.) $\tau' = \tau[v_a \mapsto \tau(v_a) \setminus \{0\}]$
- (3.) clk otherwise: (when no $start$ or end possible) then
 - (3a.) $\gamma' = \gamma$
 - (3b.) $\tau' = \{(v_a, \tau(v_a) \ominus 1) \mid v_a \in V\}$

where \uplus is the multiset union and $\tau(v_a) \ominus 1$ reduces all elements in $\tau(v_a)$ by one. The start transition of v_a is allowed if v_a is enabled (1.). The transition decrements the number of tokens in I_a , (1a.), and add the firing duration of v_a to the multiset $\tau(v_a)$, (1b.). An end transition for v_a occurs when there is a firing with a remaining firing duration of zero (2.). The transition produces a token on all outgoing edges (2a.) and removes the firing from the multiset (2b.). In case no start or end transition is possible for any actor, a clk transition will occur (3.). For a clk transition the number of tokens on the edges remains unchanged (3a.), but the remaining firing duration of all firings of all actors is decremented by one clock cycle (3b.). The order of $start$ and end events is not important, however, no clk transition is allowed when other $start$ or end events are possible.

2.2.1 PROPERTIES OF DATAFLOW MODELS

In this section we discuss important properties of dataflow models that we use throughout this thesis. These properties include: functional determinism, deadlock, consistency, auto-concurrency, and monotonicity.

For a dataflow model to be *functionally deterministic*, two conditions need to be fulfilled: firings of actors need to be functional and actors need to have sequential firing rules [LP95]. A firing is *functional* if it is free of side-effects and the output of a firing is purely based on the currently consumed input. Moreover, an actor can represent a computation that updates its state, since that corresponds to an actor with a feedback loop with a single token, which prevents multiple simultaneous firings of the same actor. The firing rules are *sequential* when a unique firing rule is selected by using blocking reads on the incoming edges in a predefined, but potentially input data value dependent, order.

All actors in a dataflow graph should be able to execute infinitely often in the long run. A dataflow graph is said to *deadlock* if there is an actor that at some point cannot make any more progress. Such a deadlock can for example occur when there are insufficient initial tokens on an edge such that the consuming actor of

that edge is never enabled. For HSDF graphs, a graph deadlocks when there is a cycle in the graph without initial tokens.

The number of tokens on each edge in a dataflow graph should be bounded. This bounded number of tokens ensures that a dataflow graph can be executed within bounded memory. *Consistency* of a dataflow graph can be verified by constructing a *topology matrix* $\Gamma : |E| \times |V|$ of a graph. For each edge in the graph, this matrix contains the number of tokens produced on it by an actor after one firing, and with negative numbers the tokens consumed from it. In case there is a *self-edge*, the same actor both consumes and produces the same number of tokens from / on an edge, a value of 0 is used in the matrix. A dataflow graph is consistent if $\text{rank}(\Gamma) = |V| - 1$. Moreover, a *repetition vector* \mathbf{q} can be derived from a consistent graph where $\Gamma \mathbf{q} = 0$. This repetition vector specifies the relative number of firings between all actors such that the graph returns to the same state, i.e. the same number of tokens on all edges, after all actors v_i have fired q_i many times.

In case an actor has a sufficient number of tokens on its incoming edges for multiple firings, these firings may overlap in time. These simultaneous firings of the same actor is called *auto-concurrency*. When for example a second token is produced on e_1 in Figure 2.4a before the other tokens are consumed, two simultaneous firings are possible. The combination of both auto-concurrency and the possibility for a different firing duration of an actor for each firing can result in reordering of tokens. This reordering occurs when a firing that is started later, finishes earlier when it has a lower firing duration than the firing that started before it. Reordering of tokens breaks the functional determinism of dataflow models in case the tokens model events of which the order is important. Auto-concurrency can be prevented in a dataflow model by adding a self-edge containing a single initial token to each actor, to enforce sequential firings of an actor.

Functionally deterministic dataflow models have a *monotonic* temporal behavior [WBS09]. What this means is that the start times of actors during self-timed execution of a dataflow graph can only become earlier when actors produce tokens earlier. Therefore, tokens are never produced later when the firing duration of actors are decreased, or when the concurrency is increased by adding more initial tokens on a cycle in the graph. The maximum number of concurrent firing of all actors on a cycle in the graph is bounded by the number of initial tokens on that cycle. Auto-concurrent firings of the same actor can cause reordering. Monotonicity also holds for dataflow models that use tokens that are extended with indices [Hau15, K⁺16], like the Cyclo-Static Dataflow with auto-concurrency (CSDF^a) model. The next section discusses different dataflow models in more detail.

2.2.2 MORE EXPRESSIVE DATAFLOW MODELS

Next to the HSDF model, there are many more well-known dataflow models that are related to the work presented in this thesis. The Synchronous Dataflow (SDF), Cyclo-Static Dataflow (CSDF), CSDF^a and Structured Variable-Rate Phased Dataflow (SVPDF) model are more expressive than HSDF. Mainly, these models have

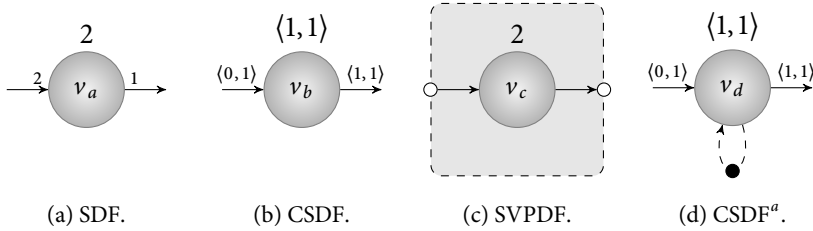


Figure 2.5: Examples of different dataflow models, each showing a single actor.

more expressive firing rules of the actors compared to firing rule of HSDF actors. Figure 2.5 shows the graphical representation of these dataflow models. In this section, these more expressive dataflow models will be discussed briefly.

The first dataflow model that targeted parallel computation was the Synchronous Dataflow (SDF) model [LM⁺87]. In this model, the production and consumption rate for each edge are known *a priori*. The production and consumption *rate* specifies the number of tokens that are atomically produced on c.q. consumed from an edge for each firing of the producing c.q. consuming actor. The differences in production and consumption rates are for example used in Digital Signal Processor (DSP) applications to represent up- and downsampling. Initially, the SDF model did not include a notion of time. However, static schedules can be created for SDF models. In such a *static schedule*, the order of firings of different actors is defined at compile time. These schedules can be repeated infinitely, where the size of the schedule depends on the production and consumption rates of the actors. In this thesis we will mainly focus on applications where all rates are one, and an actor consumes a single token and produces a single token for each edge it is connected to. This single rate version of the SDF model is referred to as HSDF.

The *SDF* model [LM⁺87] extends the HSDF model with positive integers for the number (\mathbb{N}) of tokens consumed and produced on an edge. The firing rule of an SDF actor therefore specifies the number of tokens the actor requires on each of its incoming edges before it can fire. Also an integer number is specified for the number of tokens produced after the firing for each of its outgoing edges. For cycle graphs these consumption and production rates can lead to inconsistencies as discussed in Subsection 2.2.1. As a result, the number of tokens on a cycle is not bounded. For a consistent SDF graph, a *repetition vector* can be derived that specifies the relative number of firing between all actors. After all actors in a consistent graph fire the number of times as defined by the repetition vector, the number of tokens on each edge is unchanged. This repetition vector is also used to transform an SDF graph into an equivalent HSDF graph [SB09]. The entries in the repetition vector directly determine how many HSDF actors are required to represent an SDF actor. After the transformation, analysis techniques for HSDF graphs can be applied.



In *CSDF* [B⁺96], each SDF actor can consist of multiple phases, each with its own firing rule and firing duration. Multiple firing rules are therefore allowed for a *CSDF* actor, however, the actor must cycle through this list of phases and firing rules in a predefined order, i.e. cyclo-static. After an actor finished the last phase in the list, the actor returns to the first phase. In contrast to SDF actors, the number of tokens produced or consumed can be zero in some phases of an *CSDF* actor. Also *CSDF* graphs can be transformed into *HSDF* graphs [B⁺96].

SVPDF is a dynamic dataflow model [GHB13]. In such a dynamic dataflow model, actors execute data-dependent. In these dynamic dataflow models, not all actors can be scheduled at compile time, and therefore require run-time scheduling. Other dynamic dataflow models like Boolean Dataflow (BDF), add dynamic switch and select actors to SDF. Including these dynamic actors makes the BDF model Turing complete [BL93]. It is therefore undecidable in general whether a BDF graph deadlocks or whether it can be executed within bounded memory. However, an *SVPDF* graph, consists of hierarchical blocks, where the sub-graph inside a block can be analyzed as an SDF graph. These blocks correspond to while-loops that execute data-dependently for potentially an infinite amount of times. Moreover, these blocks can be nested. The top level of a *SVPDF* graph consists of nested blocks, where the bottom level contains an SDF sub-graph. An *SVPDF* graph can automatically be constructed from a program that is described by nested loops [GHB13, GHB14]. Since the original nested loop program can always be executed sequentially, also the parallel implementation can always be executed in the same order as the sequential execution order in the nested loop program.

The *CSDF^a* model [K⁺16] is closely related to the *CSDF* model. However, in *CSDF^a* the consumption order of tokens is not based on the production times of tokens and the FIFO order of tokens on edges. Tokens are annotated with an index, and tokens can be consumed out-of-order, independently of time, based on the order of the index of tokens. This out-of-order consumption, and as a result also production of tokens, allows *auto-concurrent* firings of actors, while still maintaining functional and temporal deterministic behavior of *CSDF^a* graphs.

2.2.3 ANALYSIS

Two types of analysis techniques have been developed to analyze dataflow models: symbolic analysis and simulation-based approaches. The symbolic analysis methods can only be applied to *HSDF* graphs, and therefore, requires a transformation to an *HSDF* graph when a more expressive dataflow models is used. This transformation is expensive as the resulting *HSDF* graph has a exponential number of actors in the worst-case compared to an originally more expressive dataflow graph [PBL95]. Simulation-based approaches explore the state space of a dataflow graph without the need for a transformation to *HSDF* graphs, and its analysis resembles the analysis of a timed transition system as discussed in Subsection 2.1.4.

Symbolic analysis of dataflow models relies on the fact that the slowest cycle in a strongly connected dataflow graph with constant firing durations determines the

throughput of the graph. This slowest cycle is the cycle for which the cycle ratio is equal to the Maximum Cycle Ratio (MCR) [Rei68]. For each simple cycle (where no actors or edges are repeated) the sum of firing durations of all actors on the cycle is divided by the sum of tokens on the edges that form the cycle. The MCR $\mu(G)$ of an HSDF graph G is defined as:

$$\mu(G) = \max_{c \in \mathcal{C}(G)} \frac{\sum_{v_i \in V(c)} \rho_i}{\sum_{e_i \in E(c)} \delta(e_i)} \quad (2.19)$$

where $\mathcal{C}(G)$ is the set of simple cycles in G , $V(c)$ the set of actors on a cycle c , and $E(c)$ the set of edges which form this cycle. The throughput of an HSDF graph is the inverse of this MCR. Polynomial time algorithms have been developed to calculate the MCR [DIG99]. Although Howard's algorithm [CTCG⁺98], has a higher complexity, it often performs better in practice [DIG99].

Given a throughput constraint for a dataflow graph, also the minimum required number of initial tokens can be determined analytically. These initial tokens can for example be used to model the capacity of a blocking FIFO buffer. The number of initial tokens on each cycle follows from the sum of the firing durations and the throughput constraint.

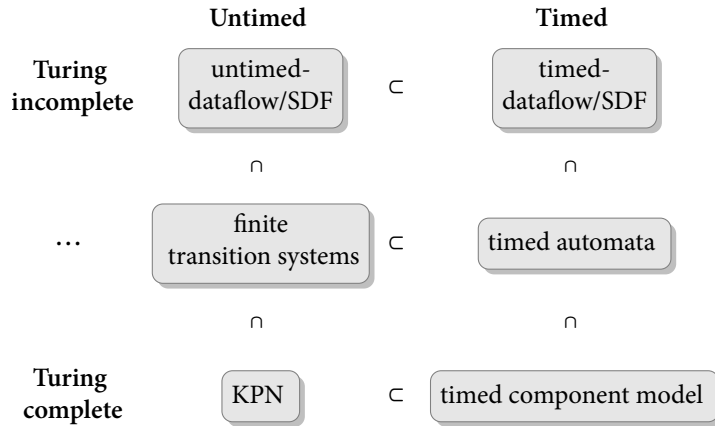
Simulation-based analysis approaches do not require a transformation to an HSDF graph. In [GGG⁺06], this transformation is avoided by analyzing a transition system as described in Section 2.2, extended with production and consumption rates. However, for some types of SDF graphs there is a long initial phase for simulation-based approaches, such that transforming the graph to HSDF and using MCR analysis is nevertheless faster [dG⁺12].


2.3 ANALYSIS MODELS FOR CONCURRENT SYSTEMS

In this chapter, we have introduced two formal models for the analysis of concurrent systems: timed automata and dataflow models. These two models are used throughout this thesis. In this section, we intuitively present the relation between these two models and some other strongly related models.

Firstly, analysis models can be characterized as either timed or untimed. Secondly these models differ in what can be expressed in them, up-to the point where these models even become Turing complete. Figure 2.6 visualizes this relation for a number of models and shows intuitively their relation using the subset relation. In general, the analysis of a less expressive model is easier.

Untimed models are a subset of timed analysis models. The untimed version of SDF has been introduced in [LM⁺87]. Later dataflow models were extended with time, by introducing a firing duration of actors [SB09]. Also for timed automata, untimed versions exist, of which finite transition systems are an example. Since this thesis focuses on real-time systems, mainly timed analysis models are of interest.



 Figure 2.6: Relation between different formal models for the analysis of concurrent systems.

Many dataflow models exist, and we have presented some of these models in Subsection 2.2.2. In order to simplify the comparison between analysis models, we only consider the most popular dataflow models in this section, which are not Turing complete and do not allow choice to be expressed. Therefore, we do consider the SDF, HSDF and CSDF models, but exclude dynamic dataflow models such as BDF. These dataflow models have strong analytical properties and are relatively easy to analyze.

The relation between timed-dataflow models and timed automata will be discussed in more detail in Chapter 5. In that chapter, functional deterministic dataflow models that support reordering are transformed into timed automata. These timed automata can be analyzed to derive properties of the dataflow model. The main advantage of timed automata is that more details of a system can be included in the model compared to what can be modeled into dataflow models, which leads to more accurate analysis results.

The most expressive models are Turing complete, and are shown in the last row in Figure 2.6. For Turing complete models, it is in general undecidable whether analysis of the model will eventually terminate. Therefore, these models are not practical for the analysis of real-time systems, where it must be determined whether the temporal constraints will always be met. Kahn Process Networks (KPNs) is such a model that is Turing complete [Kah74, LP95]. The timed component model [H⁺16] can be seen as a timed version of KPNs. Both functional and temporal determinism can be maintained in the timed component model by making use of an index of an event besides a timestamp, which defines when an event has been produced.

The relation between the analysis model in Figure 2.6 does *not only* hold for the expressiveness of these models. Whereas the dataflow models in the top row have re-

strictive firing rules, also the largest over-approximation is applied for these models. The advantage of this over-approximation is that the resulting monotonic model can be analyzed with an algorithm that typically has a low run-time, and which makes use of iterative fixed-point computation. Models that are displayed lower in Figure 2.6, like timed automata, require a smaller over-approximation to ensure that the analysis problem is decidable. As a result, the analysis results are more accurate, but this comes at the cost of a larger run-time of the analysis algorithm.

The results for dataflow analysis are often algebraic, and valid for arbitrary parameters such as buffer sizes and firing durations. This is in contrast to the results of a model checker of timed automata, which are only valid for specific model parameters. A minor change in the model parameters, e.g. an adaptation of a buffer size, will require the model checker to be run again, which leads to a higher run-time, and does provide less insight in the trade-offs than an algebraic expression.

In the next chapters techniques are described that extend the scope of both dataflow and timed automata based analysis approaches for real-time multiprocessor systems. We will describe techniques that improve the accuracy of dataflow analysis, as well as techniques that reduce the run-time of timed automata based analysis approaches.

2.4 SUMMARY

Two formal models for the analysis of concurrent systems were introduced in this chapter: timed automata and dataflow models.

To understand the analysis of timed automata, we introduced transition systems in Section 2.1. Then we discussed reachability analysis, which is used to derive properties of these transition systems. Including time in transition systems might at first hand appear to make their analysis undecidable, since there can be an infinite number of cases that need to be considered. However, for timed automata a finite bisimulation, i.e. an equivalence relation, can be constructed for region graphs, which makes analysis of timed automata decidable.

The semantics of the simplest dataflow model, i.e. HSDF, was presented in Section 2.2. This model can also be represented as a transition system, which is similar to timed automaton. Several properties of dataflow models relevant to this thesis were discussed. Slightly more expressive dataflow models were also briefly introduced. One way to analyze these more expressive models is to transform them into HSDF graphs and use MCR analysis to derive the throughput of these graphs.

The relation between these two models and some other strongly related models was presented in Section 2.3. We classified these models on whether they are timed, and what can be expressed in the models.

ENFORCING MUTUALLY EXCLUSIVE TASK EXECUTION IN MODAL APPLICATIONS

ABSTRACT – Modal data-driven real-time applications, where tasks are scheduled using budget schedulers, often result in pessimistic analysis results. The pessimism is, among others, a result of interference that occurs at the moment when the application is switching between different modes. We introduce a lock in this chapter, which is used to make tasks that belong to different modes to execute mutually exclusive to prevent this type of interference. A dataflow model from a sequential specification of a modal application is automatically generated, including constraints resulting from locks. This dataflow model can be analyzed to verify satisfaction of temporal constraints.

The effects of *mode changes* on the schedulability of a task set executed on single processor systems have been extensively studied [S⁺89, TBW92b, RCo4b, Gua09a, S⁺10]. These works require that mode change control software in the kernel of the operating system takes care that a new mode change is not started during a mode change. An exception are dataflow analysis techniques [WBS10, GS10, GHB13, S⁺13] which do allow the start of mode changes during a mode change. Furthermore, these dataflow analysis techniques are intended for multiprocessor systems, which we address in this thesis. However, dataflow analysis techniques

This chapter is based on [GK:2].

ignore that the execution of tasks belonging to different operation modes can be enforced to execute mutually exclusive. This results in more pessimism in the analysis results.



Tasks in different modes are often not active at the same time and can therefore share resources without interfering with each other. Dataflow analysis techniques can be used to analyze this resource sharing for systems in which budget schedulers [WBS09, SBW09] are applied. These budget schedulers guarantee a minimum budget during a replenishment interval and thereby guarantee that always, thus also during mode transitions, at least a minimum amount of processor time is reserved for the execution of each task. As a result, a single Worst-Case Response Time (WCRT) can be derived per task at design time. Throughput analysis of the task graph is based on these WCRTs. The reservation of the resources simplifies this analysis drastically. However, dataflow analysis techniques neglect that tasks that execute mutually exclusive in different modes will not interfere. By enforcing tasks to execute mutually exclusively, the pessimism in the analysis results can be reduced.

In this chapter we present a dataflow analysis approach which takes into account that tasks execute mutually exclusive in different modes, which can reduce the pessimism in the analysis results. To enforce mutual exclusive execution when it is beneficial, a new lock is introduced. This lock allows parallel execution of a group of tasks, but enforces serial execution between groups. The lock is inserted by a compiler, which transforms a sequential specification of the application into a parallel task graph and an SVPDF model [GHB13]. The lock is inserted in such a way that all tasks can still execute in the same order as in the sequential input specification; therefore deadlock-free execution is guaranteed. The generated SVPDF model is a dynamic dataflow model in which mutual exclusion can be expressed. This SVPDF model is used to determine whether addition of a lock results in satisfaction of the throughput constraint and is used to compute the required buffer capacities.

The outline of this chapter is as follows. First, we position our contribution relative to related work in Section 3.1. In Section 3.2, we present the basic idea behind our approach. The different types of mutual exclusivity that we distinguish are described in Section 3.3. That mutual exclusivity results in tighter WCRTs is shown in Section 3.4. The realization of our lock is described in Section 3.5. Furthermore, it is shown that the generated parallel task graph that includes locks is always deadlock-free. The SVPDF model that we use is described in Section 3.6. Modeling mutual exclusivity in an SVPDF model of the application is explained in Section 3.7. The applicability and benefits in terms of throughput and processor utilization for a WLAN application are discussed in Section 3.8. Finally, the conclusions are stated in Section 3.9.

3.1 RELATED WORK

In this section we compare our lock with locks described in literature and discuss other approaches to handle and analyze mode switches.

Mutual exclusive access to resources is obtained by making use of locks. Such locks are usually implemented with atomic read-modify-write operations such as test-and-set and load-link-store conditional [CSG99]. These type of locks are unsuitable for real-time systems because they are based on a retry mechanism which makes them non-starvation-free [Her88]. Detailed knowledge, execution rates and execution times, of all tasks using these locks is then required to be able to provide temporal guarantees. Starvation-free versions of locks do exist but often incur a much higher overhead. Examples of starvation-free locks are the Bakery lock [Lam74] and Szymanski's mutual exclusion algorithm [Szy88]. The lock proposed in this chapter is starvation-free but does not introduce a high overhead. A key difference with other locks is that the proposed lock enforces an order in which groups of task can acquire the lock.

Ordinary load and store operations are used to guarantee mutual exclusive access in the Bakery lock and Szymanski's mutual exclusion algorithm. These locks require that sequential consistency [Lam79] is supported as the memory consistency model by the multiprocessor hardware. FIFO buffers can also be realized using ordinary load and store operations [N⁺02]. However, they require a much weaker memory consistency model [vdBB07] that only guarantees that writes issued by a processor complete in the order that they are issued, and that read and writes that access the same memory do not overtake each other. Circular buffer [BB]So8 implementations have been proposed that can be seen as a generalization of these FIFO buffers because they allow multiple readers and writers. The lock proposed in this chapter is based on the same principles as these circular buffers.

The Giotto approach [HHK01] is suitable for modal real-time applications, but requires that the tasks are executed on a time-triggered multiprocessor system. In such a system, tasks are executed according to a precomputed schedule. This schedule is strictly periodic except during mode transitions. Moreover, this schedule is constructed at design time purely based on the WCET of the tasks. The approach described in this chapter relies on a data-driven execution of the tasks which results in more scheduling freedom and allows aperiodic task execution. To guarantee satisfaction of the throughput constraint a worst-case schedule can be computed for aperiodic tasks based on a more accurate two parameter work-load characterization [H⁺13b] of the tasks instead of using WCETs.

Techniques for the prevention of overloads that result in violation of the WCRTs during the transition between modes on single processor systems have been deeply investigated in the real-time literature [S⁺89, TBW92b, Gua09a] and a survey of mode change protocols for FPP scheduled systems can be found in [RC04b]. Most of these protocols delay the start of tasks during a mode change. Some of these works are geared towards servers of which budget schedulers are a subclass [S⁺10].

All these approaches assume that a subsequent mode change is not started before the previous mode change completes. Furthermore, they are only applicable for acyclic task graphs while the method described in this chapter can handle cyclic task graphs and overlapping mode changes are supported. Cycles in the task graphs are a result of data dependencies and of the use of buffers with a bounded capacity.

Classical dataflow models such as HSDF, SDF [LP95] and CSDF [B⁺96] as introduced in Section 2.2 can only model static applications, i.e. applications of which their synchronization behavior is independent of the input data. In [DSB⁺13], SDF models are analyzed using static-order schedules, however, conditionally executed tasks cannot be encoded in these static-order schedules. Therefore, these models are unsuitable for the modeling of modal applications. However, the recently introduced Variable-Rate Phased Dataflow (VPDF) [WBS10] and Scenario-Aware Dataflow (SADF) [T⁺06, S⁺13] models are suitable to describe modal applications. Generation of a parallel task graph and a corresponding structured version of the VPDF model, which is called SVPDF, is presented in [GHB13]. An advantage of this approach is that the task graph and the corresponding analysis model are deadlock-free. In this chapter we present the modeling of mutual exclusive execution in the SVPDF model. This model is used to show that an admissible *worst-case schedule* exists that satisfies the throughput constraint imposed by the periodic source of the application. A schedule is admissible if all tasks do not execute before sufficient data and space is available. The production moments of the data in this worst-case schedule are upper bounds on the production moments of the tasks in all possible (aperiodic) run-time schedules. The SVPDF model is generated by a compiler from a sequential description of the application in the Omphale Input Language (OIL) language.

The dataflow analysis techniques discussed in this chapter are applicable in combination with budget schedulers [WBS09]. As discussed in Subsection 1.2.1, budget schedulers are a subclass of servers that guarantee a minimum cycle budget in a replenishment interval. In [WBS09] it has been shown that the worst-case effects of run-time task scheduling by budget schedulers can be included in firing durations of the actors of dataflow graphs. A budget scheduler with priorities is introduced in [SBW09] which allows reducing the WCRT of one task at the cost of an increased WCRT of the other tasks. Most budget schedulers are work-conserving and as a result allocated budget for a task that is not used by this task becomes available for other tasks.

The budget schedulers used in this chapter are a subclass of the servers that are used in [S⁺10]. The approach in [S⁺10] adapts the parameters of the server during a mode transition by a separate mode change controller in the scheduling kernel. The approach described in this chapter does apply a different approach in which parameters of the scheduler are implicitly adapted due to the fact that resources are not used anymore by some of the tasks after a mode transition. Furthermore, the activation and deactivation of tasks is a responsibility of the tasks themselves and is part of the description of the application. This description is a sequential program

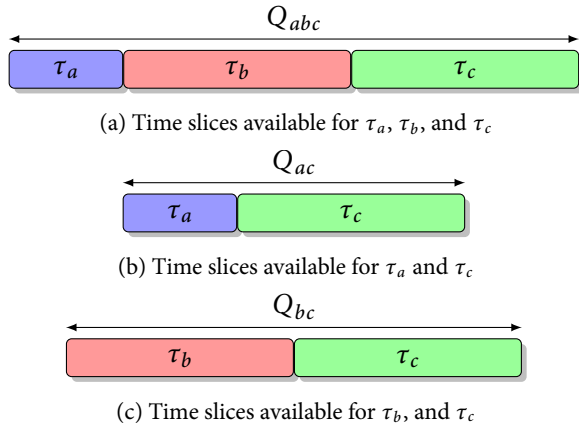


Figure 3.1: Replenishment interval of mutually exclusive tasks

with while-loops and if-conditions instead of tasks with scheduling parameters and a separate description of the behavior of a mode change controller. Another important observation is that we are dealing with a multiprocessor system and therefore the implementation of a (centralized) mode change controller would not be straightforward.

3.2 BASIC IDEA

In this section we present the basic idea behind our approach. With a didactic example, we illustrate that the WCRTs of repetitively executed tasks can be reduced and a higher minimum throughput is obtained if information about mutually exclusive execution is taken into account. We derive these tasks from a sequential program in which if-conditions and while-loops are used to describe modes and mode transitions. Tasks execute data-driven such that variations in execution time can be exploited [H⁺13b]. We show the counterintuitive effect that tasks resulting from different while-loops or branches of an if-then-else statement do not necessarily execute mutually exclusive. We explain why mutual exclusivity can be enforced with locks without introducing deadlock as a result of that the tasks in the task graph are generated from a sequential program. We furthermore explain the modeling of the sequence constraints that result from a lock in an SVPDF model. This model is used for throughput analysis.

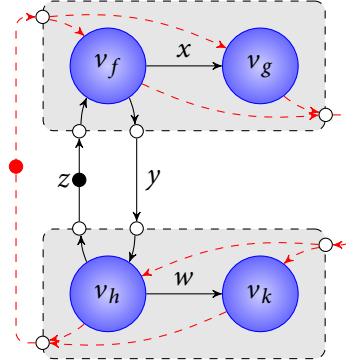
Figure 3.1 illustrates that the WCRTs of tasks can be reduced by taking into account that tasks execute mutually exclusive. One replenishment interval of a budget scheduler is shown in Figure 3.1a, in which three time slices are available for respectively the execution of tasks τ_a , τ_b , and τ_c . These tasks execute until they exhaust their budget after which they continue their execution in the subsequent replenishment interval. A task voluntarily suspends its execution in case it is not enabled because



```

1 z = 0;
2 loop {
3   loop{
4     f(out x, out y, z);
5     g(x);
6   } while(x);
7
8   loop{
9     h(out w, out z, y);
10    k(w);
11  } while(w);
12 } while(1);
  
```

(a) Code example



(b) SVPDF model

Figure 3.2: Mutual exclusion of while-loops

there is insufficient input data or output space to start the execution of the task after which a task switch occurs. In our example we will assume a budget of one, two, and two time units for tasks τ_a , τ_b , and τ_c respectively.

The *replenishment interval* is reduced in case τ_a and τ_b execute mutually exclusive. The length of the replenishment interval becomes three in case only task τ_a executes and four in case that only τ_b executes, as shown in Figure 3.1b and Figure 3.1c respectively. A reduction of the replenishment intervals of the tasks results in a reduction of the WCRTs, as directly follows from the WCRT equation [WBS07b] in Equation 3.1. In this equation \hat{R}_i is the WCRT, Q_i the replenishment interval, B_i the WCET, and S_i the budget of τ_i .

$$\hat{R}_i = B_i + (Q_i - S_i) \left\lceil \frac{B_i}{S_i} \right\rceil \quad (3.1)$$

During a mode transition it might occur that τ_a finishes at the end of its slice and immediately after that a mode switch occurs after which τ_b starts its execution. However, this does not result in a longer WCRT for τ_a and τ_b than Q_{ac} and Q_{bc} respectively, as will be explained in Section 3.4.

The tasks are derived by a multiprocessor compiler from a sequential program of which an example is shown in Figure 3.2a. Each while-loop in this program corresponds to a mode of the application. After parallelization, a task graph is obtained. This task graph can be modeled with the SVPDF model in Figure 3.2b. Every function in the sequential program corresponds to one task in the task graph and every task corresponds to one actor in the SVPDF model. These actors are represented by nodes in the SVPDF model. Every variable in the sequential program is converted

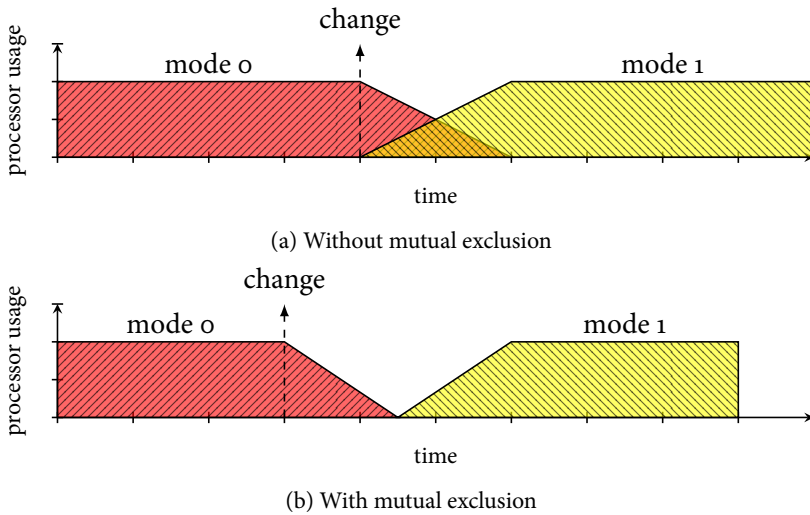
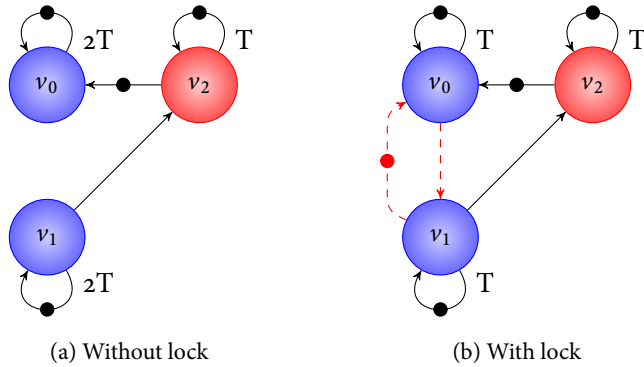



Figure 3.3: Processor usage during a mode switch

in a circular buffer and each circular buffer corresponds to at least one edge in the SVPDF model.

To understand the example it is sufficient to assume that the SVPDF model in Figure 3.2 has an HSDF-like behavior, i.e., actors can only fire if at least one token is present on all its inputs and a token is produced on all outputs when an actor finishes its firing. We can therefore conclude from this model that actor v_f and v_h cannot fire at the same time as a result of the cycle in the model with one token. This cycle is a result of the fact that function h in the second while-loop cannot start before the value y of the first while-loop becomes available. Also, function f cannot execute for successive executions before the value z from the second while-loop becomes available. However, we can also conclude from the model that actor v_g and v_k can fire at the same point in time because they are not part of a cycle with a single token. They can fire at the same time if there is an input token present for v_g and v_k , which can occur after v_f has produced a token on each of its outputs and as a result v_h fires and produces a token for v_k before v_g finishes its execution. A similar situation can occur during the execution of the task graph after τ_f has produced a data item for τ_g and then a mode switch occurs.

Figure 3.3a illustrates intuitively that executions of tasks belonging to different modes can execute simultaneously during a mode transition. Locks can be used to prevent that tasks belonging to different modes execute simultaneously. This situation is shown in Figure 3.3b. This figure also shows the counterintuitive effect that the mode transition will in some case occur earlier despite that less tasks execute in parallel during a transition. The reason is that the WCRTs of the tasks can be reduced by making use of the knowledge that they execute mutually exclusively. This



 Figure 3.4: HSDF model before and after adding a lock

results in an improvement of the minimum throughput as computed with dataflow analysis.

Mutual exclusive execution is usually only enforced between tasks that execute on the same processor. Therefore even if locks are applied, a mode transition usually starts on different processors at different moments in time. As a result, tasks belonging to different modes will execute on different processors at the same moment in time. The lock presented in this chapter can also be used to enforce mutually exclusive execution of tasks on different processors which might be beneficial because it can reduce contention at shared memory ports and thereby reduce the execution times of the tasks. However, this option will not be detailed in this chapter.

The enforcement of mutual exclusivity with locks results in additional constraints on the order in which the tasks can execute. These constraints are modeled with the red dashed edges in the SVPDF model in Figure 3.2b. These edges form a cycle with one token and as a result the firings of all actors in a block do not overlap with firings of actors in another block. However the actors in a block can still fire concurrently.

The locks are added in the tasks in such a way that deadlock does not occur. This is possible because the tasks and the task graph are derived from a sequential program which is deadlock-free by definition. Therefore we can insert the acquires and releases of the locks according to the order defined by the sequential program that is still a valid order. Other execution orders of the tasks do not result in a different functional behavior of the tasks because we can rely on the fact that the task graph that we create can be represented as a functionally deterministic dataflow model.

Adding locks can reduce the WCRTs of the tasks but does not necessarily improve the throughput of the application. The reason is that the lock enforces besides mutual exclusivity, also the execution order as defined in the sequential program. This execution order might not be the order that results in the maximum throughput even in the case that the WCRTs are reduced. For ease of understanding we

illustrate this with an HSDF example instead of an SVPDF model. In Figure 3.4a an HSDF model is shown in which actor v_0 and v_1 execute on one processor and v_2 on another processor. The WCETs of the actors is T and \hat{R}_0 and \hat{R}_1 is then $2T$ under the assumption that each actor has a budget T . In this case the throughput is determined by the cycle with the highest cycle mean which is $2T$. Figure 3.4b corresponds to the case that τ_0 and τ_1 execute mutually exclusive as a result of a lock and that in the sequential program first the function that corresponds to τ_0 is executed before the function corresponding with τ_1 . As a result of the lock \hat{R}_0 and \hat{R}_1 are reduced to T . However, because the lock enforces an execution order, additional edges should be added in the HSDF model which are dashed and colored red in Figure 3.4b. These additional edges increase the maximum cycle mean to $3T$ and reduce the throughput to $\frac{1}{3T}$. This shows that the decision whether adding a lock is beneficial requires global analysis of the dataflow model.

3.3 TYPES OF MUTUAL EXCLUSIVITY

In this chapter we distinguish between two types of mutual exclusivity: intra-iteration and inter-iteration mutual exclusivity.

Tasks are *intra-iteration mutually exclusive* if there is no overlap in their execution within one iteration of a while-loop. An example of such mutual exclusivity is an if-else-statement in which during one iteration of the loop either functions in the if-branch or the else-branch execute, but not both. Another example is if there is a data-dependency between two statements, preventing them from executing simultaneously.

Tasks can also be *inter-iteration mutually exclusive*. This means that tasks derived from functions located in the same while-loop do not execute simultaneously when considering different loop iterations. An example of this type would be two functions where the first function writes to a variable read by the second function and vice-versa. In an SVPDF model such a case can be found if there is a block, modeling a while-loop, having two actors on a cycle with one token on that cycle. Tasks that are intra-iteration mutually exclusive do not have to be inter-iteration mutually exclusive. For example the two branches of an if-else-statement are intra-iteration mutually exclusive but due to a pipelined execution they do not need to be inter-iteration mutually exclusive. In such a case tasks from both branches can execute in different iterations of the surrounding while-loop simultaneously.

If tasks in a task graph have intra-iteration and inter-iteration mutual exclusivity, this can be exploited by the method introduced in this chapter. If tasks are both intra and inter-iteration mutually exclusively, they always execute mutually exclusive. The derived SVPDF model can be used to determine which tasks are mutually exclusive by searching cycles having one token on them. Exploiting this information does not require a change in the task graph. If such cycles do not exist, the lock introduced in Section 3.5 can be used to enforce mutual exclusivity and thus to create mutual exclusivity.

3.4 RESPONSE TIMES TDMA

In this section we show that reduced replenishment intervals can be substituted in the WCRT equation given that some of the tasks scheduled by a budget scheduler are deactivated during a mode change and others are activated. We make use of the fact that a budget scheduler allocates budgets to the tasks in a fixed cyclic order. We also make use of the property that a task returns control to the scheduler when not enabled.

As described in Section 3.2 it can be the case that during a mode transition all tasks scheduled on a processor use their time slice immediately after each other. Therefore, the situation as depicted in Figure 3.1a can occur. This suggests that full replenishment interval, Q_{abc} , should be used in Equation 3.1, which is however not always the case. Instead of the full replenishment interval, the shorter replenishment Q_{ac} and Q_{bc} intervals can be used for τ_a and τ_b respectively while for τ_c the longer interval Q_{abc} must be used to compute its worst-case response time.

The reason that a shorter replenishment interval can be used for τ_a and τ_b is that they are mutual exclusive and because the WCRT is defined as the maximum time between enabling and finish of task. We will make use of case-distinction to show that a valid bound on the WCRT is computed in case we use the reduced replenishment intervals in Figure 3.1b and Figure 3.1c in Equation 3.1 for the computation of \hat{R}_a and \hat{R}_b .

It is given that τ_a and τ_b execute mutually exclusive. Therefore we know that before a mode change only one of these tasks is active. As a consequence we can assume that Q_{ac} and Q_{bc} are valid replenishment intervals for τ_a and τ_b , respectively.

When a mode change occurs there can be a transition from the first mode in which τ_a is active to the second mode in which τ_b is active. After τ_a finishes its execution it will yield the processor. As a result τ_b will receive its budget in Q_{bc} time. A similar situation occurs for the transition from the second mode in which τ_b is active to the first mode in which τ_a is active. Here τ_a will receive its budget in Q_{ac} time after τ_b finishes its last execution in the previous mode.

Because in all the possible cases the budgets become available for τ_a and τ_b in respectively Q_{ac} and Q_{bc} we conclude that correct WCRTs are obtained when they are used in Equation 3.1.

For τ_c the situation is different than for τ_a and τ_b . For this task it can only be guaranteed that Q_c is available in every Q_{abc} because between two slices for τ_c there can be an execution of τ_a and τ_b while τ_c is enabled.

The same reasoning as applied in this section can be used to proof similar results for the case that an arbitrary number of mutual exclusive tasks are scheduled together with an arbitrary number of tasks that are not mutual exclusive.



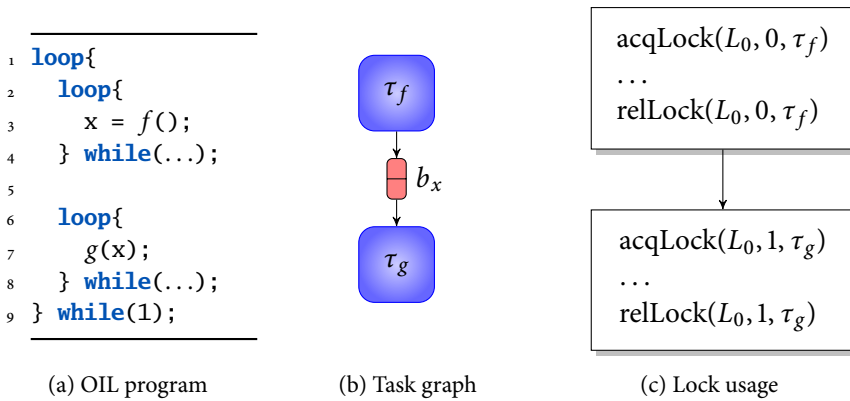


Figure 3.5: Mutual exclusion of while-loops

3.5 REAL-TIME LOCK IMPLEMENTATION

This section describes our realization of the lock and the code generation done by our automatic parallelization tool. Furthermore, it presents the proof that the lock insertion method described in this section does not introduce deadlock.

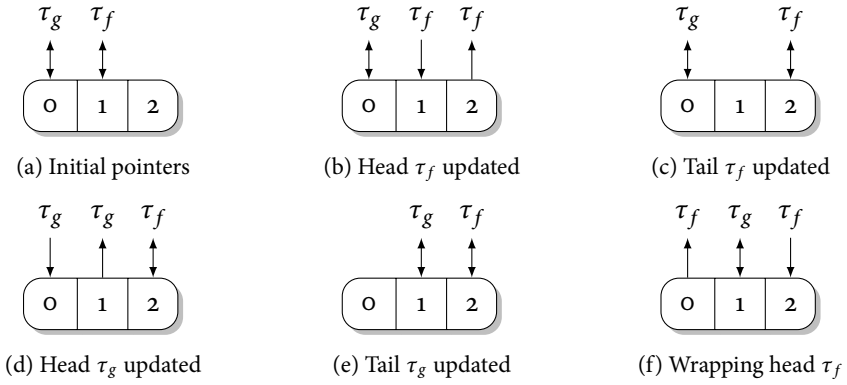
3.5.1 REALIZATION


We first describe the realization of a lock that is suitable for the most basic case which is when two tasks are made mutually exclusive. We then extend this realization for an arbitrary number of tasks. Finally, we explain an additional generalization to make the lock suitable for mutual exclusive execution of groups of tasks.

An OIL program with two modes in which each mode consists of one function is shown in Figure 3.5a. In this program the function f is executed for an unknown number of times after which function g is executed for an unknown number of times. This is repeated forever. The loop conditions are left implicit in order to simplify the example. This program is converted into the parallel task-graph shown in Figure 3.5b, where task τ_f is extracted from function f . Buffer b_x is extracted from variable x .

The realization of the lock is inspired by the implementation of circular buffers [BBJSo8, BBS11]. These buffers can be implemented with ordinary load and store operations instead of atomic read-modify-write operations which is needed to make them starvation-free. These atomic read-modify-write operations are not needed because each shared variable that is used in the buffer implementation is updated by only one task.

Similar to a circular buffer, the lock can be used by a task by calling two functions: *acqLock*, and *relLock*. This is illustrated in Figure 3.5c. The arguments to these



 Figure 3.6: Head and tail pointer updates for a lock consisting of two tasks

functions are a lock identifier, the execution order and a reference to the task. For every lock it holds that tasks using the same execution order argument can execute simultaneously while tasks with a different execution order argument execute in the order indicated by this argument.

The *implementation of the lock* consists of a head and a tail pointer for each task using the lock. The head pointer of a task is incremented during an *acqLock* call and the tail pointer is incremented during a *relLock* call. When the pointers reach the end of the array they wrap around to the beginning of the array. Before a head pointer can be updated to a next location it must be verified that no tail pointer of an other task points to that location, i.e. the *acqLock* call blocks until this is the case. If the function *acqLock* blocks, a yield call is executed indicating to the scheduler that the next task can now use its budget by resuming its execution. This is summarized in the following pointer rules for the lock:

1. Head pointer can only move to the next location if there is no tail pointer with a different execution order, and blocks otherwise.
2. First the head pointer of a task is moved, then the corresponding tail pointer.
3. Pointers are initialized in the reverse of the execution order, with one initially location at the end without pointers.
4. Pointers wrap around.

For the two tasks τ_f and τ_g in the example from Figure 3.5 an array with three elements is allocated. One for each task and an empty location. The pointers are initialized as shown in Figure 3.6a. The head pointer is visualized as pointing upwards and the tail pointer as pointing downwards. Initially the head pointer of τ_f is the only pointer that can be updated without violating the rule that the head pointer may not point to the same array element as the tail pointer of an other task. Thus the *acqLock* call of τ_f updates the head pointer of τ_f , as shown in Figure 3.6b. After

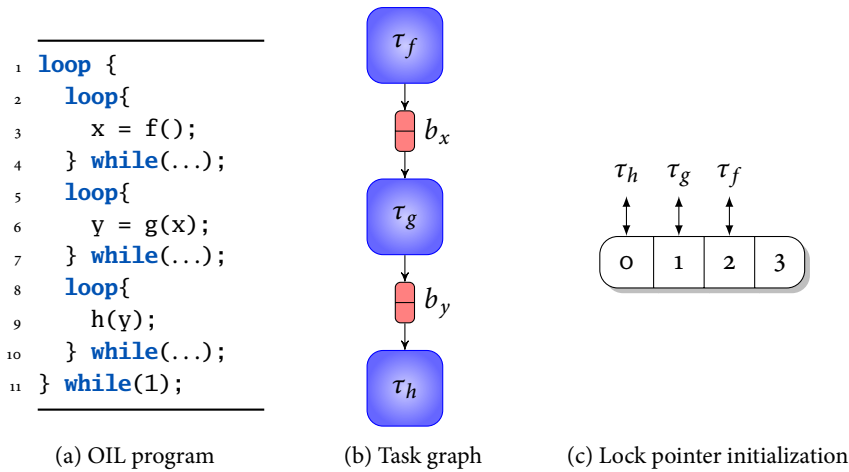


Figure 3.7: Example containing three modes

the *relLock* call of τ_f also the tail pointer is incremented as shown in Figure 3.6c. From this figure we can now conclude that only the head pointer of τ_g can be incremented, see Figure 3.6d. After function g is executed its *relLock* call will update the tail pointer of τ_g as shown in Figure 3.6e. At this point only the head pointer of τ_f can be updated after which this pointer will wrap around to the begin of the array as shown in Figure 3.6f. These steps can be repeated forever.

The in the previous paragraph described lock can be made suitable for n tasks in a relatively straightforward way. This lock requires an array with $n + 1$ locations. Figure 3.7a shows such a program containing three modes where all details about variables are left out for clarity. The corresponding task graph consists of three tasks and is shown in Figure 3.7b. The initialization of the pointers of the lock is depicted in Figure 3.7c. The conditions under which the pointers are allowed to be incremented remains as described in the previous paragraph.

This lock can be generalized to support groups of tasks which execute mutually exclusive with other groups of tasks. An example of an OIL program in which such mutual exclusion can be exploited is shown in Figure 3.8a. In the OIL program there are two modes but now with two functions f and g in the first mode and one function, h , in the second mode. We construct a task graph from it as shown in Figure 3.8b. Figure 3.8c shows that the tasks derived from the functions f and g can execute simultaneously. Here, the execution order parameter of these tasks is both zero, indicating there is no constraint between the tasks. The main modification of the lock is that the pointers of the tasks that belong to the same group, point initially to the same location and these pointers can be incremented independently of the position of the other pointers that belong to the same group.

Since there are two groups in this example, an array of three elements is created,

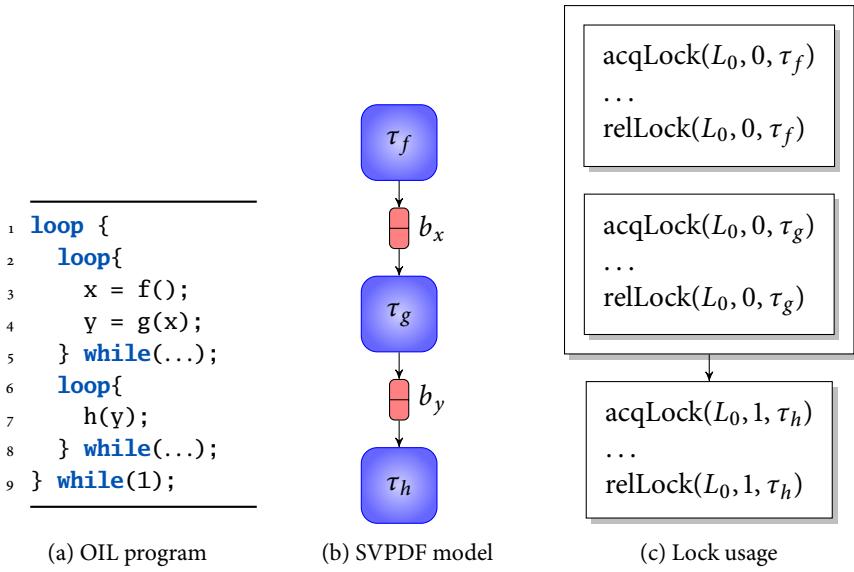


Figure 3.8: Example usage of the lock having groups of tasks

one element for each group and a free one. The initial locations for the pointers are set according to the defined rules and are shown in Figure 3.9a. Tasks τ_f and τ_g both belong to the same group and can both move their head. Task τ_g performs this movement first as shown in Figure 3.9b. Now there are two options; the head of τ_f or tail of τ_g . The first option is shown in Figure 3.9c. Now both tasks can only update their tail in an arbitrary order for example first τ_g and then τ_h as shown in Figure 3.9d and Figure 3.9e. At this point the only possible movement is the head of τ_h as illustrated in Figure 3.9f. This sequence of head and tail updates can be repeated indefinitely.

3.5.2 CODE GENERATION

We now show how such a lock can be used by an automatic parallelization tool such that tasks execute mutually exclusive. The code between the calls to the functions *acqLock* and *relLock* executes mutually exclusive as dictated by the rules outlined in the previous section.

For the simple example with two functions in two modes as shown in Figure 3.5, the implementation of the extracted tasks is shown in Figure 3.10. The *acqLock* function is the first statement in the outer while-loop and the *relLock* function is the last function such that the first inner loop is mutually exclusive with the second inner loop. The execution order argument is derived from the order of the statements in the sequential OIL program. Because *f* is before *g*, task τ_f has number zero and τ_g number one. Basically, an intra-iteration dependency is added between

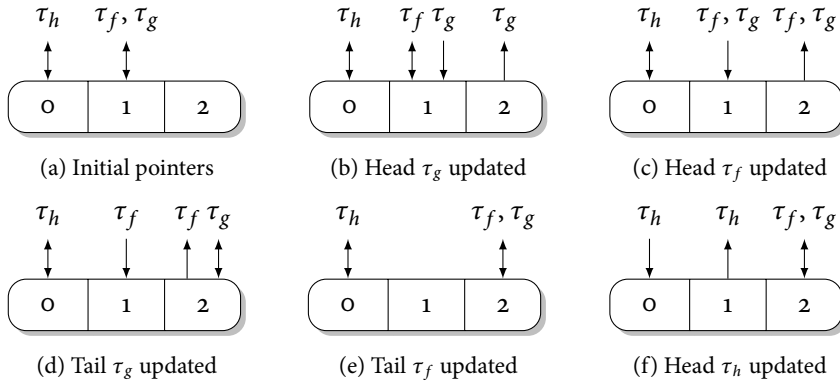


Figure 3.9: Head and tail pointer updates for a lock consisting of two groups of tasks

```

1 do{
2   acqLock(L0, 0, τf);
3   acqProd(bx);
4   do{
5     write(bx, f());
6   } while(...);
7   relProd(bx);
8   relLock(L0, 0, τf)
9 } while(1);

```

(a) Generated code of τ_f

```

1 do{
2   acqLock(L0, 1, τg);
3   acqCons(bx);
4   do{
5     g(read(bx));
6   } while(...);
7   relCons(bx);
8   relLock(L0, 1, τg);
9 } while(1);

```

(b) Generated code of τ_g

Figure 3.10: Tasks resulting from Figure 3.5

f and g . If the lock wraps back to execution order number zero, an inter-iteration dependency is added.

A similar generation of code can be done for if-statements. In Figure 3.11a a simple OIL program with an if-statement is shown and the obtained task graph after parallelization is shown in Figure 3.11b. Both tasks τ_g and τ_h read from a buffer b_x . The order from the sequential program is visualized in Figure 3.11c. Because the if-else-statement has two branches, the two possible orders can occur as shown.

Statements in different branches of an if-else-statement must be in one group if they use the same lock. This is because deadlock can occur otherwise, because there is no sequential order defined between statements in two branches. However, statements in the if-branch are already intra-iteration mutually exclusive with statements in the else-branch. Statements in one branch can be made mutually exclusive using a second lock.

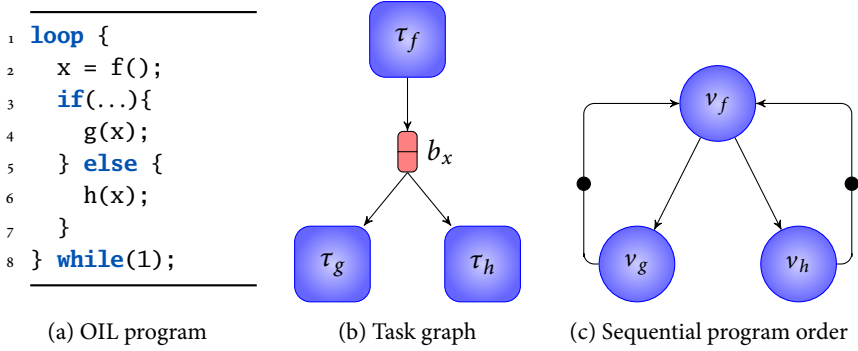


Figure 3.11: Mutual exclusion in a model application described by a conditional-statement

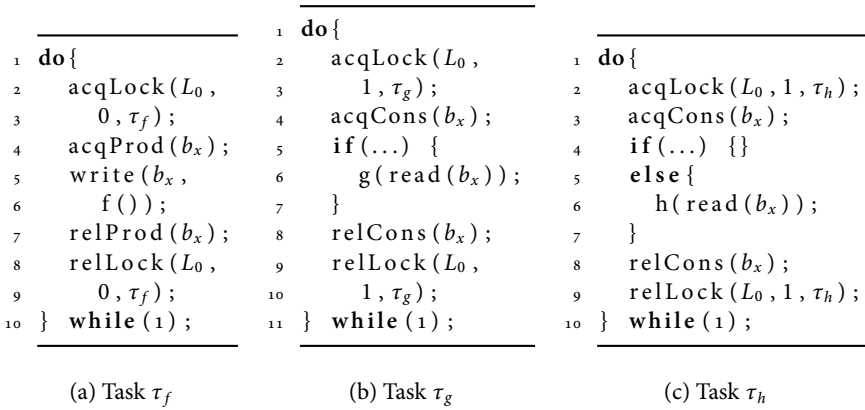


Figure 3.12: Generated tasks from Figure 3.11a including a lock

Assume that we indicate that functions g and h should execute mutually exclusive with function f . Note here that g and h are already intra-iteration mutually exclusive, but not inter-iteration and thus an overlap in execution can occur as a result of pipelining. After adding these functions as one group in a lock they also execute inter-iteration mutually exclusive. After adding the lock, the implementation of the tasks becomes as shown in Figure 3.12. The *acqLock* call in these tasks is again the first statement in the while-loop and the *relLock* is the last statement. This ensures again that the entire loop body is mutually exclusive with the loop body of other tasks. Note here that also the acquire and release functions for the buffers are placed around the if-statement. This enables the derivation of an SVPDF model and guarantees a deadlock-free execution when no mutual exclusivity locks are used.

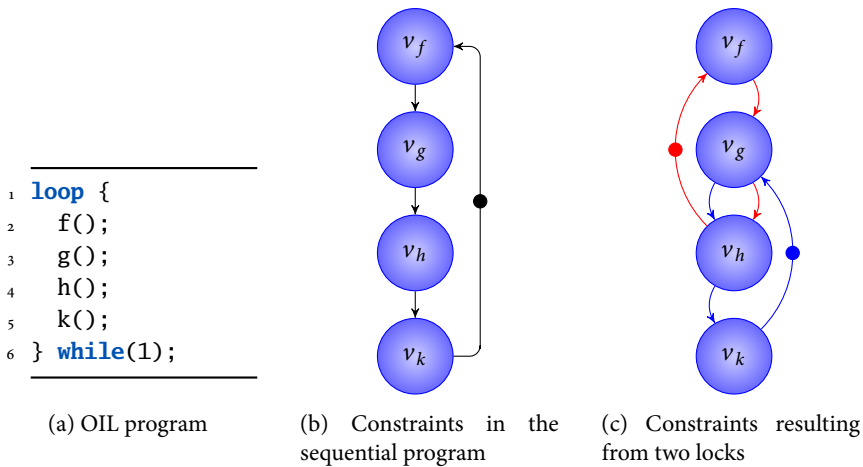


Figure 3.13: Sequential program consisting of four functions. Ordering is enforced in the model by partially preserving the sequential ordering

3.5.3 DEADLOCK-FREEDOM

In this section we explain in more detail why the insertion of the acquire and release calls for the locks will not introduce deadlock.

Deadlock-freedom of task graphs resulting from OIL program is explained using the program shown in Figure 3.11a. The ordering constraints resulting from the sequential program are shown in Figure 3.13b. Assume that we require that the tasks that result from the functions f , g , and h should execute mutually exclusive as well as that the tasks that result from the functions g , h and k should execute mutually exclusive. To achieve this we make use of two locks. The first lock enforces the execution order f , g , h , and then f again in the next iteration. This is modeled by the red edges in Figure 3.13c. The second lock enforces the execution order g , h , k and then g again in the next iteration. This is modeled by the blue edges in Figure 3.13c. Each time we refer to the next iteration an initial token is placed on the corresponding edge.

The reasoning why the resulting task graph will remain deadlock-free after locks are added can be seen as follows. When a number of groups of statements in a loop is made mutual exclusive, intra-iteration dependencies are added between these groups. As presented in Section 3.5, these intra-iteration dependencies are added in such a way that they follow the order of the statements in the sequential specification. These added intra-iteration dependencies can thus never make the sequential schedule inadmissible. In Figure 3.13c dependencies are added for example from function f to function g and from function g to function k . The sequential schedule of Figure 3.13b shows that function g is scheduled after f and function k after g . Therefore, after adding the constraints of the lock, the sequential schedule is still

admissible.

Moreover, when making groups of statements mutual exclusive, an inter-iteration dependency is added from the last group back to the first group. In the sequential schedule we have that the first statement of a loop executes after the last statement of the previous iteration of that loop. A group of statements can by definition never contain a statement that occurs earlier or later in the sequential specification than this first or last statement respectively. Adding an inter-iteration dependency between the last mutual exclusive group to the first group can thus never invalidate the sequential schedule. Consider for example the dependency in Figure 3.13c that prescribes that function f in iteration $i + 1$ can only be executed after function h in iteration i is finished. This dependency does not invalidate the sequential schedule because in this schedule, function h in iteration i executes before function k in iteration i which on its turn executes before function f in iteration $i + 1$.

3.6 SVPDF MODEL

In this section, we present the SVPDF model, which can be automatically derived from an OIL program. The SVPDF model is based on the VPDF model but contains additional structure in the form of hierarchical blocks to enable efficient analysis. These blocks can be seen as `do-while` loops that iterate for an unknown number of times. In Section 3.7, we will model the constraints that result from locks in this SVPDF model.

In this chapter we only consider the derivation of the SVPDF model for scalar variables. However, actors in this model can be extended with phases indicating a sequence of how many tokens need to be consumed or produced every firing of an actor [GHB14]. This number of tokens is based on the synchronization done by tasks. The modeling of mutual exclusivity can be done in a similar way as described in this chapter for this more expressive model. For ease of understanding we therefore omit the phase information from the model.

An SVPDF model is a directed graph $G = (V, E, \mathcal{P}, \delta, \rho)$ that consists of a set of actors V connected by a set of directed edges E . An actor $v_i \in V$ communicates to another actor v_j by producing tokens on an edge $e_{ij} \in E$. Actors in an SVPDF model are not auto-concurrent, meaning that at most one firing of an actor can occur simultaneously. The number of initial tokens on an edge is given by $\delta : E \rightarrow \mathbb{N}_0$. In short, δ_{ij} , are the number of initial tokens on e_{ij} . An actor v_i is enabled to fire if there are tokens on all its incoming edges. After its firing duration $\rho_i : V \rightarrow \mathbb{N}_0$, a token is produced on all its outgoing edges. The maximum number of tokens on a cycle remains constant and is for example used to model the capacity of a FIFO buffer.

An SVPDF model is structured into blocks with port actors on the edges of a block. A block is characterized by a parameter $p \in \mathcal{P}$. A parameter p defines the number of consecutive firings of actors in that block in respect to the actors surrounding that block. The value of p is unknown during analysis and can be infinite. A

block only introduces structure and does not fire itself. Port actors are used to provide communication between actors in and outside of a block. A port actor either converts a token on an input edge directed from outside of a block inwards to p tokens on its output edges or it converts p tokens to one token if the direction of the edges is from inside a block to outside of that block. A more detailed explanation of the SVPDF model and port actors can be found in [GHB13].

The blocks are used to describe modal behavior of an application, where it is unknown how many iterations an application remain in a certain mode. Tasks in these modes can interact with the environment via periodic sources and sinks. Only tasks in the active mode can read from a source or write to a sink, such that every sample produced by a source is only read in a single mode. Therefore an actor derived from a source or sink is copied into every block in which it is used. Such an actor only produces or consumes tokens if the mode corresponding to the block is active. These sources and sinks execute periodically and therefore impose a throughput constraint on the execution of the application.

3.7 LOCK FROM SEQUENTIAL SPECIFICATION

In this section it will be shown that a corresponding SVPDF temporal analysis model can always be derived from a sequential OIL program in which the locks are specified. We first derive an SVPDF model from an OIL program without considering the locks. This is followed by the modeling of the constraints that result from the locks. In this section the modeling approach is shown for one lock, but the modeling of multiple locks is analogous to the modeling of one lock.

A task graph without mutual exclusive tasks can be modeled as an SVPDF model as follows. For every task an actor is included in the model. Every buffer is modeled by two oppositely directed edges where tokens on the edge from the producer to the consumer represent the full locations in the buffer and tokens on the other edge, empty locations. The initial tokens on this edge equals the buffer capacity. For every while-loop in the OIL program a block is included in the model. If a task contains this loop, the actor corresponding with this task is a (nested) child of the block corresponding with this loop. Blocks are nested analogously to the structure of while-loops in the OIL program. The value of the parameter characterizing a block corresponds to the number of iterations of the while-loop.

We now show that the lock as generated by the method introduced in this chapter can be modeled in an SVPDF model. Two cases can be distinguished in the generation of the lock, either groups of tasks in one while-loop are made mutually exclusive, or these groups are distributed over multiple while-loops.

We first consider the most simple case of two groups of tasks in one while-loop, where each of the groups consists of only one task. When tasks are made mutually exclusive this can be represented by a cycle through the corresponding actors with one initial token on that cycle. This is illustrated by Figure 3.14a which contains two actors corresponding with two mutually exclusive tasks. Between these two

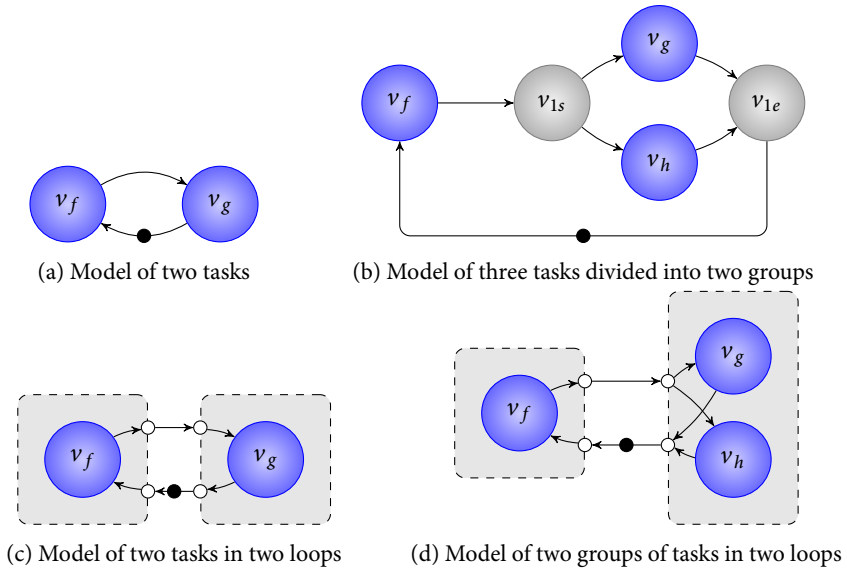


Figure 3.14: SVPDF models corresponding with mutually exclusive tasks

actors a cycle is added consisting of two oppositely directed edges representing the ordering constraints enforced by the lock. The initial token on the bottom edge indicates the task that is enabled first by the lock, i.e. the task corresponding with the first function in the sequential ordering specified by the OIL program.

When there are multiple tasks in a group no single cycle can be created anymore in the model because tasks in a group can execute simultaneously. Therefore, for every group consisting of more than one task two additional actors are added to the model. The first actor represents that all tasks in a group can start after the tasks in the previous group have finished. The second actor represents that the next group can only start after all actors in the current group have finished their firing. This is illustrated in Figure 3.14b in which there are two groups of tasks. The first group contains only one task, and thus no additional actors are required. The second group contains two tasks and thus two corresponding actors. Two additional actors are now added. Actor v_{1s} allows both actors v_g and v_h to start because separate edges (v_{1s}, v_g) and (v_{1s}, v_h) are added. Actor v_{1e} enforces that both actors are finished before actors in the next group can fire, which is again actor v_f in the figure. This is modeled by an edge from every actor in the group to actor v_{1e} .

When groups of tasks belonging to different while-loops, the corresponding actors are in different blocks. The approach described above must then be modified such that the correct rate conversion occurs by means of port actors. A port actor is added whenever a constraint following from the lock generation crosses a while-

loop boundary and thus results in a rate-conversion. Figure 3.14c shows an example of two groups of which the corresponding tasks are in different while-loops. In the example a group consists of one task, and thus one actor. In the figure the cycle from actor v_f to v_g and back crosses four block boundaries and thus four port actors are added. These port actors model that a task in the corresponding while-loop can execute until the loop condition becomes false. The initial token that indicates which actor can initially start is placed before the port actor which is located on the outer-most block corresponding with the first while-loop in the ordering defined by the sequential specification. In the figure this token is placed before the port actor on the top in the left block, assuming that the left block corresponds to the first while-loop.

Finally, groups of tasks in multiple while-loops can also contain multiple tasks per group. This is illustrated in Figure 3.14d where the second group contains two tasks. Here the port actors are also used to indicate the simultaneous enabling of tasks in a group and the waiting until all tasks in a group are finished. Thus, these port actors are used instead of the two additional actors inserted for the case shown in Figure 3.14b.

3.8 CASE STUDY

In this section we illustrate the approach presented in this chapter by means of a simplified WLAN 802.11g receiver application of which the OIL program is shown in Figure 3.15. In the application, first a header must be detected by the *detectHeader* function in an input stream delivered by a source ADC executing time-triggered at 250 kHz. After a header is found it is decoded by the *decodeHeader* function and then *NSym* symbols in the received packet are decoded by the functions in the second inner while-loop. In the second loop, first a symbol is transformed by an *fft* into the frequency domain. The resulting data is then demapped, deinterleaved and convolutional decoded. Finally, a cyclic redundancy check (CRC) is performed to verify the correctness of the resulting data.

From this WLAN application a task graph is extracted by the compiler with seven tasks and seven buffers. The periodic source imposes a throughput constraint of 250 kHz which corresponds to a period of 4 μ s.

In this example we assume that the *detectHeader*, *decodeHeader* and *fft* task execute on the same processor and a WCET of 3 μ s, 1 μ s, and 3 μ s, respectively. If we allocate a budget of 0.5 μ s in a replenishment interval of 1.5 μ s for these tasks then it follows from Equation 3.1 that the WCRT of the *detectHeader* task is equal to 9 μ s. Because the period of the source is 4 μ s we can immediately conclude that the throughput constraint cannot be met.


By inserting a lock we can reduce the WCRT of the *detectHeader*, *decodeHeader* and the *fft* task. We can indicate in an OIL program by means of the *lock* keyword that tasks must execute mutually exclusive, as shown in Figure 3.15. The parameters behind this keyword are groups of tasks and each group of tasks is encapsulated by



```

1 lock (detectHeader) (decodeHeader);
2 lock (detectHeader decodeHeader) (fft);
3
4 source ADC @ 250 kHz;
5
6 loop{
7   loop{
8     detectHeader(ADC, out vh, out h);
9     if(vh){
10      NSym' = decodeHeader(h);
11    }
12  } while(!vh);
13  n = 0;
14  loop{
15    x = fft(ADC);
16    y = demap(x);
17    z = deint(y);
18    w = convDecode(z);
19    crc(w);
20    n' = n + 1;
21  } while(n < NSym);
22 } while(1);

```

 Figure 3.15: Simplified WLAN application

brackets. In the WLAN specification in Figure 3.15 there are two groups of tasks behind the first *lock* keyword. The first group contains the *detectHeader* task and the second group the *decodeHeader* task. As a result of the lock only one of these tasks execute at any point in time on the processor and their execution order will be the order from the OIL program. Behind the second *lock* keyword there are two groups of tasks with in the first group the *detectHeader* task and *decodeHeader* task and in the second group the *fft* task. As a result these three tasks obtain a WCRT equal to their WCET. These WCRTs are low enough to meet the throughput constraint given that the buffers are sized properly by making use of the SVPDF model of the application.

The SVPDF model of the WLAN application is shown in Figure 3.16. The source is not shown for clarity of the figure. In this figure the red dashed arrows denote the edges added to enforce mutual exclusivity between the *detectHeader*, *decodeHeader* and the *fft* tasks and between the *detectHeader* and *decodeHeader* tasks. The remaining tasks all run on a separate processor and therefore have a WCRT equal to their WCET which are, in application order starting with *demap*, 2.5 μ s, 3 μ s, 2.5 μ s and 4 μ s for *crc*. With this SVPDF model, buffer sizes are computed for δ_h , δ_{hv} , δ_x , δ_y , δ_z , δ_w , δ_{NSym} of 1, 1, 2, 2, 2, 2, and 1, respectively.

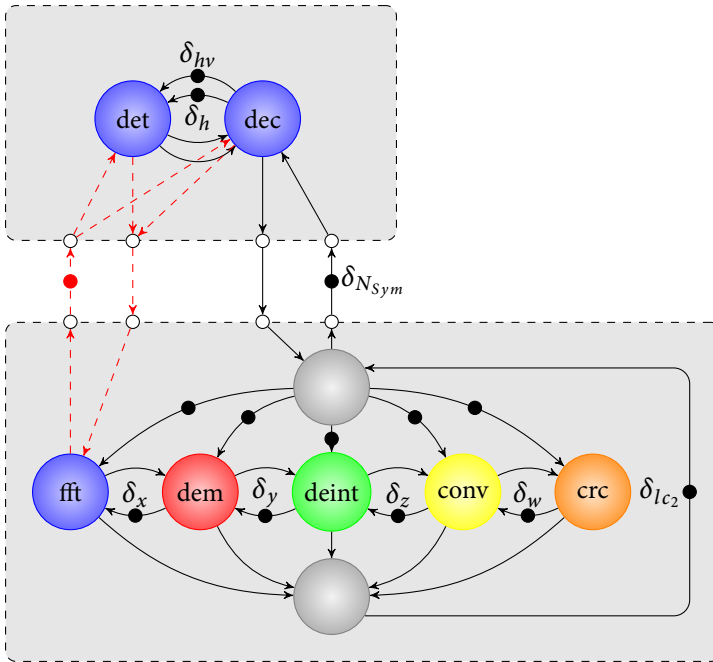


Figure 3.16: SVPDF model of the application in Figure 3.15

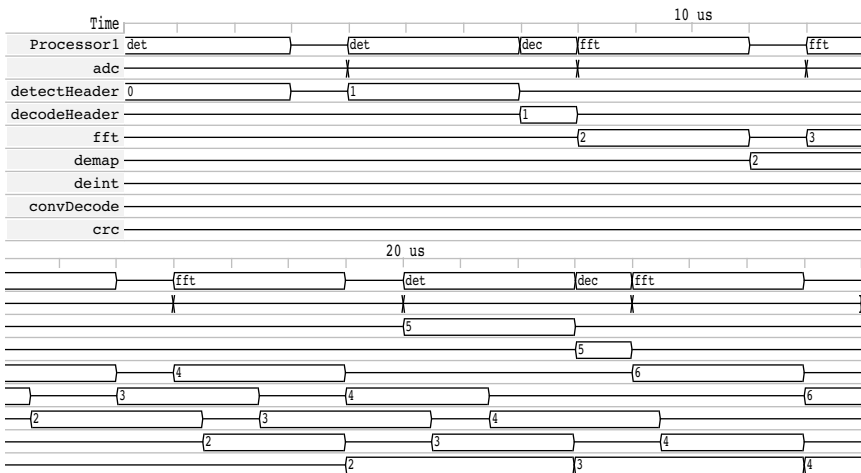


Figure 3.17: Execution trace of the WLAN application from Figure 3.16 with mutual exclusivity applied

Figure 3.17 shows an execution trace derived with the dataflow simulator HAPI [BPvMo5, KHB16b] given the WCRTs when mutual exclusive execution is enforced. The num-



bers in the traces indicate the invocation number of the tasks. The trace is shown for a packet size of 3 symbols. As a result of the applied locks the *detectHeader*, *decodeHeader*, and *fft* task can execute on the same processor as is visualized in the trace for *Processor1*, instead of that 3 processors are required. This improves the utilization of one processor and frees two other processors. The execution of the application is pipelined and tasks belonging to different modes execute on different processors at the same point in time despite that locks are applied. The trace shows for example that *detectHeader* and *deint* execute in parallel.

3.9 CONCLUSION

In this chapter we presented a dataflow analysis approach that takes into account that tasks execute mutually exclusively which improved the temporal analysis results and processor utilization. We furthermore introduced a starvation-free lock which allows us to enforce mutual exclusive execution of tasks. This lock allows parallel execution of tasks in a group of tasks but enforces sequential execution between groups of tasks. A key difference with existing locks is that groups of tasks can only acquire the lock in a predefined order.

We furthermore showed that mutual exclusive execution of tasks can be modeled in an SVPDF model which is used for checking whether the throughput constraint is satisfied after adding locks. This model is generated by a compiler from a sequential OIL program that describes the modal real-time stream processing application.

We also showed that the resulting parallel task graph is deadlock-free despite that additional constraints are introduced that enforce an execution order of groups of tasks as a result of the locks. The task graph is deadlock-free because lock statements are added such that no constraints are introduced that prevent the execution order as defined by the sequential program.

The introduction of our lock in an application can improve the processor utilization as is demonstrated using a WLAN application. In this application two locks are introduced. Insertion of these locks reduced the worst-case response times such that three tasks can share the same processor which improves the utilization of this processor and frees two other processors.

COMPOSITIONAL ANALYSIS OF MODES AND FPP SCHEDULING

ABSTRACT – The temporal analysis of modal applications where tasks are scheduled using the FPP scheduling policy leads to very pessimistic results. Therefore, we propose a compositional analysis approach in this chapter, that allows modes to be characterized in isolation, even at different levels in the hierarchy of an application. Locks and barriers are added in an application such that the temporal behavior of modes can be characterized independently. Existing analysis method can then be used within these modes.

Stream processing applications such as software defined radios and vision applications are typically executed on embedded multiprocessor systems under real-time constraints. These applications often contain multiple processing modes and cyclic dependencies. Examples of processing modes found in software defined radios are the detection, synchronization and decoding mode. The cyclic dependencies are a result of feedback loops and the use of bounded FIFO buffers for inter-task communication. These stream processing applications can be described as task graphs of which the tasks are executed on shared processors. The tasks executed on a processor are scheduled according to a scheduling policy. In Chapter 3, we only addressed the analysis of modal applications with budget schedulers. Examples of other scheduling policies are RR and FPP. Currently, however, there is no suitable temporal analysis technique available for modal applications that supports this broader class of scheduling policies.

This chapter is based on [GK:3].



The minimum throughput of a modal stream processing application can be determined using dataflow analysis techniques given a VPDF [WBS10] or a Finite State Machine-based Scenario-Aware Data-Flow (FSM-SADF) [GS10] model if budget schedulers are applied [WBS07a, WBS09]. An example of a budget scheduler is Time Division Multiplex (TDM). For these budget schedulers it is possible to derive the worst-case response time of each task independently of the schedule of the other tasks, after which a compositional temporal analysis method can be applied.

However, many embedded operating systems only support the FPP scheduling policy. Dataflow analysis techniques for systems that use FPP have recently been introduced for multi-rate applications that can be modeled as SDF graphs [HGWB14]. However, there is currently no temporal analysis technique for modal stream processing applications that contain cycles and that are executed on multiprocessor systems which make use of FPP task scheduling.

In this chapter, we present a compositional temporal analysis approach for modal stream processing applications executed on multiprocessor systems using FPP task scheduling per processor. A compiler inserts locks and barriers in the application such that the temporal behavior of each application mode can be characterized in isolation. The locks ensure that tasks belonging to different modes do not interfere, and the barriers make the response times of the tasks independent of the production moments of tasks that belong to other modes. The locks and barriers result in additional constraints that are included in a hierarchical SVPDF model of the application. This model is used to verify the satisfaction of the throughput constraint and to compute the required buffer sizes by recursively applying a recently introduced dataflow analysis technique. Furthermore, it is shown that the approach supports response times of tasks larger than the period of the source, and allows the use of budget scheduling besides FPP scheduling. The applicability of the approach is demonstrated using a IEEE 802.11p (WLANp) application. This application can be executed in a pipelined fashion despite the additional constraints introduced by the locks and barriers.

The outline of this chapter is as follows. We first discuss related work in Section 4.1. The basic idea behind our analysis approach is presented in Section 4.2. Section 4.3 describes the analysis flow we use to analyze modal applications. In order to be able to verify if a periodic source can execute strictly periodically when switching between modes, additional constraints are introduced into the SVPDF model as described in Section 4.4. The response time equations that are introduced in Section 4.5 can be used to apply the presented analysis approach to systems in which also other scheduling policies are applied than FPP. Conditions under which response times larger than the source period are allowed are stated in Section 4.6. The applicability of the analysis approach is demonstrated in Section 4.7. The conclusions are stated in Section 4.8.

4.1 RELATED WORK

In this section we first address related analysis approached for modal time-triggered systems. Then we switch to data-driven approaches and focus on modal dataflow models. Analysis of FPP scheduling for dataflow graphs is discussed and finally analysis of budget scheduling for modal dataflow graphs.

Mode changes in FPP scheduled single processor systems have been studied extensively [S⁺89, TBW92a, Gua09b]. These works present analysis techniques to determine whether deadline misses can occur during a mode change. Overloads can be prevented by the schedulers by delaying the release of tasks [RCo4a]. A limitation of these techniques is that the next mode transition may only be started after the previous mode transition is completed. Furthermore, these techniques are only applicable for acyclic task graphs.

Dataflow models are applicable for cyclic task graphs, and a number of dynamic dataflow models have been developed that are also suitable for modal stream processing applications. In the FSM-SADF [GS10, vKSG17] dataflow model modes are described as scenarios where the possible transitions between scenarios are encoded in a non-deterministic finite-state-machine. In [vKSG17], the FSM-SADF model is extended to allow switching scenarios at run-time by executing sequences of scenarios. Processor sharing can be supported if schedulers are applied that belong to the class of budget schedulers to which TDM schedulers belong.

In this chapter, we use the SVPDF [GHB14] model that allows a hierarchical description of nested modes in an application. The dependencies are explicit in this model and it has been shown that a deadlock-free model and implementation can be automatically derived from a sequential description of an application in the Omphale Input Language (OIL) [GHB13]. The SVPDF model has only been used in combination with budget schedulers. In this chapter, however, we will show that the SVPDF model can be used if FPP schedulers are applied.

Throughput analysis of systems with multiple applications that are modeled as Mode-Controlled Dataflow (MCDF) and scheduled by FPP has been presented in [LMvB15]. However, this approach only addresses interference between tasks that belong to different applications, while in this chapter we derive the interference between tasks of the same application.

Only recently, a throughput analysis has been introduced in [HWGB13] for multi-rate applications that are modeled by SDF graphs and that are executed on multiprocessor systems using FPP schedulers. By introducing an enabling rate characterization in [HWGB14], the accuracy of the analysis technique is improved. The analysis flow based on the enabling rate characterization is further improved in [WHGB14] by taking into account that cyclic dependencies limit the maximum interference between tasks. We make use of this observation in this chapter, since we introduce cycles to make the execution of tasks mutually exclusive.

Locks have been introduced in Chapter 3 for modal applications that make tasks

scheduled by budget schedulers mutually exclusive, which can improve the accuracy of dataflow analysis. In this chapter we generalize that approach for budget schedulers, making it applicable for systems with FPP schedulers.

4.2 BASIC IDEA

In this section, we use a didactic example to explain the basic idea behind our compositional temporal analysis approach. This approach is suitable for modal stream processing applications that are executed on multiprocessor systems using FPP task scheduling per processor. The periodic source in these applications imposes a throughput constraint.

The task graph used in our didactic example is derived by a multiprocessor compiler from the OIL program that is shown in Figure 4.1a. An OIL program is a kind of C-program with some adaptations that facilitate automatic parallelization. The OIL program contains two potentially endlessly iterating while-loops, where each while-loop corresponds to a mode. After parallelization, the task graph in Figure 4.1b is obtained. Each task in the graph corresponds to a function in the OIL program, e.g. τ_a corresponds to function a . The color of a task represents the mapping to a processor. Each FIFO buffer in the task graph corresponds to a variable that is communicated between two functions.

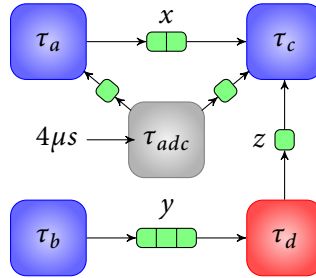
The source task, τ_{adc} , executes strictly periodic and only produces data for a single mode. The source task therefore has two output buffers such that data can be sent to one of the task reading from it, either τ_a or τ_c . The buffer that the source writes to is determined by the loop conditions in the sequential OIL program, which can change after every execution of the source task.

Besides a task graph, also an SVPDF dataflow model is created by our compiler. This model contains nested blocks where each block corresponds to a while-loop in the OIL program, as is described in more detail in Section 3.6. These blocks are depicted as dashed rectangles in the SVPDF model. The actors in a block correspond with the functions inside the while-loop. The SVPDF model that corresponds to our didactic OIL program is shown in Figure 4.1c. The nodes on the boundaries of the dashed rectangles are called port actors. The port actors at the inputs of a block perform up-sampling, i.e. multiply the number of tokens with a factor that is equal to the number of iterations of the while-loop. The port actors at the outputs of a block perform down-sampling. Only after the last iteration of a loop, a token is produced at the output of such a port actor, for other iterations no tokens are produced. Port actors have by definition a firing duration equal to zero. The other actors have a firing duration that corresponds to the response times of the tasks. Derivation of these response times is discussed in subsequent paragraphs. The solid black edges between the actors in the SVPDF model denote dependencies between actor firings. The use of FIFO buffers with a bounded capacity in the task graph results in cyclic dependencies in the SVPDF model. The number of tokens on a cycle in the SVPDF model corresponds to the capacity of a buffer. The reason for the

```

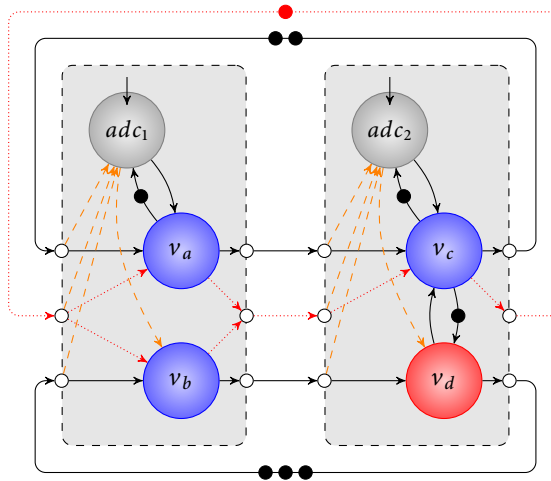
1 source ADC @ 250 kHz;
2 loop{
3   loop{
4     x=a(ADC);
5     y=b();
6   }while(..);
7   loop{
8     z=d(y);
9     c(ADC, x, z);
10  }while(..);
11 } while(1);

```




(a) Modal OIL program

(b) Corresponding task graph



(c) SVPDF model of Figure 4.1a including additional constraints of a lock and barriers

 Figure 4.1: Example of a modal program and the corresponding task graph and dataflow model

inclusion of the other edges in the SVPDF model will be explained in subsequent paragraphs.

The SVPDF model is the input of our compositional temporal analysis method which is described in Section 4.3. This method requires that response times of tasks can be determined independently of when tasks in other modes execute. Given that the response times are independent then the firing durations of the actors in one mode are also independent of the firing durations of the actors in another mode. Because the firing durations are independent we can analyze whether each deepest nested block fulfills the constraints imposed by the source independently of other blocks. After this is confirmed, we can ascend the hierarchy of blocks and perform

the same analysis on the next hierarchical level of the SVPDF model.

The fact that response times of tasks in one mode can be analyzed independently of tasks in another mode is achieved by making use of locks and barriers. Locks prevent that tasks belonging to different modes can execute at the same moment in time by making tasks in different modes, but mapped to the same processor, execute mutually exclusive. Barriers are used to verify that all inputs of a mode are available before the periodic source in a mode finishes its first execution. As a result, the response times can be determined relative to the executions of the periodic source in a block and become independent of the production moments of tasks that belong to other modes.

Dataflow analysis for systems that make use of nonstarvation-free schedulers such as FPP has been described in [HWGB13, HGWB14]. These approaches make use of a slightly modified version of the WCRT equation that has been proposed by Tindell [TBW94]. This response time equation computes a so-called busy period which calculates the maximum time it takes to finish q executions of τ_i and is defined as:

$$w_i(q) = q \cdot B_i + \sum_{j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{P_j} \right\rceil \cdot B_j \quad (4.1)$$

where J_j is the jitter in the so-called *external enabling time* of τ_j , which, on average, executes periodically with period P_j . We use $hp(i)$ as the set of tasks executing on the same processor with a priority higher than τ_i . The *external enabling time* of a task is defined as the time at which the task can read sufficient locations from the adjacent buffers. Only values of q for which it holds that $w_i(q) \geq q \cdot P_i$ need to be considered in Equation 4.1 according to [TBW94].

According to [HWGB13, HGWB14] is the worst-case response time relative to the external enabling time of a task equal to:

$$\hat{R}_i = \max_{1 \leq q} (w_i(q) - (q - 1) \cdot P_i) \quad (4.2)$$

From Equation 4.1 and Equation 4.2 we conclude that the WCRT of τ_i depends on the jitter J_j of the inputs of the higher priority tasks on the same processor. These inputs can be produced by tasks in a different mode. As a result, the WCRT of τ_i cannot be determined independently of the tasks in other modes, because the jitter can be a result of the jitter in the finish times of tasks in other modes. This is even the case if the execution of these tasks has been made mutually exclusive by making use of locks. Therefore, besides the locks, also barriers are needed to make the jitter of tasks in a mode independent of the production moments of tasks that belong to other modes.

Barriers are used to guarantee that none of the tasks in a mode can start before all inputs for that mode are available and the source inside the mode fires. By introducing barriers, actors that are not on a path from the source without initial tokens on it, like ν_d , are delayed. These actors execute time-triggered since they are

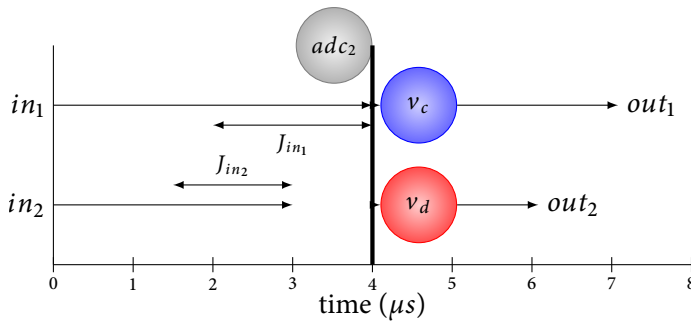


Figure 4.2: Barrier at $t = 4\mu\text{s}$ guarantees that no task belonging to a mode can start before all inputs are available and the source of that mode finishes its firing

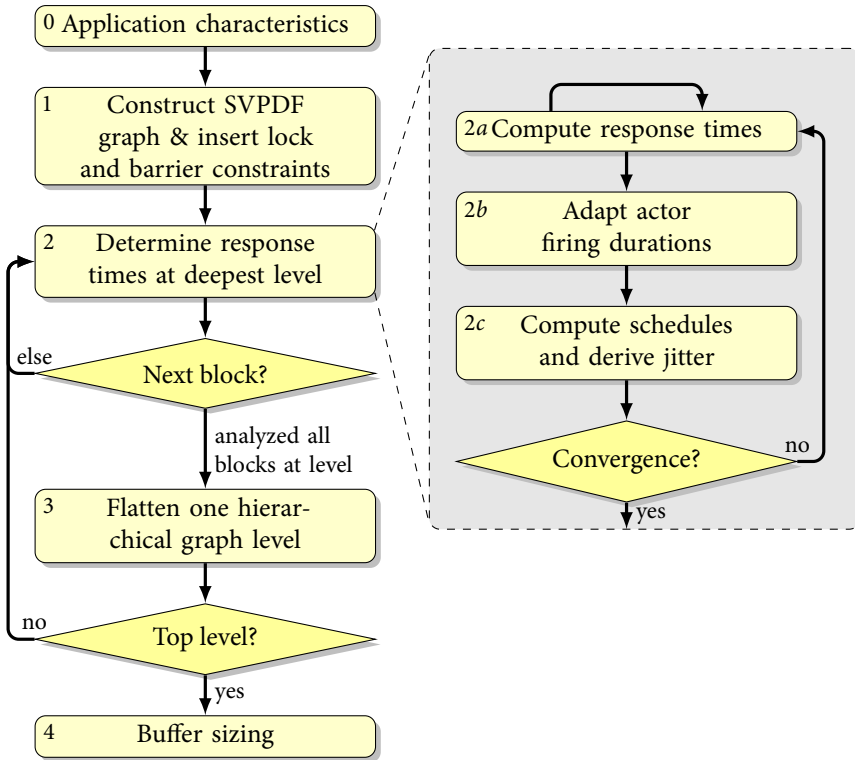
enabled by the periodic source instead of being enabled by an input of the mode they are in. Therefore, the start times of v_c and v_d in Figure 4.2 are only related to the time that the source actor, adc_2 , in the mode fires and not to the arrival times of the input data. As a result, the jitter in the production moments of tasks outside a mode has no influence on the jitter on the enabling time of the tasks that belong to the mode. Therefore response time are also independent of the jitter on the inputs caused by task outside the mode.


A barrier in the implementation is modeled with additional dependencies in the SVPDF model. These dependencies are depicted as orange dashed arrows in Figure 4.1c. Moreover, the locks which make the execution of tasks mutually exclusive are modeled with the red dotted edges in Figure 4.1c. These edges in combination with a single token on the cycles created by these edges guarantee that actors belonging to different blocks, and thus modes, can not fire simultaneously.

4.3 ANALYSIS FLOW

In this section, the applied temporal analysis flow is presented. In this flow, blocks in the SVPDF model are analyzed recursively, starting at the deepest nested blocks in the model, and subsequently one level upwards at a time after flattening the blocks of a hierarchical level. The temporal analysis flow incorporates the flow presented in [WHGB14] for the computation of the worst-case response times of the tasks. We first present an overview of the steps in the analysis flow after which the flattening step in the flow is explained in more detail in Subsection 4.3.1.

The applied analysis flow is shown in Figure 4.3. This flow is used to verify whether the temporal constraint imposed by the periodic source (sink) in the blocks can be satisfied. The periodic source must be able to fire strictly periodic, such that there are always sufficient tokens on all the incoming edges at the start of a new periodic firing. To verify these constraints the worst-case response times of the tasks are computed. The last step in the flow computes sufficiently large buffer capacities.



 Figure 4.3: Overview of the analysis flow for modal applications

The analysis flow in Figure 4.3 contains five steps of which some of the steps are executed repeatedly. Step 2 consists internally of three steps which are executed repeatedly. In step 0, the input information for the analysis flow is gathered. This is the information about the topology of the task graph, the task-to-processor assignment, the applied task scheduling policy for each processor including the scheduling parameters such as priorities, worst- and best-case execution times of the tasks, and the temporal constraints imposed by the periodic source and/or sink inside a block.

Based on the nesting of the functions in an OIL program, an SVPDF model is generated in step 1. Locks and barriers are inserted to enable compositional analysis of blocks as discussed in Section 4.2. As a consequence, the response times of tasks belonging to a mode can be determined by analyzing each mode in isolation.

In step 2a of the flow, a lower and an upper bound on the response time of the tasks in a block at the deepest hierarchical level is computed. This is done under the assumption that the inputs of the block have arrived before the source of the task has finished its execution, which is verified later. These response times are

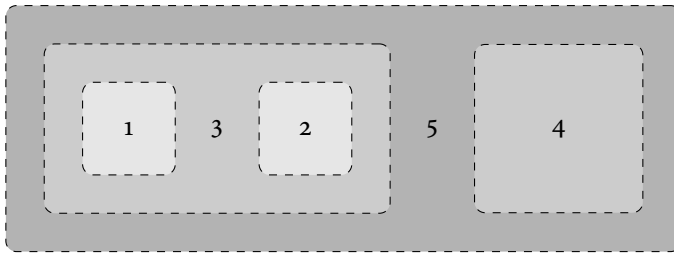


Figure 4.4: An SVPDF model with nested blocks

used to update the minimum and maximum firing durations of the actors in the SVPDF model in step 2b. Two periodic schedules are computed using these firing durations, which bound the start times of the actors. From these start times, the jitter in the production moments of the actors is computed. Given these jitters the response times are recomputed in step 2a, 2b, and 2c, which are repeated until the jitters and response times remain unchanged, or exceed a predefined limit. Step 2 is repeated until all blocks at the deepest hierarchical level have been analyzed.

Once the response times of all the tasks at the deepest hierarchical level have been computed the modes at that level are flattened in step 3. The flattening step is described in more detail in the next section.

Step 2 and 3 are repeated until all levels in the hierarchy of the application are flattened and the top level is reached, or a violation of the temporal constraints is detected. The order in which modes are flattened is explained using Figure 4.4 which shows the nested block structure of an SVPDF model. The different shadings of gray in the figure reflects the hierarchical level of a mode. Blocks in the same hierarchical level have the same shade. In the first iteration of the analysis flow the modes at the deepest level are analyzed, i.e., mode 1 and 2. After these modes are flattened, mode 3 and 4 are analyzed at the next hierarchical level. Finally, mode 5 can be analyzed after all other modes have been flattened.

In the final step of the flow, the buffer capacities are determined, given the computed best-case and worst-case schedules for the actors in each of the blocks.

4.3.1 FLATTENING OF A HIERARCHICAL LEVEL

The flattening step of the analysis flow removes the hierarchical boundaries of the blocks of one hierarchical level in an SVPDF model. This step is performed after the response times of the actors in the blocks at one hierarchical level have been derived using the analysis flow in Figure 4.3. In the remainder of this section we will first describe how a hierarchical level can be flattened. Next, we introduce the constraints on the flattening step imposed by the periodic source. Finally, we will discuss the consequences of a flattened block on the jitter and response time of tasks at a higher hierarchical level.

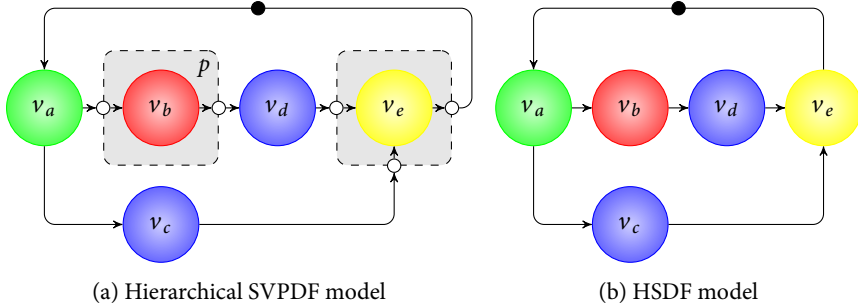


Figure 4.5: Flattening an example of an SVPDF model into a HSDF model

The WCRT of tasks can be calculated under the assumption that tasks in a block execute an infinite number of times. This is possible because the WCRT equation as stated in Equation 4.2 assumes an infinite number of executions of strictly periodically executing tasks. However, tasks in a block do not execute after a switch to a different block. Therefore the tasks do not execute infinitely often. However, tasks that do not execute, do not cause interference, which reduces the response time. The actual number of times a task executes depends on the value of a so-called parameter [GHB13] of a block, which indicates how often a block will execute. The parameter can range from one to infinite and is only a modeling construct without a counterpart in reality. Therefore an upper bound on the actual response time of the tasks in a block can be calculated by assuming that tasks in a block execute infinitely often.

The structure in an SVPDF model allows blocks to be flattened. A block is flattened by setting the response times of tasks inside the block to the WCRT calculated under the worst-case assumption that tasks in a block execute infinitely often. Therefore, the WCRT is valid for every possible parameter value. As a result, only a single iteration of the block needs to be considered at higher hierarchical levels. In the SVPDF model a single iteration of a block is equal to a parameter value of one of the corresponding blocks. A parameter value of one flattens the block, since the boundaries of a block and the port actors become redundant because up- or down-sampling of tokens is not needed anymore. Therefore, the SVPDF model in Figure 4.5a can be flattened into the HSDF model shown in Figure 4.5b. The flattened model can be analyzed using the approach presented in [WHGB14], which is applied in step 2 of our flow.

A flattened block should still adhere to the constraints imposed by the periodic source inside the block. The constraints are a result of the source task in an application that must be able to execute strictly periodically independent of the block that is active. Two constraints must therefore be satisfied [GHB13]. The first one states that when the same block is executed repeatedly the source actor in that block should execute periodically. The other constraint that must hold is that the source in the block after a mode switch must execute exactly one period after the

last execution of the source in the previous block. The first constraint is fulfilled by enforcing that a strict periodic schedule is computed for the source actor in a block. The second constraint is verified in the analysis flow, as will be discussed in Section 4.4. Satisfaction of both constraints indicate that a schedule is valid for all parameter values.

Now, it might appear that flattening a block is not allowed because the production moments of tasks in nested blocks seem to depend on the number of iterations a block is executed. For example, in Figure 4.5a the enabling of v_d depends on the finish time of the p 'th execution of v_b . The enabling jitter of v_d therefore appears to depend on the block parameter value p which would prevent the block to be flattened. However, we can use the fact that the computed schedules of the actors in a block are periodic. The actual start time of an execution of a task τ_i will be in between the schedule that forms a lower bound, \check{s}_i , and the periodic schedule that is used as an upper bound, \hat{s}_i . Therefore we can conclude from Equation 4.6 that the enabling jitter of v_d is independent of p .

$$J_d(p) = \hat{s}_d - \check{s}_d \quad (4.3)$$

$$= (\hat{s}_b(p) + \hat{\rho}_b) - (\check{s}_b(p) + \check{\rho}_b) \quad (4.4)$$

$$= \hat{s}_b + (p-1) \cdot P_b - \check{s}_b + (p-1) \cdot P_b + \hat{\rho}_b - \check{\rho}_b \quad (4.5)$$

$$= \hat{s}_b - \check{s}_b + \hat{\rho}_b - \check{\rho}_b \quad (4.6)$$

In the equations above $J_d(p)$ is the enabling jitter of v_d for iteration p , \hat{s}_d the latest possible enabling time of v_d in the periodic schedule and \check{s}_d the earliest possible enabling time. The jitter caused by τ_b itself depends on its maximum firing duration $\hat{\rho}_b$ and minimum firing duration $\check{\rho}_b$. Because the enabling jitter for v_b is independent of p , we can use a single iteration of a block for the calculation of the enabling jitter of actors at a higher hierarchical level.

The response time of tasks at a higher hierarchical level can be calculated by setting the parameters of the nested blocks to one, because this results in the maximum interference for tasks at a higher level. The response time for the tasks at a higher hierarchical level is determined by the response time, period, and jitter of tasks at that level according to Equation 4.2 after additional constraints are added to ensure that blocks can be analyzed in isolation. The jitter caused by tasks in nested blocks is independent of the parameter value of that block, according to Equation 4.6 as explained in the previous paragraph. Increasing the parameter value of a block changes the period of the tasks at a higher hierarchical level. Executing a nested block more often results in a less frequent execution of the tasks at a higher hierarchical level. The period of these tasks is thus effectively increased by executing nested blocks more often. An increase of the period of tasks results in lower worst-case response time according to Equation 4.2. The worst-case response times are therefore also independent of the parameter value of a block. Therefore, the minimal parameter value of one can be used for nested blocks to calculate an upper bound of the response time of tasks at higher hierarchical levels.

As an example, consider flattening of the nested block in the SVPDF model in Figure 4.5a, such that the response time of the tasks at the highest level in the model can be determined. First, the response times of tasks τ_b and τ_e at the deepest nested level are determined by analyzing their blocks. The blocks are flattened by fixing the response times of these two tasks to the computed worst-case response times, and removing the boundaries of the blocks as shown in Figure 4.5b. Using fixed values for the response times of the tasks in nested blocks is allowed since the additional constraints that are a result of the added locks and barriers ensure that tasks outside a block cannot increase the response time of tasks inside a block. After the nested block are flattened, the response times of τ_a , τ_c and τ_d can be calculated.

Flattening of a hierarchical level removes some potentially useful information. Consider again the SVPDF model shown in Figure 4.5a where the colors represent the mapping to a processor. On one processor, we have that v_c has a lower priority than v_d . The time at which v_d becomes enabled is dependent on p . From a certain value of p , v_c will always finish before v_d is enabled and therefore v_d will not interfere with v_c . The knowledge about a minimum number of block executions can be used as offset information to obtain a less pessimistic response time for v_c . The use of offset information to obtain more accurate results for cyclic applications that do not contain modes is presented in [KHB16a].

4.4 PERIODIC SOURCE CONSTRAINTS

In this section, we derive the conditions that need to be satisfied for applications containing multiple modes in which the same source is accessed. Additional constraints need to be introduced to verify that the source task can execute strictly periodically when switching between modes.

The source task in an application needs to execute strictly periodically. Therefore, the N actors derived from the source task, for all N blocks where the source is read, need to fire periodically. The source actors fire periodically when the time between the start times of firings of the same, or consecutive source actors, is exactly one period P_s of the source task. The two additional constraints in Equation 4.7 and Equation 4.8 on the start times of the source actors, v_s^q with $q \in \{0, 1, \dots, N-1\}$, are therefore added.

$$\hat{s}_s^q(i+1) = \hat{s}_s^q(i) + P_s \quad (4.7)$$

$$\hat{s}_s^{(q+1) \bmod N}(i+1) = \hat{s}_s^q(i) + P_s \quad (4.8)$$

where $\hat{s}_s^q(i)$ is the upper bound on the start time of source actor v_s^q in iteration $i \in \mathbb{N}_0$ if that source actor fires in iteration i . The constraint in Equation 4.8 is added in step 2c of the analysis flow presented in Section 4.3. This constraint is added in the hierarchical level where all the blocks containing source actors are flattened such that each source actor fires only once before switching to the next source actor.

As a result of the added constraints the source actors must fire strictly periodically. A delay in the enabling time of a source actor will immediately lead to a violation of the throughput constraint in the block one level higher in the hierarchy than the blocks containing the source actors, which is $2P_s$ in the example since there are two source actors. A delay in the enabling of a source actor occurs when the inputs of a mode arrive too late and the constraints resulting from a barrier do not enable the source actor in time. A violation of the throughput constraint occurs when the sum of the response times of the tasks on the path of edges without tokens from one source actor to the next source actor, via the constraint that result of locks and barriers, is larger than one period. An example of such a path is seen in Figure 4.1c from actor src_2 to v_d, v_c via the red dotted and orange dashed edges originating from a lock and barrier to src_1 . The combination of the constraints of the strictly periodic executing source and the barriers and locks ensures that mode transitions can be analyzed.

4.5 RESPONSE TIMES

In the previous sections, constraints are enforced to enable independent analysis of modes. In this section, a general WCRT equation is derived that makes use of the fact that modes can be analyzed in isolation after additional constraints are added in the dataflow model. The equation is valid for schedulers in the non-starvation-free class which includes FPP, RR and TDM schedulers. We make use of the proof that the constraints resulting from locks prevent any interference of tasks in other modes as is given in Subsection 4.5.1.

In general, the WCRT \hat{R}_i of a task τ_i consists of its WCET B_i and an upper bound on the interference I of tasks assigned to the same processor, as shown in the following equation:

$$\hat{R}_i = B_i + I_{scheduler}(i, \hat{R}_i) \quad (4.9)$$

The interference depends on the type of scheduler used on a processor. We will first consider FPP schedulers and limit \hat{R}_i to P_i , to satisfy the constraints presented in Section 4.4. Since this limitation implies that $w_i \leq P$, only a single execution of τ_i needs to be considered such that $q = 1$. The WCRT of Equation 4.2 can then be simplified to the following equation for the interference on τ_i :

$$I_{FPP}(i, \Delta t) = \sum_{j \in hp(i) \setminus M(i)} \left\lceil \frac{J_j + \Delta t}{P_j} \right\rceil \cdot B_j \quad (4.10)$$

where $M(i)$ is the set of tasks in another mode as τ_i that is made mutually exclusive to τ_i using a lock. Using the prove given in Subsection 4.5.1 the set of interfering tasks is reduced to tasks in the same mode by using locks. For RR and TDM schedulers a similar expression can be derived where the interference is a summation over the tasks assigned to the same processor. The following equation for the inter-

ference for the different types of schedulers is obtained:

$$I_{RR}(i, \Delta t) = \sum_{j \in \mathcal{T}(i) \setminus M(i)} B_j \quad (4.11)$$

$$I_{TDM}(i, \Delta t) = \left\lfloor \frac{B_i}{S_i} \right\rfloor \cdot (Q_m - S_i),$$

$$\text{where } Q_m = S_i + \sum_{j \in \mathcal{T}(i) \setminus M(i)} S_j \quad (4.12)$$

where S_i is the budget of τ_i within replenishment interval Q_m within mode m and $\mathcal{T}(i)$ the set of tasks mapped to the same processor as τ_i excluding τ_i itself. We have shown that for FPP, RR and TDM schedulers an upper bound on the interference can be derived only consisting of interference of tasks within the same mode as the task being analyzed.

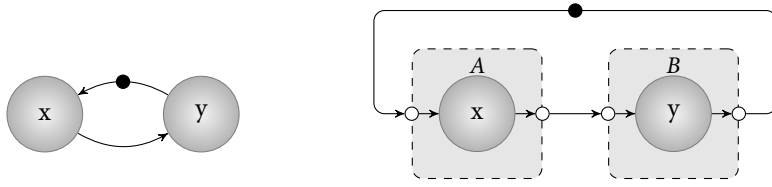
4.5.1 MUTUAL EXCLUSIVE EXECUTION USING LOCKS

In this section, we prove that the constraints imposed by a lock ensure that tasks in different modes will not interfere with each other and therefore execute mutually exclusive.

For HSDF graphs it has been proven that the number of tokens on a cycle determines the interference of tasks on the cycle [WHGB14]. A cycle with one token on it, as shown in Figure 4.6a, prevents interference between actor x and y and therefore these actors will fire mutually exclusively. In this section, it is proven that this also holds for two actors in different blocks in an SVPDF graph, like the one in Figure 4.6b. If the actors belong to the same block, then they potentially interfere.


The SVPDF graph in Figure 4.6b has two actors x and y in different modes, A and B , and the edges represent the constraints that are a result of a lock. Any number of modes and tasks within a mode is supported by the lock, but in the proof we will for clarity only consider two modes each containing a single task. In the proof we will make use of the following notation: $a < b$ means a dependency from a to b , $s_a(n)$ is the start of firing $n \in \mathbb{N}_0$ of actor a , $f_a(n)$ is the finish of the n -th firing of actor a . Function $\varphi_A(n) \in \mathbb{N}_0$ returns the block iteration counter in which firing n of an actor in block A takes place. A new block iteration is started when the next token is consumed by all port actors of the block. Similarly the function $\varphi_B(n) \in \mathbb{N}_0$ returns the block iteration counter in which firing n of an actor in block B takes place. Also functions that indicate the start and finish time of an iteration of a block are defined.

By definition an actor x can cause interference on an actor y when actor x finishes its n -th firing later than the start of the m -th firing of y . Similarly an actor y can cause interference on an actor x when actor y finishes its m -th firing later than the start of the n -th firing of x . Two actors x and y fire mutually exclusive if x cannot cause interference on actor y , and y can not cause interference on actor x . That this holds for the actor x and y in the different blocks in Figure 4.6b can be seen as follows:



(a) HSDF graph consisting of actor x and y

(b) SVPDF graph consisting of two modes, A and B , each containing one actor, x or y

 Figure 4.6: The constraints resulting of a lock for an HSDF model ((a)) and an SVPDF model ((b))

The edge e_{xy} represents a dependency. As a result of this edge it holds that:

$$f_x(n) < f_A(\varphi_A(n)) < s_B(\varphi_B(m)) < s_y(m), \quad \{n, m \in \mathbb{N}_0 \mid \varphi_A(n) = \varphi_B(m)\} \quad (4.13)$$

Since it holds that $f_A(\varphi_A(n) - 1) < f_A(\varphi_A(n))$ it follows from Equation 4.13 that:

$$f_x(n) < s_y(m), \quad \{n, m \in \mathbb{N}_0 \mid \varphi_A(n) \leq \varphi_B(m)\} \quad (4.14)$$

Now we have that x can only cause interference on actor y according to Equation 4.14 if:

$$\varphi_A(n) > \varphi_B(m) \quad (4.15)$$

A similar reasoning holds for the edge e_{yx} which has one initial token. This initial token causes that the start of the z -th block iteration of A depends on the finish of the $(z - 1)$ -th block iteration of B . Therefore it holds that:

$$f_y(m) < f_B(\varphi_B(m)) < s_A(\varphi_A(n) - 1) < s_x(n), \quad \{n, m \in \mathbb{N}_0 \mid \varphi_B(m) = \varphi_A(n) - 1\} \quad (4.16)$$

Since $f_B(\varphi_B(m) - 1) < f_B(\varphi_B(m))$ it follows that:

$$f_y(m) < s_x(n), \quad \{n, m \in \mathbb{N}_0 \mid \varphi_B(m) \leq \varphi_A(n) - 1\} \quad (4.17)$$

We have that the m -th firing of actor y can only cause interference on the n -th firing of x if the m -th firing of y can finish later than the n -th start of x , which is true according to Equation 4.17 if:

$$\varphi_B(m) > \varphi_A(n) - 1 \quad (4.18)$$

Therefore there is no interference from actor x on y and from actor y on x if Equation 4.15 and Equation 4.18 hold, thus:

$$\varphi_A(n) - 1 < \varphi_B(m) < \varphi_A(n) \quad (4.19)$$

Because there is no block iteration counter value $\varphi_B(m)$ for which Equation 4.19 holds we conclude that the actors x and y do not interfere and therefore fire mutually exclusive, which concludes the proof.

The proof remains valid for q consecutive executions of an actor x starting at firing n . This is because these consecutive executions must belong to the same block iterations such that $\varphi_A(n + q - 1) = \varphi_A(n)$. As a consequence, Equation 4.19 still holds since there is no block iteration counter value $\varphi_B(m)$ for which actors x and y can interfere.

4.6 RESPONSE TIMES LARGER THAN PERIOD

The approach presented in the previous sections is only applicable if all tasks have a WCRT smaller than the period of the source. In this section, we present conditions under which WCRTs larger than the period are possible, however, our impression is that these conditions will be rarely satisfied in practice.

The approach presented in the previous sections does not allow WCRT larger than the period of the source as a result of the barriers. These barriers were introduced to take care that the arrival times of the input data of a block could not affect the response times of the tasks in that block. However, the jitter in the arrival times of the input data can only have an effect on other tasks if these tasks have a sufficiently high priority or have a dependency towards other tasks. If this is not the case then there is no need for the barrier and WCRT larger than the period of the source can be allowed. The simplified WCRT equation derived in Section 4.5 is invalid for tasks with a response times larger than the period, since multiple executions of a task need to be considered as is accounted for in the equation stated in Equation 4.2.

Summarizing, the constraint of a barrier for a task are only allowed to be remove if the task cannot cause interference on other tasks. This can only be fulfilled if the tasks has:

- » No tasks with a lower priority mapped to the same processor
- » No dependencies within mode, excluding all port actors

In Figure 4.7, a constructed example is shown that illustrates a case in which the barrier can be removed for some inputs of a block. This allows a WCRT larger than the period of the source, which must be compensated for in a subsequent mode. In this example, there are five tasks divided across two modes. Task τ_0 , τ_1 and τ_2 share a processor, as indicated by the colors of their actors, where τ_0 has a high priority and τ_2 a low priority. The WCET of τ_0 , τ_2 , and τ_4 is $\frac{1}{4}P$ and τ_1 , and τ_3 have a WCET of $\frac{1}{2}P$. Only τ_3 has a BCET not equal to its WCET and is assumed to be 0 and therefore v_3 will introduce a jitter in the enabling time of v_1 . Based on this jitter, τ_1 can pre-empt τ_2 not only once, but twice during τ_2 's execution, increasing \hat{R}_2 to $1\frac{1}{4}P$.

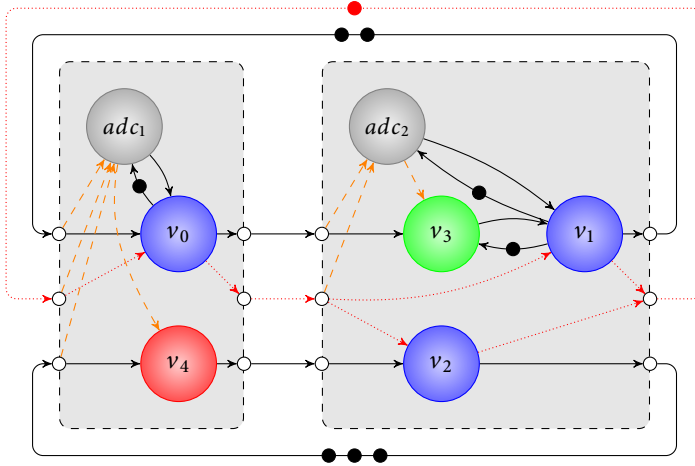


Figure 4.7: SVPDF model of an application where the WCRT of v_2 can be greater than the period of the source

When the dependencies resulting from the barrier would be introduced from the port actor producing the input of v_2 then an $\hat{R}_2 > P$ will lead to the conclusion of infeasibility given that the source period is P . However, in this example τ_2 is allowed to be enabled earlier or later than the source actor starts in its corresponding block. The enabling jitter that can be caused by removing the dependency from the input that is a result of the barrier will not cause interference on other tasks in this example, since τ_2 cannot interfere because it has the lowest priority. Moreover, the jitter of τ_2 is not propagated to the other tasks because there are no dependencies to other tasks and therefore there is no increase of the jitter of other tasks. The large WCRT of τ_2 in this case is compensated by the sufficiently small WCRT of τ_0 and τ_4 in the other mode, because the sum of their WCRTs is within the constraint of one execution of a source in each mode, which takes $2P$.

4.7 CASE STUDY

In this section, the analysis approach that has been introduced in this chapter is demonstrated using a simplified WLANp receiver application. The organization of this section is as follows. First we describe the application of which a dataflow model will be derived that includes the constraints that are a result of locks and barriers. We apply temporal analysis to this model to determine the WCRTs of tasks and the buffer sizes. A simulator is used to verify the analysis results and to generate a trace of the executions of the tasks on shared resources. Finally, we show that a pipelined execution of tasks is supported although additional constraints are introduced in the application.

The OIL program of the WLANp application is shown in Figure 4.8. The application



```

1 source ADC @ 250 kHz;
2
3 loop{
4   loop{
5     h = detectHeader(ADC);
6     vh = validHeader(h);
7     NSym' = decodeHeader(h);
8   } while(!vh);
9   n = 0;
10  loop{
11    x = fft(ADC);
12    y = demap(x);
13    z = deint(y);
14    w = convDecode(z);
15    crc(w);
16    n' = n + 1;
17  } while(n < NSym);
18 } while(1);

```

 Figure 4.8: WLANp receiver application

receives its input from a periodic ADC running at 250 kHz (a period of 4 μ s). This source imposes a throughput constraint on the execution of the application.

The WLANp application contains two modes. Each mode corresponds to a potentially endlessly repeated while-loop.

In the first mode the `detectHeader` function reads symbols from the source until it detects a header of a packet. The `decodeHeader` function extracts the size of the payload from the header while the `validHeader` function determines in parallel if the checksum of the header is valid. Only when a valid header is found the first mode is left as is specified in the loop condition.

The `fft` function in the second modes reads a number of symbols from the source based on the size of the payload and performs a transformation to the frequency domain on each of these symbols. The other functions in this mode perform demapping, deinterleaving, convolution decoding and verification of the CRC. The number of loop iterations in this mode is dependent on the result of the first mode and can be determined before tasks in the second mode start.

The multiprocessor compiler Omphale [GHB13] is used to transform the program shown in Figure 4.8 into a task graph where each function results in a task. The variables used in the program are converted into buffers to allow a pipelined execution of the tasks. The WCETs of the `detectHeader`, `validHeader` and `decodeHeader` tasks are assumed to be 2 μ s, 1 μ s and 1 μ s as is shown in Table 4.1a. The `fft`, `demap`,

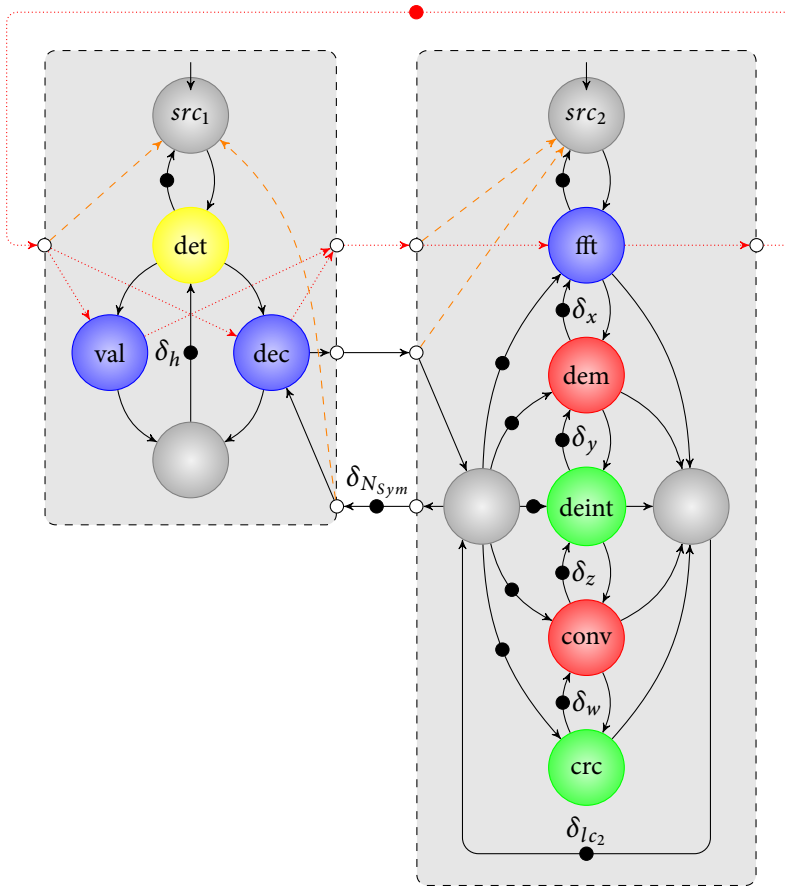


Figure 4.9: SVPDF model of the application in Figure 4.8 including additional constraints of a lock and barriers

deint, *conv* and *crc* tasks in the second mode have an execution time of 3 μ s, 1.5 μ s, 2 μ s, 1.5 μ s and 2 μ s respectively.

In this case-study we assume there are more tasks than processors. Therefore, multiple tasks are scheduled onto a processor using a scheduling policy. Multiple scheduling policies are used for different processors to demonstrate that the approach presented in this chapter can also be used for other schedulers than the FPP scheduler. Tasks *validHeader*, *decodeHeader*, and *fft* are scheduled by an FPP scheduler on one processor. On that processor task *decodeHeader* has the highest priority and task *validHeader* the lowest priority. Another processor runs the tasks *deint*, and *crc* using a RR scheduler. A budget scheduler is used on another processor to schedule the tasks *demap* and *convDecode*. Both tasks are allocated a budget of 0.5 μ s every 1 μ s. The remaining task *detectHeader* runs on a dedicated processor.

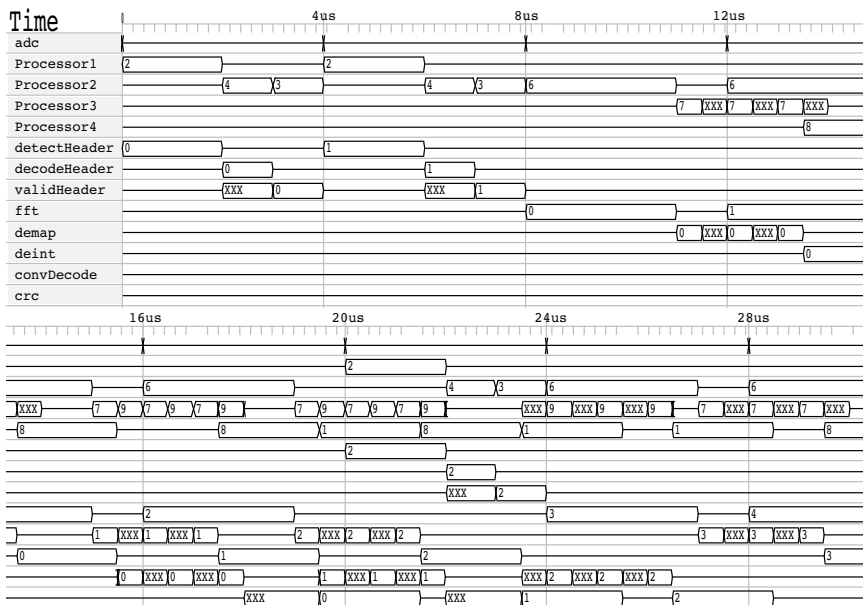
Table 4.1: Temporal analysis results of the WLANp application


(a) Task execution times and derived response times			(b) Derived buffer capacities	
task	WCET (μs)	\hat{R} (μs)	buffer	capacity
<i>detectHeader</i>	2.0	2.0	δ_h	1
<i>decodeHeader</i>	1.0	1.0	δ_{vh}	1
<i>validHeader</i>	1.0	2.0	δ_x	2
<i>fft</i>	2.5	2.5	δ_y	2
<i>demap</i>	1.5	3.0	δ_z	2
<i>deint</i>	2.0	2.0	δ_w	2
<i>convDecode</i>	1.5	3.0	$\delta_{N_{Sym}}$	1
<i>crc</i>	2.0	4.0	δ_{lc1}	1
			δ_{lc2}	4

In order to verify the temporal constraint imposed by the periodic source, the compiler also generates an SVPDF model besides the task graph. An actor is introduced for each task in this model as is shown in Figure 4.9. The blocks in the model correspond to while-loops in the application. The colors of the actors correspond to the mapping to a processor. Buffers are represented as a forward and backward edge containing the number of tokens corresponding to the capacity of the buffer. Since the *validHeader*, *decodeHeader* and *fft* tasks run on the same processor, but do not belong to the same mode a lock is added. The analysis flow requires a strictly periodic execution of the source to be able to determine response times for an FPP scheduler. Therefore, additional constraints on the start times of source actors are added in step 2c of the analysis flow. The precedence constraints that results for the lock are represented in the model by the red dotted edges. The additional constraints that are a result of the barriers are shown as the orange dashed edges.

Using the analysis flow as defined in Section 4.3 and the response time equation Equation 4.9 of Section 4.5 we derive WCRTs for the tasks in these two modes. The resulting WCRTs are listed in Table 4.1a. Table 4.1b shows the computed buffer capacities given these WCRTs. The last two buffers in the table, δ_{lc1} and δ_{lc2} , represent a buffer in which a loop condition is stored that is read by all tasks in the corresponding mode. The size of these buffers affects the maximum amount of tasks that can execute in parallel.

We use the high-level system simulator HAPI to verify the obtained analysis results. HAPI was initially a dataflow simulator [BPvMo5], but was recently extended with the addition of processor sharing. This addition allows us to obtain simulation results of task graphs, which can be used to falsify erroneous analysis results [KHB16b]. No constraint violations were detected for the derived buffer capacities as the source could fire strictly periodically. The traces generated with the HAPI simulator are shown in Figure 4.10. In this figure, there is one trace for



 Figure 4.10: Execution trace of the WLANp application from Figure 4.9 with mutual exclusivity applied

each processor in which the currently running task and the iteration number of the task is shown. In each trace pre-emptions based on either priorities or depletion of budget are indicated by X's. The trace in Figure 4.10 shows pipeline parallelism for example at a time of $24\ \mu\text{s}$, at which the *fft* task already processes the next symbol of the source whereas the CRC of a previous symbol is being computed by the *crc* task.

Pipelining of the WLANp application would be impossible for certain assignments of tasks to processors. The additional constraints in the application as a result of barriers and locks can prevent pipeline parallelism inside a mode. In the example the first task in the second mode (*fft*) is assigned to the same processor as tasks in the first mode. The combined constraints of the buffer from the source to *fft* and the lock and barrier, represented by the red dotted and orange dashed edge in Figure 4.9, require *fft* to process a sample from the source within one period when switching to the first mode. When for example task *crc* would be mapped to that processor instead of task *fft*, multiple tasks need to be finished within the same period. In the dataflow model, the path without initial tokens from actor *src₂* via *fft*, *demap*, *deint* and *convDecode* to actor *crc* would then be a bottleneck and limit pipelining since task *crc* would then release the lock. The barrier in the first mode requires the lock to be acquired before the source in the same mode finished its first execution. Therefore, it can be concluded that the amount of achievable pipeline parallelism depends on the mapping of tasks to processors when an FPP

scheduler is used.

4.8 CONCLUSION

This chapter presents a compositional temporal dataflow analysis approach for cyclic stream processing applications with modes executed on multiprocessor systems that use FPP scheduling.

The analysis approach is based on the ability to independently characterize the temporal behavior of modes. This is ensured by adding locks in the application which make the execution of tasks that belong to different modes but executed on the same processor mutually exclusive. Furthermore, barriers are added such that together with the locks the interference of the tasks in a mode is independent of tasks belonging to other modes. The additional constraints introduced by the barriers and locks guarantee that composition of modes does not change their individual characterization. As a result, applications containing a hierarchy of modes can be described in an SVPDF model. This model can be analyzed by recursively applying existing dataflow analysis techniques, to determine the worst-case temporal behavior. The SVPDF model and the parallel implementation including locks and barriers are generated by a multiprocessor compiler.

Furthermore, it is shown that the approach allows for response times of tasks larger than the period of the source. It is also shown that systems in which a combination of budget schedulers and FPP schedulers are applied can be analyzed.

The applicability of the approach is demonstrated using a IEEE 802.11p receiver application. We show that this application can be executed pipelined despite the inserted locks and barriers. The analysis results are verified using a dataflow simulator.



LATENCY ANALYSIS USING TIMED AUTOMATA

ABSTRACT – Functional and temporal guarantees cannot always be given for dataflow models supporting auto-concurrency. As a result of auto-concurrency and variations in firing durations, tokens can be produced, and more problematic for functional correctness, consumed out-of-order. The Homogeneous Synchronous Dataflow with auto-concurrency (HSDF^a) model does ensure functional correctness when tokens are produced out-of-order, but there currently does not exist an exact end-to-end latency analysis technique for these HSDF^a models. Moreover, neither does there currently exist a dataflow analysis approach for HSDF^a models that accurately takes the effect of correlation of firing durations of consecutive firings into account on the end-to-end latency. In this chapter we therefore present a transformation of strongly connected HSDF^a models into timed automata models. This enables an exact end-to-end latency analysis by performing model checking on these timed automata models.

Analysis of real-time systems can on the one hand be performed with timed-dataflow models, as described in the previous chapters. The SVPDF model used there, however, does not support auto-concurrency as it requires all actors to have an implicit self-edge. Analysis techniques have been developed for dataflow models to determine the guaranteed throughput and maximum latency [MB07]. The behavior of some of these timed-dataflow models can be described using max-plus

This chapter is based on [GK:4].



algebra [dG⁺12]. Approximation techniques are often applied in order to reduce the computational complexity of the throughput and latency analysis methods of these models [H⁺16]. These approximation techniques make use of a deterministic abstraction which reduces the accuracy.

Another approach is to model these real-time systems with timed automata. For these type of systems a partitioning can be derived into a finite number of sets of similar states. These sets are a result of restrictions to clocks with the same rate of change, $\dot{x} = 1$, and constant integer constraints on transitions. These finite number of sets can be analyzed using a model checker like UPPAAL [DILSo9] to compute the exact maximum end-to-end latency. However, such an exhaustive analysis might increase the run-time of the analysis significantly.

The HSDF^a model, which we focus on in this chapter, is closely related to the HSDF model. Like in the HSDF model, the actors in the HSDF^a model produce one token on each output queue and consume one token from each input queue per firing. However, unlike the HSDF model, the dependencies between firings are preserved despite that actors have a varying firing duration and are executed auto-concurrently. In the HSDF^a model tokens are consumed in index order instead of timestamp order. Auto-concurrency can be used to model data parallel executions of tasks [H⁺14]. As a result, the behavior of the HSDF^a model can be described with max-plus algebra, whereas this is not possible for the HSDF model with actors that have varying firing durations. Furthermore, the correlation between the firing durations of actor firings can be expressed. However, currently no exact end-to-end latency analysis technique exists that takes this correlation into account.

In this chapter we present a transformation of strongly connected HSDF^a models into timed automata. We show that by transforming HSDF^a models into timed automata, we can compute the exact end-to-end latency using UPPAAL. In the case study we compare the latency for two HSDF^a models computed using a timed automata approach with the latency obtained using a state-of-the-art dataflow based analysis technique that relies on a deterministic abstraction of the HSDF^a model. We also compare the run-times of these approaches.

This chapter is structured as follows. In Section 5.1 we discuss work related to latency analysis techniques for dataflow models and the transformation of dataflow models into timed automata. In Section 5.2, we introduce the HSDF^a model and define the latency analysis problem. In Section 5.3, we define the semantics of the HSDF^a model using max-plus algebra. In Section 5.4 we introduce the extended timed automata model that is used in UPPAAL. In Section 5.5 we show that under certain conditions a timed automata model can be created that has the same behavior as an HSDF^a model. The case study is presented in Section 5.6. We state our conclusions in Section 5.7.

5.1 RELATED WORK

In this section we present work that is related to the modeling and analysis of dataflow graphs. We first discuss the relation between some dataflow models and the HSDF^a model. Then we discuss work related to the transformation of Time Petri nets (TPNs), to which the HSDF^a model is closely related, into timed automata. Finally, we discuss related work on other end-to-end latency analysis techniques for dataflow models.

The SDF model supports integer production and consumption rates and is a generalization of the HSDF model. The CSDF model [B⁺95] has phases and is a generalization of the SDF model. Each phase can have a different consumption and production rate and these phases fire in a fixed cyclic order. In the HSDF, SDF and CSDF models, tokens can overtake each other as a result of auto-concurrency. In the CSDF^a model [K⁺16], which is based on CSDF, an index is added to tokens to take care that dependencies between firings are independent of the production order of tokens. We consider the variant of the CSDF^a model with one phase and single rate and refer to it as HSDF^a.

A transformation of TPNs into timed automata to determine the end-to-end latency is presented in [GS02]. Both HSDF^a graphs and Petri nets do allow reordering of tokens, but a key difference is that Petri nets do not preserve dependencies between iterations of data-dependent actors. As a result, the temporal behavior of HSDF^a graphs can be described with max-plus algebra, which does not hold in general for Petri nets. Petri nets support auto-concurrency. However, we show that auto-concurrency cannot be modeled in a timed automaton in general. The number of simultaneously executions needs to be upper bounded. This contradicts the claim made in [GS02] that the transformation is always possible.

End-to-end latency analysis techniques for self-timed executed SDF graphs are presented in [MB07, G⁺07]. The approach in [MB07] does consider bursts of events. However, the approach does not take into account that firing durations of actors can be correlated. The approach in [G⁺07] does not consider variations in execution times, and the correlation between successive executions. This is also the case for the timed automata based throughput analysis approach in [A⁺14]. Our approach does allow the firing durations of actors to vary each firing and takes the correlation between the firing durations of different firings into account. Our approach also supports auto-concurrent execution of actors but requires the HSDF^a graphs to be strongly connected.

5.2 THE HSDF^a MODEL

In this section we first present the HSDF^a model. Then we give a definition of the maximum end-to-end latency for this model.

An HSDF^a graph is a directed graph $G = (V, E, \delta, \rho)$ that consists of a set of actors V connected by a set of directed edges E . An actor $v_i \in V$ communicates with

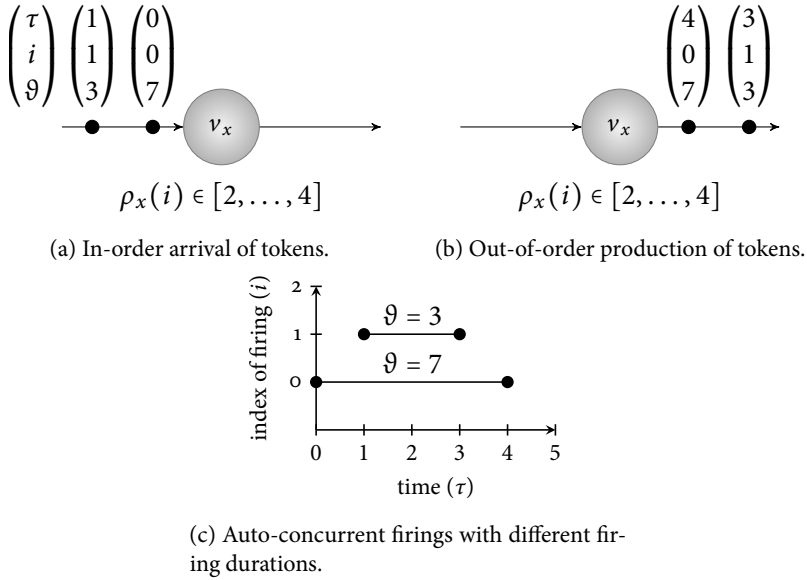


Figure 5.1: Auto-concurrency and varying firing durations cause reordering of tokens.

another actor v_j by producing tokens on an edge $e_{ij} \in E$. Each edge represents an unbounded buffer instead of a queue because tokens can be written into these buffers in a different order than they are read. Initially there are $\delta_{ij} \in \mathbb{N}_0$ tokens on an edge. An arrival of a token is an event which is represented by the tuple (ϑ, i, τ) consisting of a value ϑ , an index i , and a time-stamp $\tau \in \mathbb{R}_0$. An HSDF^a actor v_i is enabled to fire its i -th iteration if there is at least one token with index i on all its incoming edges. During self-timed execution of an HSDF^a graph, each actor fires immediately after it is enabled. When an actor fires, one token is consumed from each of its incoming edges and one token with index i is produced on each outgoing edge of the actor. The tokens are produced exactly $\rho_x(i) \in [\check{\rho}_x, \hat{\rho}_x]$, chosen non-deterministically, after the enabling of the actor. The firing duration $\rho_x(i)$ of the i -th firing of an actor v_x can be defined more precisely using e.g. a non-deterministic finite state-machine. Multiple firings of an actor overlap if there are sufficient input tokens with the required indices. A token with index $i + 1$ can be produced by an actor before the token with index i is produced even if input token i and $i + 1$ arrive at the same point in time. This is because firing $i + 1$ can have a smaller firing duration than firing i . However, the consumption order of tokens is determined by the indices and is therefore independent from the production order.

An example of reordering of tokens is shown in Figure 5.1. Two tokens arrive in-order according to both their index and timestamp, since both are increasing, as can be seen in Figure 5.1a. However, the combination of varying firing durations

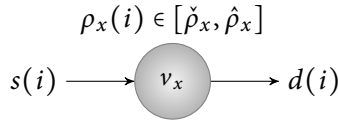


Figure 5.2: A basic HSDF^a graph.

of v_x and auto-concurrency can result in and out-of-order production of tokens, as shown in Figure 5.1b. At timestamp 1, both tokens have been consumed and are processed in parallel by v_x , as shown in Figure 5.1c. The first token experiences the worst-case firing duration and the second one the best-case firing duration of v_x . As a result, the token with a higher index is produced at an earlier timestamp than the one with the lower index, and thereby the two tokens have been reordered.

We define the *end-to-end latency* of a HSDF^a graph as $L_{sd} = \max_i (d(i) - s(i))$ with $s(i)$ the arrival moment of the token with index i in the input buffer, and $d(i)$ the production moment of the token with index i in the output buffer. Figure 5.2 shows a basic HSDF^a example consisting of an actor, v_x , for which $L_{sd} = \rho_x(i)$. Overlapping firings can be prevented by adding a self-edge with one initial token to an actor. In that case is L_{sd} only equal to the firing duration $\rho_x(i)$ if $s(i) > d(i-1)$. However, given a burst of events at the input it can be the case that the next input arrives before the previous output has been produced, i.e. $s(i+1) < d(i)$. Such a burst increases the maximum end-to-end latency as a result of that tokens will stay longer in the input buffer.

5.3 MAX-PLUS SEMANTICS OF HSDF^a

In this section we use max-plus algebra to describe the evolution of actor firing times during the self-timed execution of an HSDF^a graph. The events that we will relate in max-plus expressions are the completion of firings. These events are related through precedence constraints, which are imposed by the channels in an HSDF^a graph: the times at which an actor may fire depends on the times at which sufficient tokens become available on its incoming channels.

Max-plus algebra [B⁺92] only uses a max operator \oplus , and a plus operator \otimes . The \otimes operator has a higher precedence than the \oplus operator. The production events of actors during self-timed execution can be described using these operators.

Describing a dataflow graph, i.e. HSDF, with max-plus algebra assumes the graph to be functional deterministic, e.g. events remain in-order. The dataflow graphs we consider make use of the combination of auto-concurrency and non-constant firing durations of actors, which break this event ordering assumption. Functional determinism is obtained for HSDF^a graphs by consuming tokens in index order, even when the timestamps of tokens can be reordered. The production events of HSDF^a actor in max-plus therefore refer to the timestamp of the production of a token with a specific index.

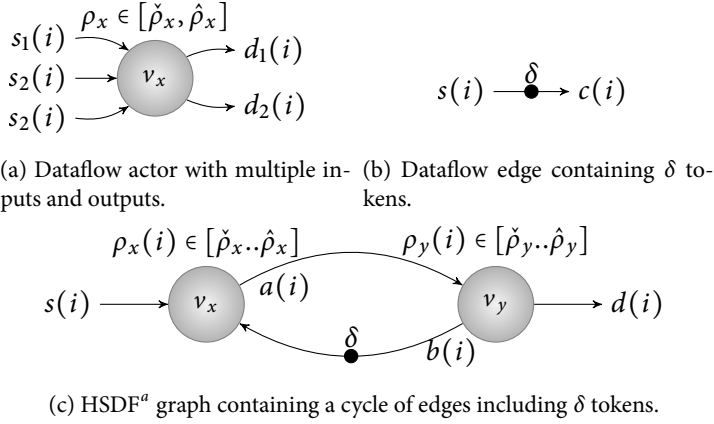


Figure 5.3: HSDF^a graphs used to explain latency derivation.

The following two cases need to be considered to be able to describe the production moments in HSDF^a graphs using max-plus:

1. Actors with multiple inputs and outputs and an iteration dependent firing duration.
2. Edges with $\delta \geq 0$ initial tokens.

If an actor has multiple incoming edges, the edge on which the token with the required index arrives latest determines the time at which the actor is enabled. Tokens are produced on all outgoing edges of an actor after its firing duration. The production moments of the actor in Figure 5.3a can therefore be described with the following max-plus expression:

$$d_1(i) = d_2(i) = (s_1(i) \oplus s_2(i) \oplus s_3(i)) \otimes \rho_x(i) \quad (5.1)$$

Figure 5.3b depicts an edge containing δ initial tokens. The consuming actor can consume δ tokens before the first firing of the actor that produces tokens on the edge. This can be described with Equation 5.2 in which $c(i)$ is the earliest possible consumption moment, and negative indices correspond with an arrival at timestamp 0.

$$c(i) = s(i - \delta) \quad (5.2)$$

The maximum end-to-end latency for a dataflow graph can be described by combining expressions for actors and edges. Tokens on an edge represent dependencies of the consuming actor on a firing with a lower index of the producing actor. Waiting until an actor is enabled by all its inputs is enforced by the \oplus operator. The delay caused by the firing duration is added in max-plus algebra using the \otimes operator. We

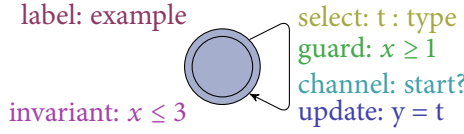


Figure 5.4: Extended timed automaton annotations.

can now derive L_{sd} for the cyclic HSDF^a graph as shown in Figure 5.3c as follows:

$$a(i) = (s(i) \oplus b(i - \delta)) \quad \otimes \quad \rho_x(i) \quad (5.3)$$

$$d(i) = b(i) = a(i) \quad \otimes \quad \rho_y(i) \quad (5.4)$$

$$L_{sd} = (s(i) \oplus b(i - \delta)) \quad \otimes \quad \rho_x(i) \otimes \rho_y(i) \quad (5.5)$$

In these expressions $a(i)$ and $b(i)$ represent the moments at which the token with index i is produced on e_{xy} and e_{yx} respectively.

Although max-plus algebra can be used to describe the constraints of an HSDF^a graph, only a constant (worst-case) firing duration is used for the actors. Using an analysis approach that can capture variations in firing durations and correlations between firing durations will lead to more tight analysis results. In the next section, we will therefore model the constraints from a HSDF^a graph in timed automata, and extend them in Section 5.6 with correlations between firing durations to improve analysis results.

5.4 EXTENDED TIMED AUTOMATA

In this section we present the extended timed automata model which is used in UPPAAL. Extended timed automata will be used in the next section to create models of HSDF^a graphs. The timed automata model as described in Subsection 2.1.4, is extended with finite data variables, urgent and committed locations and communication channel. All of these extensions only affect the ease of modeling and introduce structure in the model, but cannot introduce undecidability.

An extended timed automaton, as used in UPPAAL, is a directed graph $\mathcal{A} = (L, Act, C, \mathcal{E}, Inv, l^0)$ where L is a set of locations, Act a finite set of actions, C a set of clocks, $\mathcal{E} \subseteq L \times Act \times B(C) \times 2^C \times L$ is a set of edges, $Inv : L \rightarrow B(C)$ assigns an invariant to each location and l^0 is the initial location. The invariant of a location expresses an upper bound $\hat{c} \in \mathbb{N}_0$ on a clock $c \in C$. A location $l \in L$ can be urgent, meaning that no time is allowed to pass when an automaton is in l . When a location is committed defines that an automaton must leave the location immediately. Committed locations are used to create atomic sequences.

An edge in the automaton can have the following annotations as shown in Figure 5.4: selects, guards, synchronizations and updates. A *select* non-deterministically



chooses a value after a transition in a bounded range which can be used as variable in the other annotations. A *guard* specifies when a transition over the edge is allowed by means of a boolean expression or integer bounds on clocks. The *synchronizations* annotation can be used to define synchronization of transitions in different automata. Finally, an *update* states integer expressions of which the outcomes are assigned to variables after a transition on the edge is taken.

A system can be defined as a composition of extended timed automata. Synchronization between these automata can be defined with channels or global variables. A channel d has an emitting edge labeled $d!$ and potentially has multiple receiving edges labeled $d?$. These receiving edges block until an event is received. Only when a transition on both sides of the channel is enabled, a transition will occur simultaneously across both edges. For an urgent channel the transitions cannot be delayed and must occur immediately once it is enabled. A broadcast channel can have multiple receiving edges simultaneously synchronizing to one emitting edge. However, a transition across the emitting edge may occur without one of the receiving edges being enabled. A transition over an edge can also update a global variable. A transition over an edge in another automaton can be triggered when it contains a guard function returning a boolean which depends on the value of that global variable.

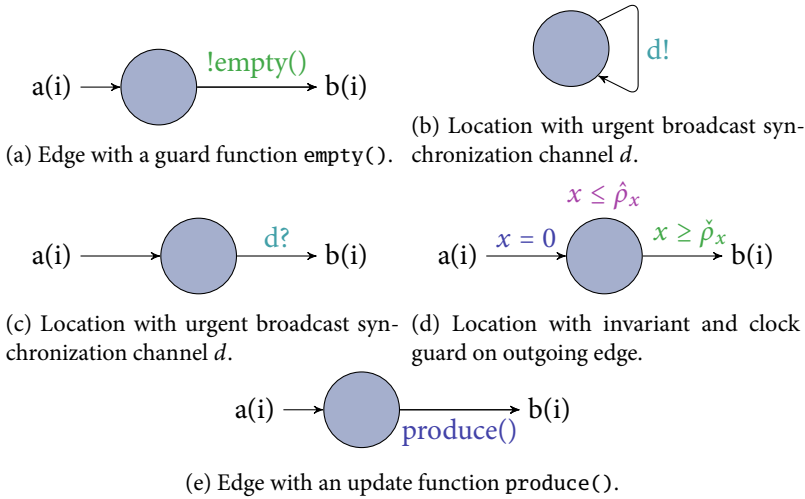
5.5 TIMED AUTOMATA MODEL OF HSDF^a GRAPHS


In this section we describe the derivation of an extended timed automaton that is semantically equivalent to an HSDF^a graph. A complete HSDF^a graph is composed as a network of timed automata. First, timed automata are defined for the two HSDF^a subgraphs presented in Section 5.2. Using this, we describe the construction of a behavioral equivalent network of timed automata of an HSDF^a graph. Finally, we discuss the consequence of that clock constraints of timed automata must be integer values.

5.5.1 UPPAAL COMPONENTS

The max-plus equation in Equation 5.1 uses the \oplus operator and describes a dataflow actor with multiple inputs. Waiting until all inputs are available can be implemented using a function `empty()` as a *guard*, which is used to check whether there are sufficient input tokens. The corresponding timed automaton is shown in Figure 5.5a. Taking the transition with the guard corresponds to the start of a firing of an actor.

Transitions in UPPAAL are not guaranteed to be taken immediately when enabled. A transition without delay can be enforced on an edge using a *synchronization* by making the channel urgent if there is already a channel. Otherwise, a dummy broadcast channel should be added, see Figure 5.5b, of which the emitting side is always enabled. The receiving side of the channel, see Figure 5.5c, will always take the transition immediately.



 Figure 5.5: Uppaal models

The addition of the firing duration of an actor can be incorporated in a timed automaton using a combination of an *invariant* and a *guard*, as shown in Figure 5.5d. One location is added, which represents the firing of an actor. The transition towards that location resets a clock x which indicates the start of a firing. The invariant, $x \leq \hat{\rho}_x$, defines the upper bound time on how long it is possible to stay in the location. The lower bound on how long it is at least required to stay in the location is defined by the *guard*, $x \geq \check{\rho}_x$.

5.5.2 DATAFLOW EDGE MODEL


Fortunately, during its execution of a strongly connected HSDF^a model, it is by definition the case that the maximum number of tokens on a cycle can never increase. The reason is that each actor on a cycle consumes one token from the cycle each firing, but also produces one token on the cycle when it finishes the firing. As a consequence, the number of tokens on an edge can never be larger than the number of initial tokens on the cycle to which the edge belongs.

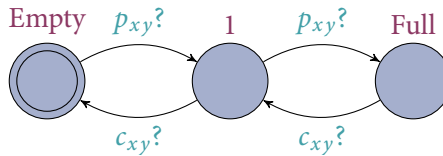
Global variables are used in UPPAAL to represent the tokens on the edges of a strongly connected HSDF^a model. An array of booleans is defined in UPPAAL using these variables, where the size of the array is equal to the maximum number of tokens that can accumulate on the edge. In a strongly connected dataflow graph, the maximum tokens on an edge is always bounded by the number of initial tokens in the cycle(s) an edge is part of. The index is always incremented modulo MAX_TOKENS, such that a fixed sized array can be used to store the indices of tokens present on the edge. Additional information about the tokens on an HSDF^a edge, such as their value, can be stored in a similar array. Together these variables


```

1 typedef int [1, NUM_ACTORS] actor_t;
2 typedef int [1, NUM_EDGES] edge_t;
3 typedef int [0, MAX_TOKENS-1] index_t;
4 typedef struct {
5     bool array[index_t];
6     actor_t from;
7     actor_t to;
8 } edge_t;
9 bool empty(edge_t &edge, index_t idx) {
10     return not edge.array[idx];
11 }
12 void produce(edge_t &edge, index_t idx) {
13     edge.array[idx] = True;
14 }
15 void consume(edge_t &edge, index_t idx) {
16     edge.array[idx] = False;
17 }
18 void consume_actor(actor_t act, index_t idx) {
19     for(e : edges_t){
20         if(edges[e].to == act) {
21             consume(edges[e], idx);
22         }
23     }

```

 Figure 5.6: Uppaal queue and access functions declarations.



 Figure 5.7: Timed automaton of an initially empty edge e_{xy} with a maximum capacity of 2 tokens.

model edges, which can be manipulated using access functions, as shown in Figure 5.6. A token with index i is produced on edge using the `produce(edge, i)` function, which is used as an *update* as shown in Figure 5.5e. This function sets the boolean at the index in the array to `True`. A token is consumed using the `consume(edge, i)` function, which sets the boolean at the index in the array to `False`. The `empty(edge, i)` function returns the inverse of the content of the array at the index i . The number of initial tokens on the edge determines the initial state of the array.

For illustration purposes an automaton of an HSDF edge, for which it is guaranteed that tokens arrive and are consumed in index order, is shown in Figure 5.7. Our implementation of the model of a dataflow edge in UPPAAL uses global variables and takes indices into account such that reordering is supported but cannot be depicted intuitively. The number of locations in the automaton of an edge is equal to the maximum tokens on the edge plus one additional location to indicate that there are no tokens. The initial location in the automaton that models a dataflow edge

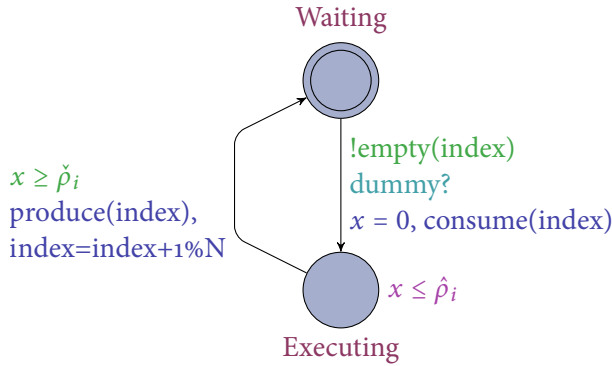
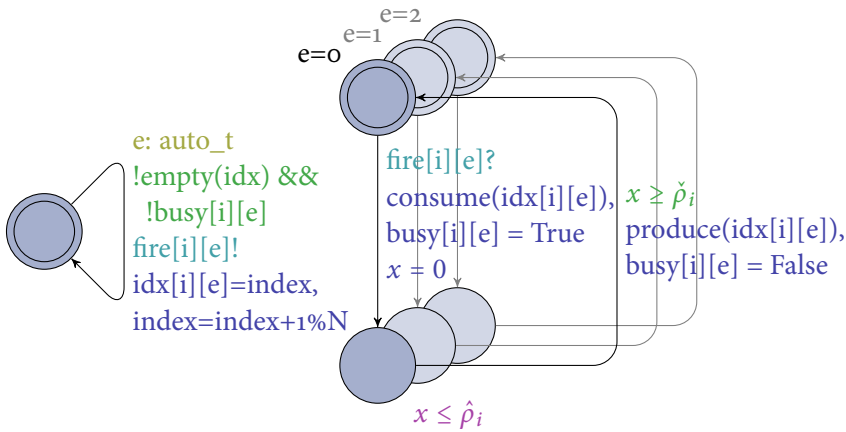
(a) Automaton modeling an actor v_i that cannot fire auto-concurrently.(b) Bounded auto-concurrency in UPPAAL for actor v_i .

Figure 5.8: Actor models in UPPAAL

is derived from the number of initial tokens on that edge. Urgent synchronization channels are used to model the instantaneous production and consumption of tokens.

5.5.3 ACTOR MODEL

A timed automaton for a dataflow actor can now be composed based on the max-plus equations in Section 5.3. The template for an actor that does not fire auto-concurrently is shown in Figure 5.8a. Two locations and one clock are used: one to represent a state in which the actor is waiting until sufficient tokens with the required index are present on all incoming edges, and one in which the actor is executing. The waiting location is left immediately when all incoming edges contain tokens, which is ensured by means of the guard function `empty(...)` and



the urgent channel dummy. Tokens are removed from the input edges using the `consume(..)` function. Internally, the `consume(..)` function calls the function `consume_actor(..)` function, as defined in Figure 5.6, which consumes tokens from all incoming edges of the actor. The same principle is applied for the `empty(..)` and `produce(..)` function to extend their use to multiple edges. In the executing location the firing duration of the actor is modeled using a combination of an *invariant* and *guard*. Finally, after the firing duration, a token is produced on all corresponding outgoing dataflow edges using the function `produce(..)`. Moreover, the index is incremented modulo the maximum number of tokens on the edges.

Auto-concurrency can only be modeled in UPPAAL in case an upper bound N on the number of actor replications that can execute concurrently is known. In that case, we replicate a modified version of the automaton in Figure 5.8a N times in the UPPAAL model. Fortunately, in strongly connected HSDF^a graphs, there are per definition never more tokens on a cycle than the number of initial tokens. As a consequence, also the number of replicas of an actor that can fire concurrently is never larger than the minimum number of initial tokens on the cycles to which the actor belongs.

The replications of an actor are controlled by an additional master automaton, as shown in the left part of Figure 5.8b. The master maintains the current index as its state, such that tokens can be consumed in the order of this index. Using the function `empty(index)` as a *guard*, the automaton determines if the actor is able to fire. In case the actor is enabled, the combination of the *select* and an additional *guard*, non-deterministically selects one of the replications of the actor that is not currently busy. The selected replication e is fired using the urgent *synchronization* channel corresponding to the actor i and replication e . The master also updates the index variable `idx[i][e]` for the selected replication, such that it consumes and produces tokens with the correct index. Mind that the update on the master, with the emitting edge of the synchronization channel, is performed before the update of a replication on the receiving edge. This allows the master to set the index of the replication. The master finally increments its index modulo N , such that the index variable always remains finite.

Each concurrent replication, as shown in the right part of Figure 5.8b, has a variable to keep track of the index of the next token that it should consume and produce. This index is set by the master when the replication of the actor fires. The functions `produce(..)` and `consume(..)` make use this index, to produce and consume tokens with the correct index. Another variable, `busy[i][e]`, tracks if the replication is currently firing. This variable is set when an replication fires, and is cleared when the firing is finished. The busy variable allows the master to start a new firing on the replication when it is not executing.

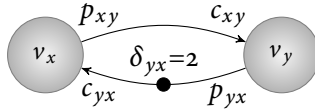


Figure 5.9: Cyclic HSDF^a graph.

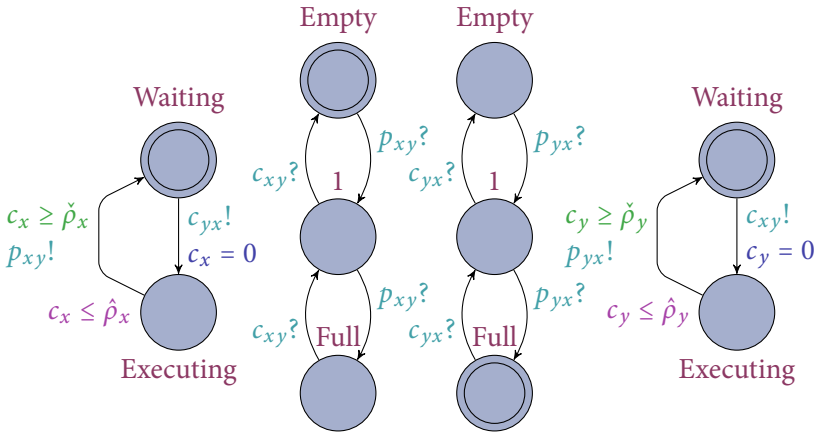


Figure 5.10: UPPAAL model for HSDF^a graph in Figure 5.9 with four automata, two for the actors and two for the edges.

5.5.4 COMPLETE AUTOMATON OF AN HSDF^a GRAPH

A complete timed automaton can now be defined in UPPAAL for an HSDF^a graph by instantiating components in the automaton for each element of the dataflow graph. For each actor v_i the actor template is instantiated. The edges e_{ij} in the dataflow graph are modeled using global variables and access functions. The automaton obtained given the HSDF^a graph in Figure 5.9 is depicted in Figure 5.10. For illustration purposes the dataflow edges are represented using automata that do not consider token reordering.

To model a source, the timed automata model as described in [HV06] is used. This source can generate bursts of events and is shown in Figure 5.11a. The source is defined by its period P and maximum jitter J . The jitter J can be larger than P . A similar automaton, as the one described in [HV06] to measure WCRT, is used for measuring the end-to-end latency. We create a separate automaton instead of including it in the source to measure the maximum time between two events as is shown in Figure 5.11b. Also burst of events are supported by choosing one start event from the channel *start* to track and storing an identifier in the variable n . The clock r used to track the latency in the location *Measuring* until the corresponding event occurs at the output by means of the channel *finish*. In the initial location, the clock r is reset every time step to limit the state space.

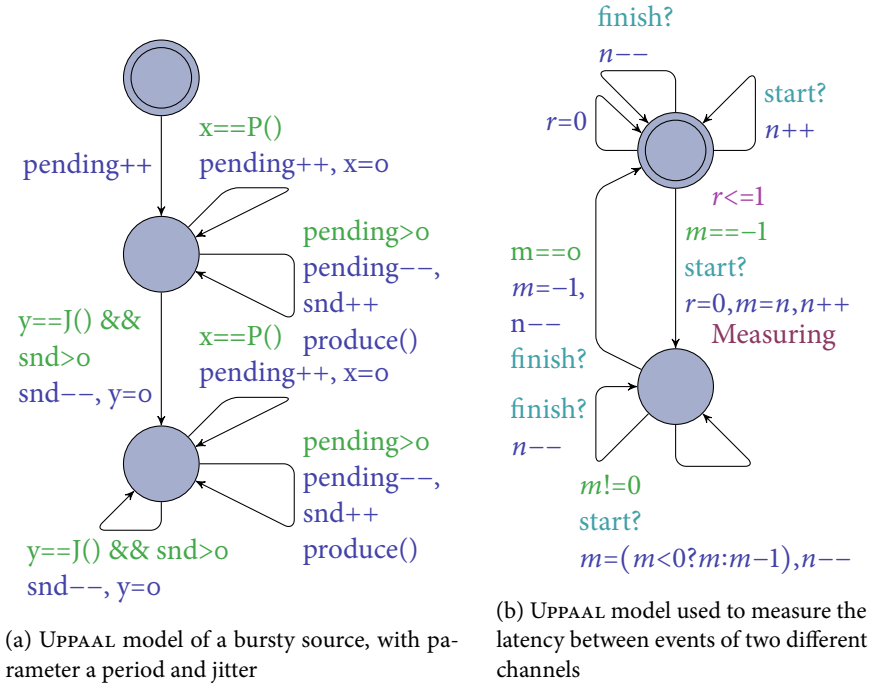


Figure 5.11: Miscellaneous UPPAAL models

5.5.5 INTEGER CLOCK CONSTRAINTS

In the extended timed automata used in UPPAAL all constraints in guards and invariants related to clocks must be integer to be able to create a finite number of similar states. Therefore, real-valued upper- and lower bounds on the firing durations in a dataflow model cannot be represented in a timed automaton without rounding. Increasing the worst-case firing duration and decreasing the best-case firing duration in the automaton of the actors is allowed because it will increase the set of behaviors that can be generated by the automaton that describes the complete HSDF^a graph and therefore can only result in an increase of the computed worst-case end-to-end latency.

5.6 CASE STUDY

In this section we compare latency analysis using an UPPAAL model with a state-of-the-art dataflow based analysis technique. We consider dataflow models in which each actor contains an internal non-deterministic finite state machine which determines the maximum firing duration of the actor in each iteration. These firing durations are not always equal to the worst-case firing durations of the actor and therefore the workload of each actor varies. In [H⁺13a] an approach that captures

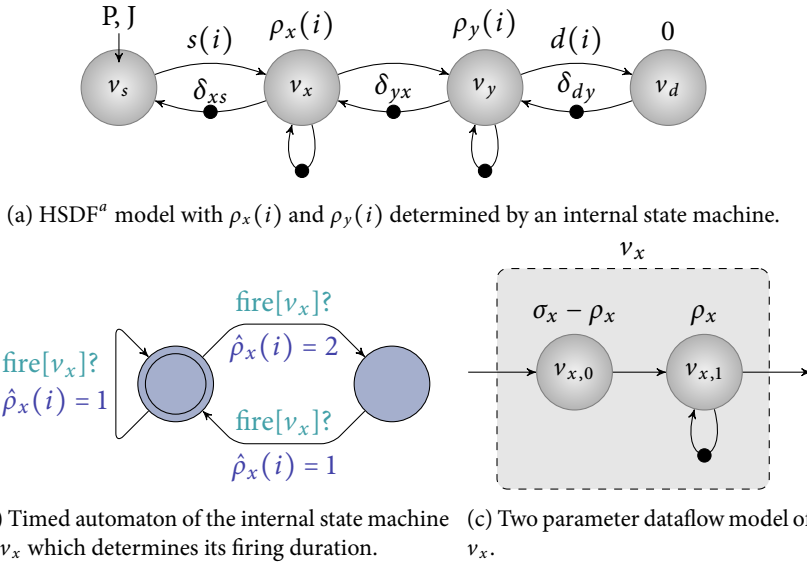


Figure 5.12: HSDF^a model used in the case study, internal state machine of the actors, and deterministic two parameter workload model for HSDF^a actor v_x .

a varying workload in a deterministic dataflow model is described. This enables the use of computationally efficient latency and throughput analysis techniques. In this case study, we use the analysis method presented in [KHB16c] which makes use of Linear Programs (LPs). Moreover, we also analyze the latency of a WLANp receiver application.

The HSDF^a graph that we first consider in this section is shown in Figure 5.12a and consists of the actors v_x and v_y , a source v_s , and a destination v_d . The source is characterized by its period P of 4 ms and jitter $J \in \{0, 4, 8, 16\}$ ms, which allows for bursts of tokens to arrive on $e_{s,x}$. The maximum number of tokens that can arrive at the same time on $e_{s,x}$ follows from these parameters and equals $\lfloor J/P \rfloor + 1$. Both actors have a firing duration which depends on the internal state of the actors. In this example we consider actors with two internal states, which results in a firing duration of 1 ms in one state, and a firing duration of 2 ms in the other state. It is possible to remain for a number of iterations in the first state, but the second state is always switched to the first state immediately. It is therefore not possible to have two consecutive firings with a firing duration of 2 ms. Such an automaton models for example the sporadic execution of an additional code-segment inside a task. We guarantee that there are sufficient initial tokens δ_{x_s} and δ_{d_y} such that v_s and v_y are never delayed by the lack of tokens on e_{x_s} or e_{d_y} . We want to analyze the maximum latency $L_{s,d}$ from the source to the destination of this HSDF^a graph.

A more accurate workload characterization can be created for actor v_x than non-deterministically selected firing durations in an interval, by making use of the

Table 5.1: Latencies obtained using UPPAAL and using a deterministic HSDF^a model for two values of δ_{yx} .

P (ms)	J (ms)	$\delta_{yx} = 1$			$\delta_{yx} = 2$		
		run-time (ms)	L_{sd} (ms)	$L_{sd}[\text{H}^+13\text{a}]$ (ms)	L_{sd} (ms)	$L_{sd}[\text{H}^+13\text{a}]$ (ms)	L_{sd} (ms)
4	0	0.02	4	4	4	4	4
4	4	0.15	6	8	5	5.5	6
4	8	0.47	10	12	8	8	8
4	12	1.24	12	16	9	9.5	10
4	16	3.06	16	20	12	12	12

additional information about the state machine inside the actors that determines the firing duration. Such a method is described in [H⁺13a], which requires that two parameters are determined: a guaranteed throughput ρ and latency σ . For both actors v_x and v_y the minimum throughput is obtained when switching happens continuously between the two internal states. This results in a guaranteed throughput of two tokens per 3 ms, therefore, $\rho = 1.5$ ms. The parameter σ in the model describes the maximum latency and is equal to 2 ms. Using these two parameters, a more accurate dataflow model can be created for v_x using two actors $v_{x,0}$ and $v_{x,1}$ as shown in Figure 5.12c. One actor has a self-edge with a single token and a firing duration equal to ρ_x , whereas the other actor has a firing duration of $\sigma_x - \rho_x$. Together, these two actor represent the throughput and latency constraints as described by the two parameters.

The additional information about internal states can also be incorporated in a timed automaton. Figure 5.12b shows an automaton which determines $\rho_x(i)$. Every time v_x fires the state is updated based on the urgent channel fire. This channel replaces the channel dummy in the actor model of Figure 5.8a. The value of $\hat{\rho}_x$ in the actor model is made equal to the firing duration $\hat{\rho}_x(i)$ as indicated by the internal state machine. The value of $\check{\rho}_x$ is set to 0.

The computed latencies for different jitters are presented in Table 5.1. The results obtained using timed automata are in the fourth and sixth column. The results obtained using the two parameter dataflow model are in the fifth and seventh column. The table shows that more accurate results are obtained using timed automata for $\delta_{yx} = 1$, for a jitter of 4 and 12 ms and equal results for the other jitter values. Adding an additional token on e_{yx} improves the result of the LP based dataflow analysis method. All results were computed within 1 ms using this method. However, model-checking still results in more accurate analysis results. Furthermore, it can be seen that the run-time of the model checker is rapidly increasing for larger jitter values despite that this example considers a very small problem instance. The negligible run-time and its better scaling of the LP based dataflow analysis method makes it more suitable for larger graphs.

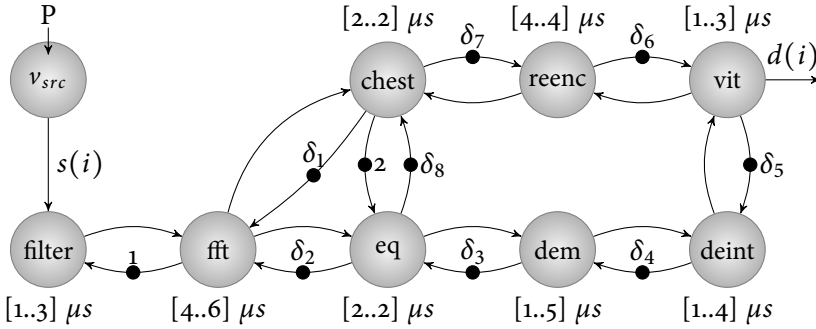


Figure 5.13: Dataflow graph of a WLAN 802.11p transceiver application. All actors have an implicit self-edge with a single token.

The last column of Table 5.1 contains the results obtained for the case that the internal state machine of the actor v_x is ignored and non-deterministically selected firing durations in the interval $[\hat{\rho}_x, \hat{\rho}_x]$ are used instead. These results are the most pessimistic ones and are the same for both analysis methods.

An HSDF^a model of a WLAN 802.11p application is presented in [KHB16c]. Figure 5.13 shows this HSDF^a model which consists of a source with a period of 10 μ s, 8 actors and 19 edges connecting them. The number of initial tokens, $\delta_1 \dots \delta_8$, are all set to 3. An automaton per actor, like the one in Figure 5.12b, determines their firing durations. UPPAAL is used to calculate L_{sd} which is equal to 21 μ s and is computed in 435 s. A latency of 23 μ s was obtained with the LP based dataflow analysis method within 1 ms. The run-time of UPPAAL can be reduced at the cost of accuracy by removing some of the automata and instead using actors with non-deterministically selected firing durations from their interval. Replacement is allowed because it results in that more behaviors are considered during analysis. The run-time is 3 s and latency 23 μ s in case all actors are replaced.

5.7 CONCLUSION

In this chapter we presented a behavior-preserving transformation of strongly connected HSDF^a graphs into timed automata. These timed automata allow for the computation of exact end-to-end latencies because the correlation between the firing durations of different firings is taken into account.

The transformation of HSDF^a graphs into a behaviorally-equivalent timed automata is possible because the number of tokens on edges in a strongly connected HSDF^a model is bounded. Therefore, buffers can be modeled using the extended timed automata of UPPAAL which by definition have a finite number of states. This also guarantees that there is a maximum number of replicas of the same actor that can fire concurrently. This guarantees that there is a finite number of concurrent state

machines, and thus states, required to model each HSDF^a actor.

In the case study we consider two HSDF^a examples for which exact end-to-end latency analysis results are obtained using timed automata and UPPAAL, whereas this is not possible using deterministic timed-dataflow models. This comes at the cost of a higher run-time which for the considered examples is less than 435 s.



HYBRID LATENCY ANALYSIS

ABSTRACT – Current latency analysis approaches for real-time multiprocessor applications often have a low accuracy or high computational complexity. Several approximative analysis approaches exist that can analyze the latency efficiently. However, these approaches often produce too pessimistic latency results and do not exploit buffer sizing nor exploit additional sequence constraints to reduce the latency. More accurate latency analysis results can be obtained using model checking of timed automata, however, potentially at the cost of an excessive run-time. This chapter presents a latency analysis approach for cyclic task graphs that combines model checking and timed-dataflow analysis. The approach is applicable for systems in which tasks are executed on shared processors using an FPP scheduling policy. The reduction in run-time is achieved by pruning the search space that needs to be analyzed using the model checker by making use of approximative dataflow analysis techniques. The approach exploits dimensioning of buffers to minimize interference and latency. Moreover, sequence constraints are introduced and automatically adapted in order to minimize the latency.

6.1 INTRODUCTION

Cyclic dependencies in real-time multiprocessor applications complicate the analysis of these applications, especially when FPP scheduling is involved. However, these dependencies can also have beneficial effects on latency. These *cyclic dependencies* can for example be a result of: feedback loops, a static task execution order, clustering of tasks [FKH⁺08] or bounded size FIFO buffers [WGH_B15]. As a result

This chapter is based on [GK:5].



of these cyclic dependencies, tasks on these cycles can only interfere a bounded number of times with each other. In [LMB⁺14], these dependencies are used to calculate upper bounds on the interference of tasks sharing the same resource. The maximum number of times tasks can interfere with each other can for example be controlled by the size of the buffer between tasks [WGH15]. Lower interference between tasks can have a positive effect on the maximum latency. But, as we show in Section 6.7, minimal buffer sizes given a throughput constraint imposed by a periodic source, do not always result in the minimal latency.

Buffer sizing cannot be used to control the interference between tasks if these tasks do not communicate. Moreover, sizing of buffers cannot delay the start of a consuming task. To provide more control over the schedule freedom we introduce sequence constraints with initial tokens between tasks. The interference between tasks, and as a result the maximum latency of the task graph, can be reduced by carefully controlling the number of initial tokens. We will show that in some cases a negative number of initial tokens is required for achieving the minimum latency. The number of initial tokens on sequence edges can be derived with a similar algorithm as is used for buffer sizing.

Several approximative analysis approaches exist that can calculate the latency for real-time multiprocessor applications, including the effects of FPP scheduling. However, well known approaches like MAST [HGGM01] and SymTA/S [HHJ⁺05] are not able to analyze arbitrary cyclic dependencies. Therefore, these approaches are not able to optimize the latency by exploiting cyclic dependencies. The iterative dataflow analysis approach in [KHB16c] is able to handle arbitrary cyclic dependencies and uses a buffer sizing technique, which can result in a latency reduction. This approach does not consider all possible buffer sizes, since otherwise monotonicity and convergence of the used analysis flow cannot be guaranteed.

Dataflow analysis techniques have shown to be suitable for the analysis of cyclic applications. Processor sharing, however, cannot be modeled explicitly and can only be included by deriving an over-approximation bound on the interference of tasks. There are techniques to prevent interference between tasks that produce more accurate analysis results by making tasks mutually exclusive as discussed in Chapter 4. However, making tasks mutually exclusive does not always result in the minimum latency. As shown in Chapter 5, exact latency analysis for dataflow graphs without pre-emptive scheduling is possible using timed automata if the execution times are natural numbers.

Timed automata are a popular formalism for modeling and analyzing real-time systems. Although latency analysis is possible, there is no computationally efficient build-in support for optimizing of costs like, buffer sizes, or the introduction of additional sequence constraints, to reduce latency. The problem of finding the minimum latency in case of cyclic dependencies is non-monotonic. Therefore all configurations of buffer sizes have to be analyzed. Moreover, without a combination of timed automata and an approximate analysis technique to upper bound and reduce the number of configurations, many of these configurations need to be

analyzed leading to a high run-time.

This chapter presents a latency minimization approach with a reduced run-time for cyclic task graphs executed on shared multiprocessor systems using FPP task scheduling. The approach identifies the configuration of buffer capacities and sequence constraints that result in the minimum latency. The approach uses approximative, but computationally efficient, timed-dataflow analysis techniques for removing all configurations from the search space that cannot result in the lowest latency since these will deadlock, won't reach the throughput of the periodic source, or impose redundant constraints. To find the configuration with the lowest latency the remaining options are evaluated using a model checking approach.

This chapter is structured as follows. In Section 6.2, we discuss related latency analysis approaches for task graphs using FPP scheduling. The basic idea behind our hybrid analysis approach is presented in Section 6.3. In Section 6.4, extended timed automata are used to create a model of task graphs. Model checking is performed on these models to analyze latency. The dataflow model is introduced in Section 6.5, which is used in analysis techniques that reduce the number of configurations that need to be analyzed. Both these timed automata and dataflow-based analysis techniques are combined in our analysis flow as described in Section 6.6. In Section 6.7, we introduce sequence constraints to reduce latency. The case study is presented in Section 6.8. We state our conclusions in Section 6.9.

6.2 RELATED WORK

In this section we present work related to latency analysis of task graphs where tasks are scheduled using FPP scheduling. We first discuss related approximative analysis approaches, which do not perform buffer sizing to reduce latency. Then we discuss both analysis approaches that use timed automata, and decidability of scheduling problems using timed automata. Finally, we discuss an approach that does perform buffer sizing.

The SymTA/S approach [HHJ⁺05], is one of the techniques which overestimates interference between tasks in order to analyze latency. The approach, however, is not able to handle arbitrary cyclic dependencies. MAST [HGGM01] is another approach that can derive a more accurate characterization of the interference. It is limited to the analysis of acyclic graphs. The MPA-RTC [TCG⁺01] analysis approach has been extended to support either cyclic data dependencies [TS09], or cyclic resource dependencies [JPTY08]. The combination of cyclic resource and data dependencies is not supported. Neither of these approaches use buffer sizing techniques to reduce latency, nor introduce additional sequence constraints to reduce latency.

Timed automata based analysis approaches can be used to determine the latency of a task graph. A number of these approaches only considers a fixed (worst-case) execution time of tasks [HVo6, MLR⁺10, PWT⁺07]. This ignores additional interference between tasks that can occur when tasks finish earlier. The TIMES-tool



[AFM⁺03, FMPY06] does consider BCETs and WCETs. We use their method of summing up the time needed to finish all released tasks to determine when a task finishes its own execution, which is described in more detail in Section 6.4. However, TIMES only considers the single processor case. The multiprocessor case is considered in [DILS09, BHM08], but these approaches are limited to acyclic task graphs. Moreover, none of these timed automata based approaches exploit buffer sizing of blocking buffers to reduce latency.

Decidability of scheduling problems using timed automata is discussed in [KY04]. This paper shows that the problem of deciding whether the deadlines are met of tasks in a task graph that is scheduled using fixed priority scheduling on a single processor, is in general undecidable. Therefore also the exact latency of an arbitrary task graph on shared resources can not be computed. More precisely the problem to verify whether the tasks meet their deadlines is undecidable if the following three conditions hold:

1. The execution times of the tasks are characterized by an upper and lower bound on a continuous time interval.
2. Tasks can announce their completion time to other tasks at every point in continuous time.
3. Each task can preempt another task at any point in time, i.e. not only at clock cycle boundaries.

The problem is not a decision problem anymore if the result of the analysis can be besides the yes/no answer also the answer that may-be the tasks meet their deadlines. Over-approximation during analysis introduces this third option and this is what has been used in TIMES [AFM⁺03, FMPY06] and is also used in this chapter. The approach in [DILS09] makes use of stopwatch automata which are in general undecidable. In UPPAAL 4.1 techniques are introduced that apply over-approximation during analysis in case stopwatches are applied in the timed automata model. This over-approximation has as a positive side-effect that the run-time is reduced. In this chapter we also introduce a timed automata model in which stopwatches are used. This enables a comparison with the case that a timed automata model without stopwatches is used. In [MDA09] it is not assumed that tasks can finish at every point in time. This makes the schedulability of the decision problem decidable but results in an impractically large automata model if it is considered that tasks can be preempted at every clock cycle boundary.

The approach in [KHB16c] minimizes buffer sizes given a throughput constraint. However, it does not minimize the latency at the same time. We do make use of this approach in our hybrid analysis flow to find safe bounds on buffer capacities, by analyzing the buffers as non-blocking buffers. This approach over-approximates interference. Moreover, a small generalization had to be made to be able to compute the number of initial tokens on sequence constraints since sequence constraints are a more general version of blocking buffers.

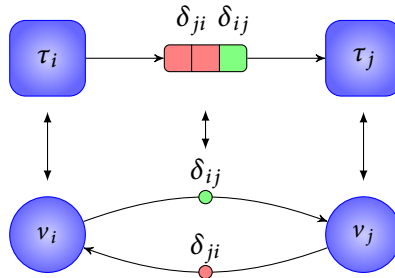


Figure 6.1: One-to-one relation between a task graph containing a blocking buffer and a dataflow graph.

6.3 BASIC IDEA

The idea behind our analysis approach is presented in this section. We start by showing the relation between a task graph containing a blocking buffer and a dataflow model of it, where cyclic dependencies are present. Then, we show how these cyclic dependencies can influence interference between tasks. Finally, we indicate how we can minimize latency using these cyclic dependencies.

The dependencies in a task graph have a one-to-one relation with the edges in a dataflow graph. This is illustrated in Figure 6.1, where two tasks, τ_i and τ_j , communicate using a blocking buffer. Initially this buffer contains one full container δ_{ij} and two empty containers δ_{ji} . A task can only execute and write to such a buffer after it first obtained an empty container. After its execution, it produces a full container, which can be read by the consuming task. In the dataflow graph, the initially full containers of the buffer are represented as tokens on the edge from actor v_i to v_j , which correspond to the tasks, whereas the tokens on the edge from v_j to v_i represent empty containers.

Scheduling freedom is affected by these cyclic dependencies that model blocking buffers. Based on these cyclic dependencies we can derive an upper bound on the number of times tasks can be pre-empted using the FPP scheduling policy. The number of times τ_i can be pre-empted by τ_j is therefore bound by $\delta_{ij} + \delta_{ji}$. Therefore we can conclude that interference between tasks can be controlled by choosing the number of containers in the buffer, e.g. by using buffer sizing techniques.

The relation between buffer sizes and latency for a graph is non-monotonic [WGHB15]. Therefore, in order to obtain the minimum latency for a task graph, we have to verify latency for all possible buffer sizes of all buffer, which we call *configurations* in this chapter. In order to obtain latency results for this unstructured analysis problem, model checking is performed. An upper bound on the size of each buffer can be calculated using approximative analysis techniques. The upper bound represents the situation where the schedule of tasks is not influenced by the size of buffers because there are always sufficient empty containers. As a



result the write of a buffer will never be blocked and thus delayed because there is always space in the buffer. The lower bound on the buffer size is determined by the number of full containers. In case the buffer size is equal to one, a fixed execution order of the tasks is enforced. Such an order is called a *static-order schedule*. Timed automata of the task graph are generated for all possible buffer sizes between these computed lower and upper bounds. These timed automata are then analyzed using UPPAAL to find a configuration with the minimum latency.

6.4 TIMED AUTOMATA

A template-based timed automata model is constructed in this section to allow different configurations of a task graph to be verified automatically using UPPAAL. We use the extended timed automata model which is already introduced in Section 5.4. Using these extended timed automata, we define parameterized templates for the different components in a tasks graph¹. The maximum latency of a task graph is then verified using UPPAAL for a network of timed automaton instances.

Based on these extended timed automata we will now define a template for tasks, schedulers, and FIFO buffers. Given these templates, a network of timed automata is created for a specific configuration of a task graph where the buffer sizes are configured and a number of tasks and processors are instantiated. Each configuration will therefore result in a unique network of timed automata.

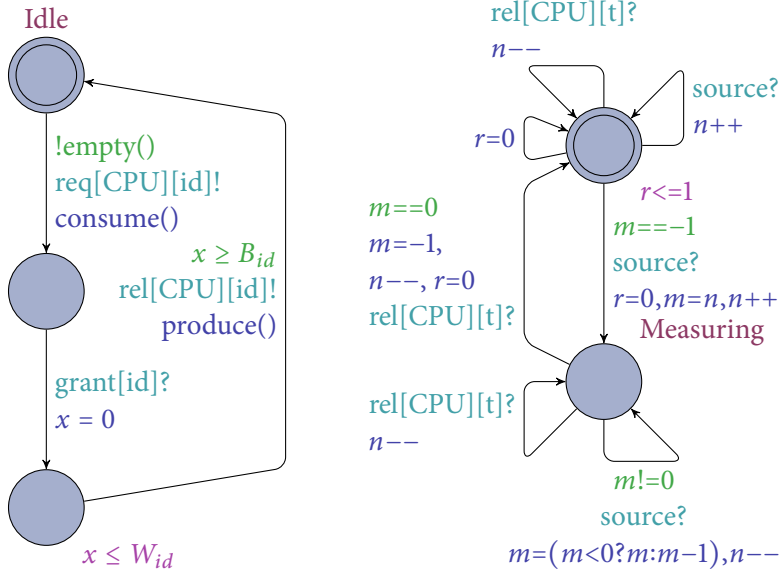
6.4.1 FIFO BUFFER

A FIFO buffer is modeled in extended timed automata using global variables and access functions to manipulate and check these variables, as described in Subsection 5.5.2. Each buffer is represented by two edges, each containing the producer task, consumer task, and a number of tokens on the edge. The number of tokens is initialized with the number of full or empty containers in a buffer. The function `empty()` is used to check if there are insufficient tokens, e.g. < 1 . Tokens are produced using `produce()`, which increments the number of tokens by one. The number of tokens is decremented by one using `consume()`.

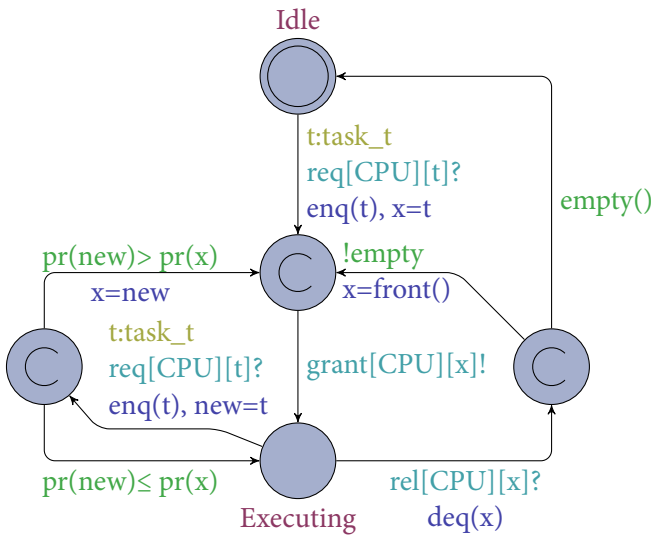
6.4.2 TASK TEMPLATE

The timed automaton of a task consists of three locations as shown in Figure 6.2a. The transition from the initial location, *Idle*, is guarded by the expression which checks if there are sufficient tokens on *all* its incoming edges. The urgent broadcast channel *req* is used to signal the processor that the task is ready to execute and tokens are consumed from all incoming edges using `consume()`. In the second location, time passes until execution on the processor is granted using the urgent broadcast channel *grant* on a specific processor, CPU, to which the task is statically assigned. During the transition to the 3rd location, the clock *x* is reset. This clock

¹Available at <https://github.com/gkuiper/UppaalWLAN>



(a) Timed automaton template for a task id. (b) Timed automaton used to measure latency between the source and sink t.



(c) Timed automaton template for a processor CPU using FPP scheduling.

Figure 6.2: Timed automata templates for different components in a task graph.

represents the execution time of the task. This location can be left when the clock is between B_{id} and W_{id} which are equal to the BCET and WCET. During the



transition to location *Idle* tokens are produced and the processor is signaled about the completion of the task using the broadcast channel *rel*.

Two options are implemented to incorporate the inference caused by pre-emptions of tasks with a higher priority. One option uses over-approximations of the finish times to guarantee decidability of the scheduling problem, the other option uses stopwatches.

Over-approximating pre-emptions

Pre-emptions can be over-approximated by deriving a lower and upper bound on the completion time of higher priority tasks. For each pre-emption the bounds of lower priority tasks, B_{id} and W_{id} in Figure 6.2a, are incremented respectively with the BCET and WCET of the task which triggered the pre-emption. This over-approximation on the finish time of the task enforcing the pre-emption ensures decidability.

Stopwatches

Using stopwatches, a more intuitive task template can be constructed in which the clock tracking the execution time is paused when a task is pre-empted. The processor must set a global variable `runs[CPU]` to indicate which task is currently executing on a processor. The invariant of the 3rd location in Figure 6.2a must be extended to also set the derivative of clock x to either 0 or 1 depending on the tasks that is executing, e.g. $x' == (\text{runs}[\text{CPU}] == id)$. The clock is stopped if $x' == 0$. The global variable `runs[CPU]` is set to the *id* of the task that is granted access to the processor and is reset when the task releases the processor.

6.4.3 PROCESSOR TEMPLATE

A processor template for the FPP scheduling policy is constructed using a task queue. This queue contains all tasks that have issued a request to indicate they are ready to execute. For FPP scheduling, the queue is sorted based on the priority of the tasks. The timed automaton consists of two locations where time can pass, *Idle* and *Executing*, and three committed locations used to create atomic sequences where time cannot advance. This timed automaton is shown in Figure 6.2c. From the *Idle* and *Executing* location, a transition occurs when a task issues a request via the *req* channel. In that case, the new task is enqueued, $enq(t)$. When the transition is taken from *Idle* the queue was empty and the task is immediately granted access to the processor using the channel *grant*. Otherwise, the priority of the currently executing task and the new task are compared. Based on the outcome, the current task can resume execution or is pre-empted and the new task is granted access to the processor. After a task finished its execution, the processor receives a signal on the *rel* channel and as a result, removes that task from the queue. The task in the front of the queue with the highest priority is then granted access, or the processor idles if the queue is empty.

The latency in a network of timed automata is measured using a separate timed automaton. The automaton as shown in Figure 6.2b measures the latency between an event produced by the source from channel *source* and the release event of the task producing the output for the sink, using the channel *rel*. The source, which produces events, is characterized by a period and jitter. Since multiple source events can be produced before a release event is produced that is a result of the first source event, a non-deterministic selection is made of an event that is tracked. This non-deterministic selection results in that UPPAAL will check all source events. Variable *n* tracks the number of outstanding events from the source and is incremented for each event from the source. This variable is copied to variable *m* when a source event has occurred and the transition to the location *Measuring* is made. Variable *m* tracks the number of remaining release events before the event corresponding to the source event is found. Release events belonging to previous source events are skipped and decrement *m*. Only when both $m == 0$ and a release event occurs, the matching release event is found such that the latency measurement is stopped by returning to the initial location. In the initial location, the clock *r* is reset every time unit since no measurement is taking place to reduce the state space. This is implemented using the invariant $r \leq 1$ and the edge with the update $r = 0$. The maximum latency can then be obtained by querying the suprema of clock *r* in the location *Measuring*: $\text{sup}\{\text{Measuring}\}: r$.

6.5 TIMED-DATAFLOW

Computationally efficient approximative analysis of task graphs can be performed using timed-dataflow analysis techniques. Based on the HSDF dataflow model, as described in Section 2.2, we derive techniques in this section to prune the number of configurations that need to be analyzed in our approach using timed automata. Finally, we describe dataflow-based approximative analysis method that we use to derive upper bounds on buffer capacities.

The dependencies of a task graph can directly be translated into an HSDF graph as discussed in Section 6.3. However, the effect of FPP scheduling cannot be modeled directly in HSDF graphs because HSDF graphs do not allow modeling of choice. Therefore, bounds on the response times of tasks are used as firing durations of the corresponding actor. As a consequence accurate analysis latency results cannot be obtained directly but these approximations can be used to prune the number of configurations that need to be analyzed using UPPAAL.

6.5.1 DEADLOCK

By definition, a HSDF graph deadlocks if it contains a simple cycle without initial tokens. We can therefore prune, i.e. discard, all configurations with simple cycles without initial tokens. This is especially useful after introducing additional con-

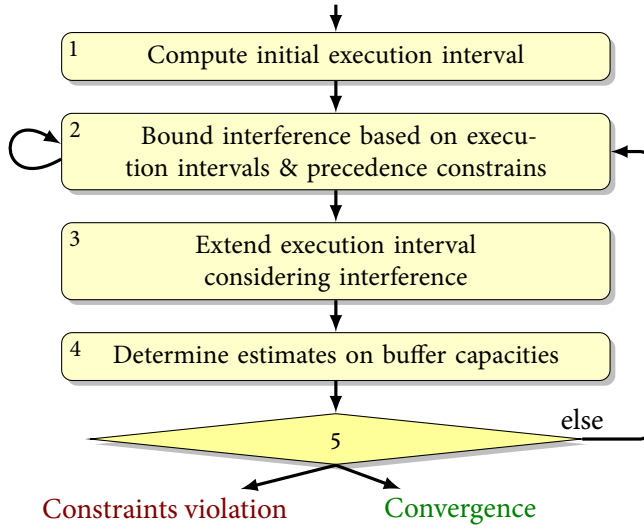


Figure 6.3: Analysis flow of the approximative dataflow analysis approach.

strains as presented in Section 6.7. UPPAAL can also verify the absence of deadlock, however, less computationally efficient.

6.5.2 MINIMUM GUARANTEED THROUGHPUT

Another method to prune the number of configurations is by considering the throughput constraint imposed by the source. The guaranteed throughput of HSDF graph must be greater than the throughput of the source to prevent the source from blocking as a result of a full buffer. We cannot determine the throughput of a dataflow graph accurately when FPP scheduling is involved. However, a lower bound on the guaranteed throughput can be determined by setting the firing duration of all actors to the WCET of the corresponding tasks, ignoring any interference. A computationally efficient MCR algorithm [Rei68] can then be used to calculate the MCR, which corresponds to the inverse of the minimum guaranteed throughput. All configurations where the minimum guaranteed throughput is less than the period of the source are discarded.

6.5.3 APPROXIMATIVE DATAFLOW ANALYSIS

The approximative dataflow analysis method we use is based on execution intervals of tasks [KHB16c]. The execution interval \mathcal{I}_i consists of the earliest possible start time and the latest possible finish time of a task τ_i . These intervals are determined from a best-case dataflow model using BCETs, and a worst-case dataflow model using WCETs and upper bounds on task interference.

The analysis flow of this approximative dataflow approach is shown in Figure 6.3. Initially, the execution intervals are determined without considering interference as shown in step 1 of the flow. Initial buffer sizes are chosen such that the graph is deadlock-free. In step 2, an upper bound on the possible interference between tasks is calculated given the execution intervals. The precedence constraints originating from the buffers in a task graph are included in this interference calculation. The execution intervals are extended in step 3 using the calculated interference. Buffer sizes for iteration $k \in \mathbb{N}$ are calculated based on the latest finish time of the consuming task τ_j , $\hat{\mathcal{I}}_j$, and earliest start of the producing task τ_i , $\check{\mathcal{I}}_i$:

$$\delta_{ji}^k = \max \left(\left\lceil \frac{\hat{\mathcal{I}}_j - \check{\mathcal{I}}_i}{P_j} \right\rceil, \delta_{ji}^{k-1} \right) \quad (6.1)$$

where δ_{ji}^k is the number of initial tokens on e_{ji} in iteration k and P_j is the period of the consuming task. The buffer sizes are not allowed to decrease compared to the previous iteration in order to guarantee termination of the analysis flow. Step 2 to 4 form an iteration of the analysis flow, which is repeated until a constraint violation occurs or the execution intervals and buffer sizes have converged, i.e. stay the same.

Since buffer capacities are determined in every iteration of this analysis flow, we refer to this method as iterative buffer sizing. Note that we cannot guarantee that the configuration of buffer sizes leading to the minimal latency of a task graph is considered using iterative buffer sizing.

The analysis flow can also analyze buffers without blocking write, i.e. non-blocking buffers. In that case, the buffer sizes are calculated such that there always is at least one empty container available for a task that is enabled, to prevent buffer overflows. While iteratively computing blocking buffer capacities is a means to limit interference in the analyzed application, iteratively computing non-blocking buffer capacities is merely another way to model maximum interference, without influencing the analyzed application. Representing blocking buffers as non-blocking consequently allows derivation of an upper bound on their capacity, which thereafter can be used to limit the number of states in UPPAAL.

6.6 HYBRID ANALYSIS

Our hybrid analysis approach, which combines analysis techniques from timed-dataflow and timed automata, will be described in this section. The hybrid analysis flow is shown in Figure 6.4, which will be described step-by-step. The flow uses a task graph as an input containing buffers, tasks, and a task to processor assignment.

In the first step, a lower and upper bound is derived on the capacity of all buffers. The upper bound is calculated using the approximative analysis technique as described in Section 6.5. This upper bound is determined by analyzing each buffer as non-blocking. The lower bound is set to the maximum of 1, which prevents deadlock, and the number of initially full containers in the buffer.

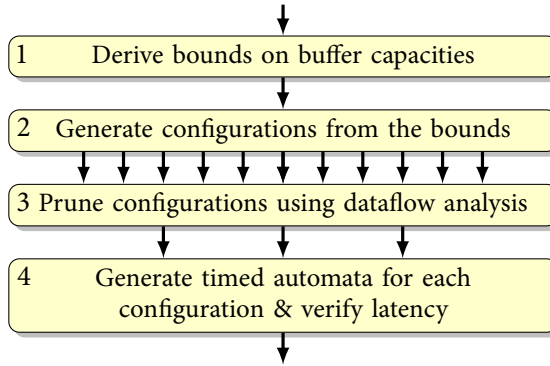


Figure 6.4: Hybrid analysis flow.

Once these bounds are determined, configurations of the task graph are generated in step 2. Since the latency is non-monotonic in the buffer capacities, we have to consider all possible sizes of each buffer within these bounds. The set of configurations therefore consists of the product of the number of buffer sizes of all buffers.

Many configurations are generated for which we can guarantee, using dataflow analysis techniques, that these will not result in the minimal latency. Step 3 performs pruning of the configurations using the deadlock and minimum guaranteed throughput techniques described in Section 6.5.

In the final step, a network of timed automata is created for each remaining configuration. As described in Section 6.4, a timed automaton template is initiated for all buffers, tasks and processors. The sizes of the buffers are fixed as specified in the configuration. The model checker UPPAAL is used to verify the maximum latency of the network of timed automata. Once all configurations are verified, the one resulting in the minimum latency is selected to finish the analysis.

In the next section we discuss the introduction of additional sequence constraints. The same hybrid analysis flow can be used for these constraints, since these constraints can be seen as a generalization of the constraints imposed by buffers.

6.7 SEQUENCE CONSTRAINTS

In this section we present a method that potentially reduces latency by introducing additional *sequence constraints*. We introduce pairs of these sequence constraints as cyclic dependencies between two tasks to have direct control over the maximum interference one of these tasks can cause on the other. This construct of sequence constraints can be seen as a generalization of a blocking buffer, with an important difference that it does not store data.

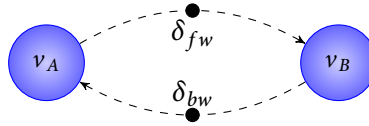


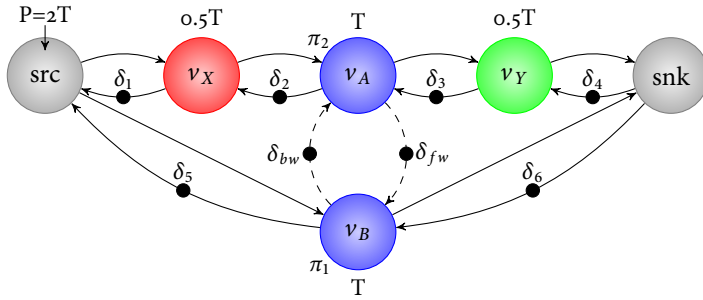
Figure 6.5: Dataflow graph where sequence constraints are inserted between v_A and v_B .

In general, buffers only allow their capacity to be changed in case it does not result in a change of the functional behavior of an application. The number of initially full containers in a buffer is fixed and therefore there is only control over the number of initially empty containers. As a result, changing the buffer capacity will only change the temporal behavior of an application and not the functional behavior (unless it deadlocks). The sequence constraints can be seen as a generalized buffer, it not only allows control over the number of initially empty, but also over the number of initially full containers. Adding these sequence constraints does not influence the functional behavior of an application because it only affects the schedule freedom and does not result in transfer of data.

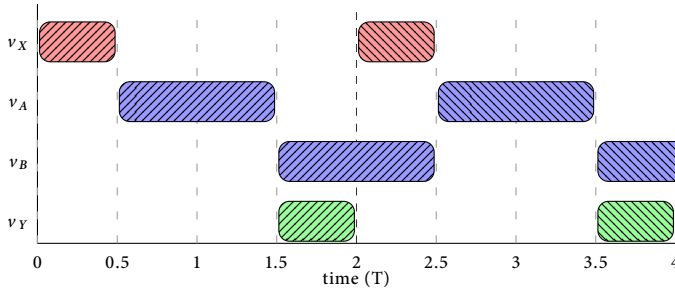
Formally, a sequence constraint between τ_A and τ_B contains a number of containers $\delta_{AB_{fw}}$ on the forward edge e_{AB} , and a number of containers $\delta_{AB_{bw}}$ on the backward edge e_{BA} . Let $s_A(i)$ be start of the i 'th ($i \in \mathbb{N}$) execution of τ_A , $f_A(i)$ the finish of the i 'th execution of τ_A and logically $f_A(i) \geq s_A(i)$. The sequence constraints introduce the following constraints: $s_B(i) \geq f_A(i - \delta_{AB_{fw}})$ and $s_A(i) \geq f_B(i - \delta_{AB_{bw}})$. Notice that a deadlock will occur when $\delta_{AB_{fw}} + \delta_{AB_{bw}} \leq 0$. In case $\delta_{AB_{fw}} + \delta_{AB_{bw}} = 1$, only one execution order is enforced between τ_A and τ_B , which therefore is a static execution order.

A one-to-one translation is possible from a sequence constraint to a corresponding dataflow model. The forward and backward containers of a sequence constraint are equivalent to initial tokens on the corresponding edge. An example of a dataflow model containing sequence constraints is shown in Figure 6.6a. The example consists of four tasks; τ_A , τ_B , τ_X and τ_Y . τ_A and τ_B share a processor and sequence constraints are therefore inserted between τ_A and τ_B . These tasks are converted to actors in the dataflow model. The edges corresponding to the sequence constraints are shown as dashed lines in the graph.

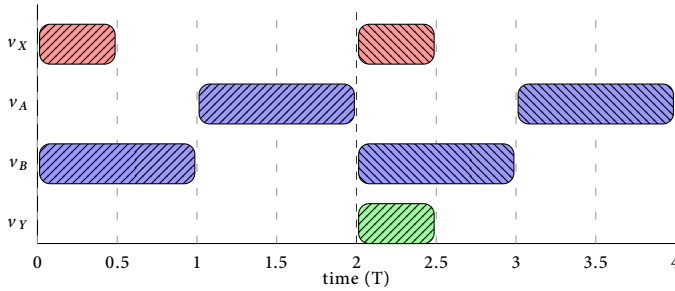
Since the sequence constraints can directly be expressed in a dataflow model, existing analysis techniques can be applied to it that determine the required number of tokens given a throughput constraint. The difference with buffer sizing for the number of empty containers is that, for each sequence constraint, the number of containers on two edges can freely be chosen. The approximative timed-dataflow approach in [KHB16c] is therefore extended to derive an upper bound on the number of initial tokens on both the forward as the backward edge using the existing buffer sizing algorithm. The hybrid analysis, as presented in Section 6.6, is therefore also applicable for graphs containing sequence constraints.



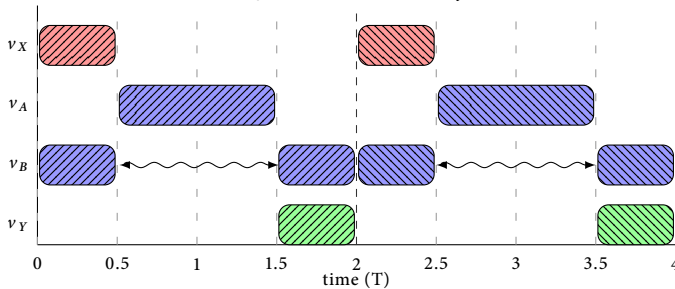
(a) Dataflow graph where a static execution order of v_A and v_B increases the end-to-end latency.



(b) Schedule of Figure 6.6a for which $\delta_{fw} = 0, \delta_{bw} = 1$.



(c) Schedule of Figure 6.6a for which $\delta_{fw} = 1, \delta_{bw} = 0$.



(d) Schedule of Figure 6.6a for which $\delta_{fw} = 1, \delta_{bw} = 1$.

Figure 6.6: Example where a static execution order of v_A and v_B increases the end-to-end latency.

We will use the examples in Figure 6.6a to illustrate that enforcing a static execution order between τ_A and τ_B by using sequence constraints will lead to a higher latency than when τ_A (with a high priority) can pre-empt τ_B (low priority). A static order schedule between these tasks can be created where τ_A must execute before τ_B by selecting $\delta_{AB_{fw}} = 0, \delta_{AB_{bw}} = 1$ as shown in the schedule in Figure 6.6b. The other option is to execute τ_B first, $\delta_{AB_{fw}} = 1, \delta_{AB_{bw}} = 0$ as shown in the schedule in Figure 6.6c. Both options result in a latency from source to sink of $2.5T$ as computed by our hybrid analysis approach. However, when we allow some scheduling freedom by placing one initial token on both edges of the sequence constraint, $\delta_{AB_{fw}} = 1, \delta_{AB_{bw}} = 1$, the latency is $2T$ as also shown in the schedule in Figure 6.6d. Therefore, we can conclude that enforcing a *static execution order* between tasks mapped to the same processor does not always result in the minimum latency.

6.7.1 NEGATIVE TOKENS

Recently the use of negative tokens has been introduced in dataflow models [HB16, dGHKB13]. A negative number of tokens delays the enabling of an actor since the number of tokens on an edge must be positive before the consuming actor is enabled. A negative number of tokens can be used in sequence constraints to model that there are several initial executions of a task before a static execution order between two tasks starts.

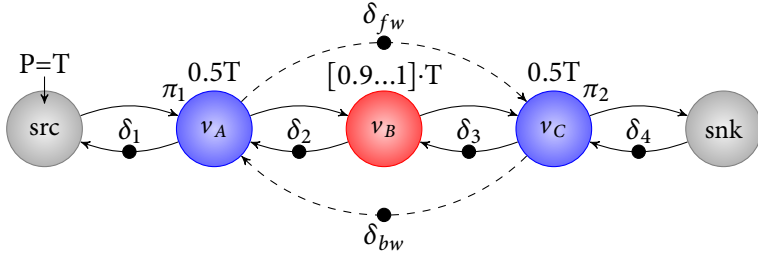
Figure 6.7a shows an example of a dataflow graph, for which enforcing a static execution order, without using negative tokens, will result in a throughput violation. A static order is enforced by setting $\delta_{fw} = 0, \delta_{bw} = 1$. In that case there are insufficient tokens on cycle $v_A \rightarrow v_B \rightarrow v_C \rightarrow v_A$ to keep up with the periodic source. When introducing a negative token, $\delta_{fw} = -1, \delta_{bw} = 2$, the throughput of the source will be achieved, while there is a static execution order after v_A is executed twice.

For this example, creating a static order with negative tokens results in a latency of $2T$, see Figure 6.7b. Whereas, without negative tokens, the latency will be $3T$. The range of the firing duration of v_B can result in v_C pre-empting v_A , which leads to this increased latency as is shown in Figure 6.7c. The approximative analysis method [KHB16c] is not able to compute a feasible result since its analysis did not converge.

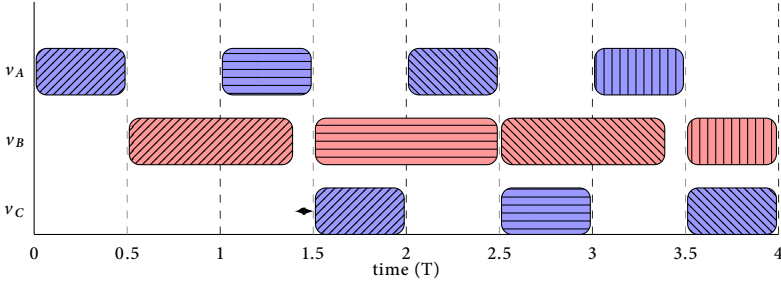
6.7.2 REDUNDANT CONSTRAINTS

Sequence constraints introduce additional constraints in an already connected dataflow graph of an application. Therefore, adding sequence constraints can lead to redundant constraints, more configurations and an increased in run-time. We now present two techniques to reduce this redundancy.

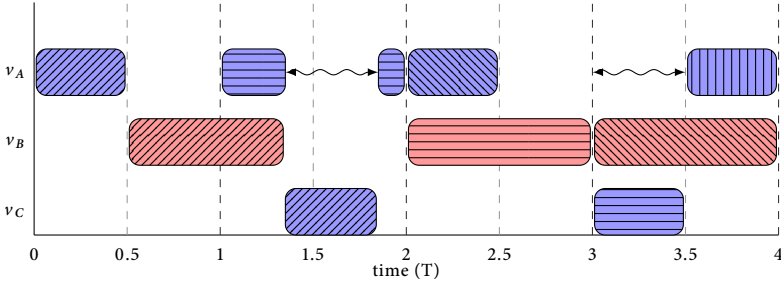
We will exploit that within one application graph, a path is required from the source to all tasks without tokens. In a dataflow model of that graph, there are tokens on the edges in the opposite direction to represent buffers with a minimum of one empty container. This connected graph allows us to create a token distance matrix,



(a) Dataflow graph where a negative number of tokens on δ_{fw} between v_A and v_C can be useful.



(b) Schedule of Figure 6.7a for which $\delta_{fw} = -1, \delta_{bw} = 2$.



(c) Schedule of Figure 6.7a for which $\delta_{fw} = 0, \delta_{bw} = 2$.

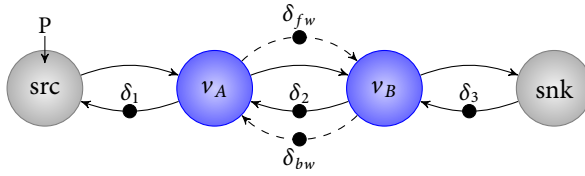


Figure 6.8: Dataflow graph where the sequence constraints between v_A and v_B is redundant to the constraints of the buffer between the actors.

containing the minimum number of tokens on the path $\mathcal{P}(v_i, v_j)$ between all actors v_i, v_j . For each configuration with buffer sizes and specific number of tokens on the forward and backward edge of the sequence constraints, this matrix can be constructed.

First of all, the sequence constraints are always redundant when these sequence constraints are inserted between two tasks, τ_A and τ_B , that are already connect by a buffer. Such sequence constraints can safely be removed to reduce the number of configurations. An example is shown in Figure 6.8, where the sequence constraints between v_A and v_B can be removed to reduce the number of configurations.

Secondly, sequence constraints impose redundant constraints if: $\delta_{fw} > \mathcal{P}(v_a, v_b)$ or $\delta_{bw} > \mathcal{P}(v_b, v_a)$. Configurations where this equation holds can be skipped without the possibly of missing a configuration, which leads to the minimal latency. In the example shown in Figure 6.7a, configurations where $\delta_{bw} > \delta_2 + \delta_3$ can be skipped.

6.8 CASE STUDY

In this section we will compare our timed automata-based analysis approach to a state-of-the-art approximative analysis approach for a WLAN 802.11p transceiver application which contains cyclic dependencies due to a feedback loop in the algorithm.

The application consists of eight tasks that are each mapped to one of four processors using an FPP scheduling policy. Figure 6.9a shows the task graph of this application, where the colors of the nodes represent a mapping to a processor. The priority of tasks is indicated next to them, as π_i , where π_1 is the lowest possible priority. The figure also shows the WCET of all tasks or the range [BCET..WCET] for tasks without constant execution times. The color of the containers in the buffers connecting the tasks indicates if they are initially empty (red) or full (green). The number of containers in each buffer in Figure 6.9a shows the size of the buffer, which does not influence the schedule of the tasks as determined by an approximate analysis approach. The sum of these buffer capacities is 21. Without buffer size optimizations, these buffer sizes will be set for the analysis of the application. In this example, the source has a period of 10 μs and a jitter of 5 μs . For this application,

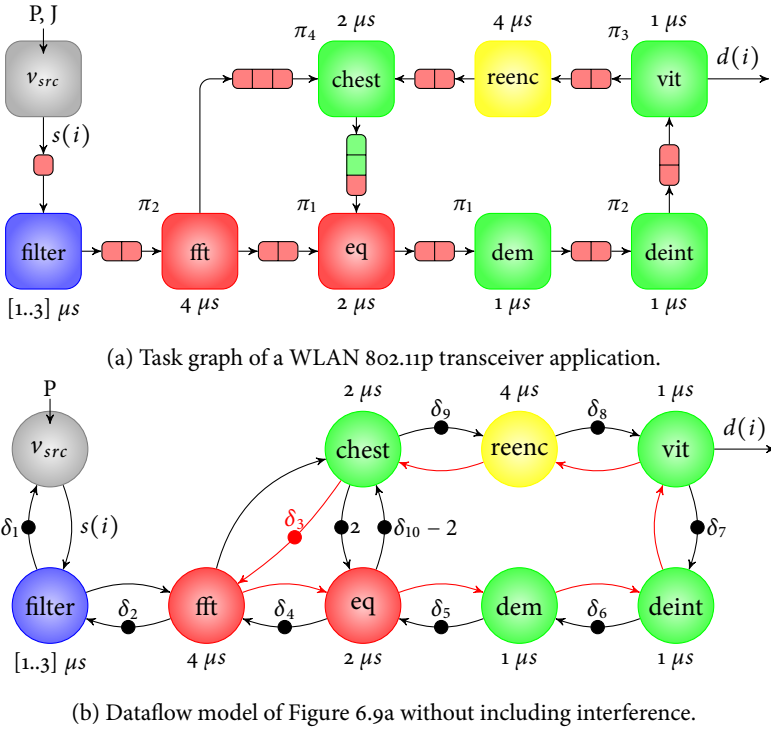


Figure 6.9: Task graph and equivalent dataflow model of a WLAN 802.11p transceiver application.

we want to minimize the maximum latency between the i 'th start of the source, v_{src} , and the corresponding i 'th production of the Viterbi task, vit .

The approximate analysis approach makes use of the fact that lower buffer capacities can result in a lower end-to-end latency [KHB16c]. Buffer sizing is therefore performed iteratively inside the analysis algorithm and sizing of buffers is not deferred until the analysis of response times of all tasks is computed. In order for the analysis flow to guarantee convergence, the iterative buffer sizing step increases the buffer sizes monotonically. Therefore, this buffer sizing technique does not guarantee that all possible buffer sizes are considered.

Our timed automata-based approach can offer more accurate latency analysis given buffer capacities. We analyze all possible buffer sizes, thereby guaranteeing the buffer size configuration that leads to the minimum latency to be present in the analyzed set of configurations. However, many of these configurations need to be considered and the evaluation of each of these configurations takes a significant run-time.

We will now compare the computed buffer sizes, end-to-end latency and run-time

Table 6.1: Settings used for the two analysis approaches and resulting configurations to consider.

#	Analysis method	Setting	Configurations or iterations
1	Approximative	Iterative buffer sizing	2
2	Approximative	Iterative sequence constraints	2
3	Approximative	Iterative combined	2
4	UPPAAL	Buffer sizing	768
5	UPPAAL	Sequence constraints	128
6	UPPAAL	Combination	8168
7	UPPAAL	Comb. no pruning	98304
8	UPPAAL	Comb. stopwatches	8168

for the two approaches using the WLANp transceiver application. These numbers will be used to draw conclusions about which analysis method to use in which case. For each approach three different settings will be used as shown in Table 6.1: buffer sizing, introducing sequence constraints, and a combination of both. We also included results of an analysis option (#7) where no pruning was performed to identify the difference in run-time. Moreover, stopwatches are used in option #8, which can result in a speedup, but potentially also less accurate analysis results.

For the timed automata-based analysis method we list the number of configurations that need to be verified in the last column of Table 6.1. In case buffer sizing and introducing sequence constraints are combined (#6) the number of configurations to consider is 12 times less than the product of both individual cases (98304, option #7). Firstly, the constraints that can be imposed by three of the seven sequence constraints becomes redundant when also performing buffer sizing and these sequence constraints are therefore removed. Secondly, after the task graph is translated into a dataflow graph, the minimum throughput can be verified. In this case, all configurations where the size of the buffer FFT to CHEST is less than two are discarded since the throughput is then limited by the sequentially executed tasks on the cycle FFT-EQ-DEMAP-DEINT-VIT-REENC-CHEST-FFT highlighted in red in Figure 6.9b, which cannot keep up with the periodic source. All remaining configurations (8168) could keep up with the source without the source blocking on full buffers. The approximative analysis approach did, for this example, always converge after two iterations as shown in Table 6.1.

The results of the analysis of the WLANp transceiver are shown in Table 6.2. Two minimization goals: minimal buffer sizes and minimal latency, are presented in separate parts of the table. The approximative timed-dataflow approach, however, does not differentiate between these goals since its goal is to minimize buffer sizes.

The first part of the table shows that when minimizing buffer sizes, both the approximative timed-dataflow approach as UPPAAL-based approach are able to reach the optimum of 12, even without making use of additional sequence constraints.

Table 6.2: Analysis results obtained using the configurations in Table 6.1 for two minimization goals: minimizing buffer sizes and latency.

#	Minimizing buffers		Minimizing latency		Total run-time (s)
	Σ buffer	Latency (μ s)	Σ buffer	Latency (μ s)	
1	12	17	12	17	< 1
2	21	17	21	17	< 1
3	12	17	12	17	< 1
4	12	16	13	15	$6.8 \cdot 10^4$
5	21	15	21	15	$1.8 \cdot 10^4$
6	12	16	13	15	$8.5 \cdot 10^5$
7	12	16	13	15	$6.3 \cdot 10^6$
8	12	16	13	15	$4.4 \cdot 10^4$

Although both approaches result in the same minimum total buffer size, the analysis using UPPAAL results in a lower latency.

The second part of the table shows that the minimum latency is only reached using UPPAAL, and is equal for all its analysis options (#4 to #8). Direct control over the interference between tasks using sequence constraints is advantageous in this case, since the least amount of configurations are considered (128 for option #5) and it results in the lowest run-time. The approximative timed-dataflow approach does result in a higher latency than our timed automata-approach, and interestingly reaches this latency of 17 μ s for a different configuration where the buffer sized are smaller than the configuration as determined by UPPAAL with a corresponding latency of 15 μ s. The difference between the best configuration for both approaches regarding latency is only in the buffer from FILTER to FFT, where the minimum latency is obtained for a buffer size of 2, where the approximative timed-dataflow approach used a buffer size of 1. We also analyzed the configuration of buffer sizes as found by the approximative timed-dataflow approach by using UPPAAL to compare the analysis results. This resulted in a latency of 16, which shows that higher buffer capacities, as derived by our approach, can lead to a lower latency. Also when using sequence constraints to optimize the latency, there is a difference between the two approaches. The approximative timed-dataflow approach sets the number of tokens on all forward edges to 0 and 1 for all backward edges. The minimum latency obtained by using UPPAAL is for a configuration where the backward edge from DEMAP to CHEST is different and contains 2 tokens instead of 1 for the approximative timed-dataflow approach. In case stopwatch automata are used (#7), the same minimal latency is found. However, latency analysis results are more pessimistic as a result of over-approximations for configurations where FFT can pre-empt EQ, i.e. the size of the buffer between FFT and EQ is larger than one.

The improvement in analysis results of our approach compared to the approximative timed-dataflow approach comes at the cost of a significant increase in run-time.

The approximative timed-dataflow approach is always finished well within 1 second as shown in the last column in Table 6.2. On the other hand, the verification time in UPPAAL of each analysis option is between $2.2 \cdot 10^4$ and $9.7 \cdot 10^5$ seconds for option #4-#6 as shown in Table 6.1. The run-time is increased by one order of magnitude when the pruning step is not used as is shown for option #7 in Table 6.2. Using stopwatches (#8), a speedup of a factor 20 is achieved. When running 16 threads in parallel on a multicore server (2x Intel Xeon CPU E5-2630 v3 @ 2.40GHz), the total run-time for the option where sequence constraints are introduced (#4) is about 23 minutes.

6.9 CONCLUSION

In this chapter we presented a hybrid analysis approach that determines the minimum latency of a cyclic task graph by adapting buffer sizes and sequence constraints using a combination of model checking and approximative analysis techniques. Each task is scheduled on one of the processors using an FPP scheduling policy.

Computationally efficient dataflow analysis techniques are used to derive lower and upper bounds on buffer sizes. In this way, the number of buffer size configurations is reduced that need to be considered by more accurate, but computationally intensive, model checking using UPPAAL. To be able to perform model checking, a network of timed automata is generated for each of the remaining configurations. The latency of each configuration is determined using UPPAAL and the best configuration is selected.

Next to the cyclic dependencies resulting from blocking buffers, additional sequence constraints are inserted. Bounds on the number of initial tokens on these sequence constraints is determined by a slightly modified version of an approximative buffer sizing algorithm. These sequence constraints can potentially reduce the latency of a task graph.

We compared the results of our hybrid analysis approach with a state-of-the-art approximative dataflow analysis approach, which uses an iterative buffer sizing technique. Using our approach, the analyzed latency decreased from 17 μ s to 15 μ s. The decrease in latency is obtained at the cost of a run-time of 23 minutes instead of a fraction of a second.

CONCLUSION

ABSTRACT – In this chapter we state the conclusions and contributions of this thesis. We also present some interesting directions for future work that are extensions of the approaches presented in this thesis.

In this thesis, we address the analysis of CPSs. In particular, we address the analysis of the embedded real-time system within a CPS implemented on a multiprocessor system. We focus on modem applications that run on such systems that are part of a vehicle-to-vehicle communication system.

The type of real-time systems we consider uses blocking buffers for the communication between tasks. These tasks are ready to execute when they can read a sufficient amount of data from their buffers. Most analysis approaches do not consider real-time systems, where buffers can introduce cyclic dependencies. To improve the analysis results, we do consider cyclic dependencies. Moreover, we consider dynamic applications, where some tasks may execute conditionally, based on the values of the incoming data. Tasks can execute in parallel on multiple processors, or execute concurrently and share a processor with other tasks. A run-time scheduler determines when a task is allowed to execute. In this thesis, we address the analysis of systems with three subclasses of run-time schedulers: budget schedulers, starvation-free schedulers, and non-starvation-free schedulers.

We analyze these real-time systems using dataflow models and model checking of timed automata. Dataflow models can be analyzed computationally efficiently, since the analysis is based on the iterative computation of fixed-points, which is relatively fast. However, schedulers cannot be modeled directly in dataflow models. The effect of scheduling is included in dataflow models based on a monotonic over-approximation for the function that calculates the interference between tasks.

This monotonic over-approximation enables the iterative fixed-point analysis. In timed automata, schedulers can be modeled directly, which can lead to more accurate analysis results. However, the run-time of the model checker for these timed automata can be very high.

7.1 SUMMARY

Real-time applications containing multiple modes and tasks that are scheduler using budget schedulers, often result in pessimistic latency analysis results. In order to improve the latency, we reduce the pessimism in the analysis results for tasks scheduled using a budget scheduler. Our dataflow analysis approach takes *mutually exclusive* execution of tasks into account to reduce this pessimism, as is presented in Chapter 3. Furthermore, we introduced a starvation-free *lock*, which allows us to enforce mutually exclusive execution of tasks. This lock allows *parallel* execution of tasks within the same group of tasks, but enforces *sequential* execution between different groups of tasks. A key difference with existing locks is that groups of tasks can only acquire the lock in a predefined order. This order is derived by a compiler from a sequential OIL-program, which describes a modal real-time stream processing application. From this sequential program, an SVPDF dataflow model is also generated that is used for checking whether the temporal constraint is satisfied after adding locks. We also show that the resulting parallel task graph generated by the compiler is always *deadlock-free*, despite that additional constraints are introduced that enforce an execution order of groups of tasks as a result of the locks. The task graph is deadlock-free because lock statements are added in such a way that no constraints are introduced that would prevent the same execution order as defined by the sequential program. As a result of introducing locks in both the application and the dataflow analysis model, we obtain more accurate analysis results.

A *compositional* temporal dataflow analysis approach is presented in Chapter 4. This approach targets applications with modes executed on multiprocessor systems that use non-starvation-free schedulers, like the FPP scheduler. The analysis approach relies on the ability to *independently characterize* the temporal behavior of modes. Different modes can be analyzed in isolation by introducing additional constraints in the application in the form of *locks* and *barriers*. Locks ensure that tasks belonging to different modes, which are also executed on the same processor, execute mutually exclusive. Barriers ensure that the interference of tasks in one mode is independent of tasks belonging to a different mode. The additional constraints introduced by the barriers and locks guarantee that the composition of modes does not change their individual temporal characterization. As a result, applications containing a hierarchy of modes can be described in an SVPDF model. The SVPDF model and the parallel implementation, including locks and barriers, are generated by a multiprocessor compiler. This model is analyzed by recursively applying existing dataflow analysis techniques. This approach determines the worst-case temporal behavior of the entire modal application.

In Chapter 5 we have presented a behavior-preserving *transformation* of strongly connected HSDF^a graphs into timed automata. These timed automata allow for the computation of *exact* end-to-end latencies, because the correlation between the firing durations of different firings is included in the analysis. The transformation of HSDF^a graphs into behaviorally-equivalent timed automata is possible because the number of tokens on edges in strongly connected HSDF^a models is *bounded*. Therefore, buffers can be modeled using the extended timed automata of UPPAAL, which by definition have a finite number of states. This also guarantees that there is a maximum number of replicas of the same actor that can fire in parallel. This limits the number of concurrent state machines – and thus states – required to model each HSDF^a actor.

A *hybrid analysis approach* is presented in Chapter 6. This approach determines the minimum latency of an application by adapting buffer sizes and sequence constraints using a *combination of model checking and approximative dataflow analysis* techniques. We extend the transformation in Chapter 5 of dataflow models to timed automata with *schedulers*. Each task is scheduled on one of the processors by using an FPP scheduling policy. Computationally efficient dataflow analysis techniques are used to derive lower and upper bounds on buffer sizes. This reduces the number of buffer size configurations that need to be considered by more accurate – but more computationally intensive – model checking of timed automata, using UPPAAL. To be able to perform model checking, a network of timed automata is generated for each of the remaining configurations. The latency of each configuration is determined using UPPAAL, and the best configuration is selected. Next to the cyclic dependencies resulting from blocking buffers, additional *sequence constraints* are inserted. These constraints can be seen as a more general applicable lock, although they do not support groups of tasks, but only introduce constraints for pairs of tasks. Bounds on the number of initial tokens on the sequence constraints is determined by a slightly modified version of an approximative buffer sizing algorithm. These constraints can potentially reduce the latency of a task graph.

7.2 CONTRIBUTIONS

In this thesis we have addressed the following *research objective*:

Define techniques that improve the accuracy of the real-time analysis results, and also increase the class of real-time multiprocessor systems and applications that can be analyzed, while minimizing the run-time of the analysis algorithms.

We have approached this objective for static applications by combining dataflow analysis with model checking of timed automata, and by introducing additional constraints to improve analysis results. Dataflow based analysis is very computationally efficient, since it uses monotonic over-approximations of the system, which allow efficient iterative fixed-point analysis. However, the dataflow model thereby abstracts from scheduling decisions, which can lead to a loss of accuracy. Timed au-

tomata often offer more accuracy, since schedulers can be modeled in more detail, however, over-approximations are still required for the analysis to remain decidable. A 100% accurate analysis is, however, probably not possible for the targeted systems, since the analysis problem is in general undecidable. Therefore, approximations have to be applied to the model checking approach, to remain decidable. Including scheduling in the timed automata increases the accuracy of the analysis results compared to dataflow analysis approaches, but leads to a non-monotonic system. This increases the run-time of the model checker for these systems.

By combining both analysis methods, more accurate analysis results are obtained. Dataflow analysis is used to limit the search space of timed automata. Furthermore, we introduce additional sequence constraints in applications. The constraints are also reflected in the analysis models. In general, the constraints limit the possible interference between tasks. This reduces scheduling freedom, which can lead to a lower run-time of the model checker. Moreover, these additional constraints enable the analysis of dynamic applications for non-starvation-free schedulers. For these dynamic application, the interference of tasks in one mode is made independently of the tasks in other modes.

Dataflow analysis is suitable for system *synthesis*, where buffer sizes that lead to the lowest latency can be determined. The approximation applied in dataflow analysis allows it to quickly analyze the system for different buffer sizes. Model checking is more suitable for *verification*, where a given system with fixed buffer sizes is accurately analyzed to verify whether it satisfies temporal constraints.

For the analysis of dynamic applications, which contain multiple modes, we propose to insert locks and barriers in these applications. The sequence constraints resulting from these locks and barriers ensure that each mode can be characterized in isolation, since interference between tasks belonging to different modes is prevented. This enables dataflow analysis for dynamic applications, which are scheduled using non-starvation-free schedulers, and reduces the latency for systems with budget schedulers.

The main *contributions* of this thesis are:

- » We have introduced a *lock* for dynamic applications that are scheduled using budget schedulers. The lock prevents interference between tasks in different modes by introducing additional sequence constraints. Thereby, the minimum throughput derived by dataflow analysis on the resulting dataflow graph is improved. (Chapter 3)
- » We have enabled the analysis of dynamic applications with non-starvation-free schedulers, by introducing *barriers*. The combination of locks and barriers allows for compositional analysis of modes, where each mode can be analyzed in isolation. This reduces the run-time of the analysis algorithm. (Chapter 4)
- » We have presented a *transformation* of dataflow graphs into temporally equivalent timed automata, to obtain exact analysis results. Model check-

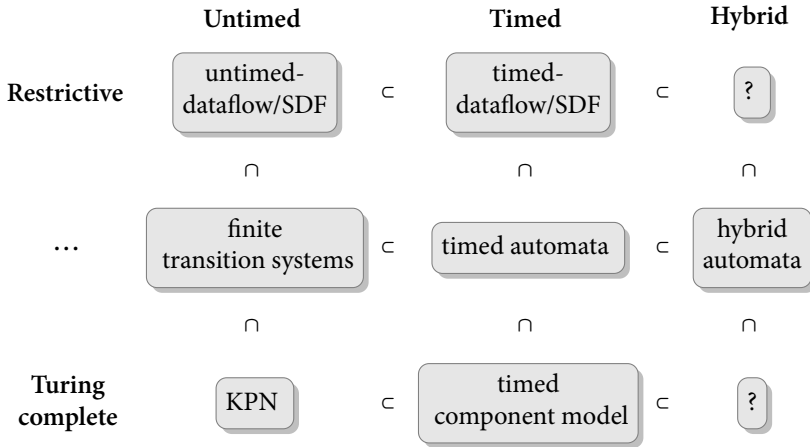
ing of the timed automata results in more accurate analysis results, and allows analysis of systems that make use of out-of-order communication. Furthermore, the correlation between the execution time of subsequent executions of the same task, and of different tasks can be taken into account. This way, analysis results are obtained that are more accurate than what is currently possible with state-of-the-art dataflow based analysis techniques. (Chapter 5)


- » We combine computationally efficient dataflow analysis techniques with model checking of timed automata, for the analysis of applications executed on multiprocessor systems with non-starvation-free schedulers. Dataflow analysis is used to derive bounds on buffer sizes at a low run-time. Accurate analysis results are obtained by model checking of timed automata models, where the buffer sizes are fixed to values within the bounds that were derived by dataflow analysis. (Chapter 6)
- » We introduce sequence constraints between pairs of tasks. This creates cyclic dependencies, which limit the interference between tasks. This can result in a reduction of the latency. (Chapter 6)

7.3 RECOMMENDATIONS FOR FUTURE WORK

Based on the research conducted in this thesis we give the following recommendations for future work:

- » We insert locks and barriers in applications based on the assignment of tasks to processors and on the modes in the application. The sequence constraints we use are more general and can be applied between all pairs of tasks to potentially improve analysis results. However, we do not present any algorithm on where to insert these constraints, but explore all possible combinations. An interesting direction for future work is to derive an efficient algorithm or heuristics for the automatic insertion of these constraints, with the goal to minimize latency or meet a throughput constraint.
- » A similar algorithm can be derived for the subproblem for identifying which tasks should execute mutually exclusive, by using locks. We enforce tasks to execute mutually exclusive when they belong to different modes in the application, and are assigned to the same processor. However, we observed that there are more cases where it is beneficial to execute tasks mutually exclusive. The case study in Section 6.8 showed that many tasks, but not all, must be enforced to execute mutually exclusive in order to minimize the latency.
- » For the analysis of dynamic applications, we only performed dataflow analysis after locks and barrier are inserted in the application. It would be interesting to compare the differences in accuracy and run-time of dataflow analysis, where locks and barriers are inserted, to results obtained with



 Figure 7.1: Figure 2.6 extended with hybrid models.

timed automata, which allow analysis without the need to insert additional constraints.

- » The analysis of run-time scheduling using timed automata relies on approximations to remain decidable, either using an over-approximation in the model, or during model checking of a timed automata model that contains stopwatches. The two different approaches are not compared thoroughly. Using stopwatches leads to a significant reduction in run-time. In the general case, timed automata extended with stopwatches are undecidable. This undecidability can in specific cases be prevented, which might be the case when taking into account that pre-emptions can only happen on clock cycle boundaries in a processor.
- » A broader class of systems that also include the continuous time part of a CPS, are hybrid systems. This class of systems can be described by hybrid automata, which are more expressive than timed automata, as also shown in Figure 7.1. The analysis of these hybrid automata is even more computationally expensive than timed automata, which justifies research into more abstract models and analysis approaches. It would be interesting to study whether the techniques presented in this thesis can be generalized such that they become applicable for hybrid systems.

ACRONYMS

A	ADC	Analog-to-Digital Converter
B	BCET	Best-Case Execution Time
	BDF	Boolean Dataflow
	BFS	Breadth-First Search
C	CPS	Cyber-Physical System
	CRC	cyclic redundancy check
	CSDF	Cyclo-Static Dataflow
	CSDF ^a	Cyclo-Static Dataflow with auto-concurrency
	CTL	Computation Tree Logic
D	DAC	Digital-to-Analog Converter
	DFS	Depth-First Search
	DSP	Digital Signal Processor
F	FIFO	First-In-First-Out
	FPP	Fixed Priority Pre-emptive
	FSM-SADF	Finite State Machine-based Scenario-Aware Data-Flow
H	HSDF	Homogeneous Synchronous Dataflow
	HSDF ^a	Homogeneous Synchronous Dataflow with auto-concurrency
K	KPN	Kahn Process Network
L	LP	Linear Program
M	MCDF	Mode-Controlled Dataflow
	MCR	Maximum Cycle Ratio
O	OIL	Omphale Input Language
R	RR	Round-Robin
S	SADF	Scenario-Aware Dataflow
	SDF	Synchronous Dataflow
	SVPDF	Structured Variable-Rate Phased Dataflow
T	TDM	Time Division Multiplex
	TDMA	Time Division Multiple Access
	TPN	Time Petri net
V	VPDF	Variable-Rate Phased Dataflow

W	WCET	Worst-Case Execution Time
	WCRT	Worst-Case Response Time
	WLANp	IEEE 802.11p

SYMBOLS

Notation	Description
\mathbb{N}_0	set of natural numbers including zero.
\mathbb{N}	set of natural numbers.
\mathbb{R}_0	set of real numbers including zero.
\mathbb{R}	set of real numbers.
v	actor in a dataflow graph.
ρ	firing duration of an actor.
$\hat{\rho}$	maximum firing duration of an actor.
$\check{\rho}$	minimum firing duration of an actor.
V	set of actors.
\hat{s}	maximum start time of an actor.
\check{s}	minimum start time of an actor.
b	buffer.
e	edge in a dataflow graph.
E	set of edges.
G	dataflow graph.
δ	initial tokens on an edge.
p	parameter of a block in an SVPDF graph.
\mathcal{P}	set of parameters.
τ	task.
S	budget of a task on a processor.
w	busy period for consecutive executions of a task.
\mathcal{I}	execution interval of a task.
J	jitter of a task.
P	period of a task.
π	priority of a task on a processor.
Q	replenishment interval of budget of a task on a processor.
\hat{R}	maximum response time of a task.
\mathcal{T}	set of tasks.
B	worst-case execution time of a task.
i	index of a token.
τ	timestamp of a token.
ϑ	value of a token.

Notation	Description
Act	finite set of actions.
C	set of clocks.
\mathcal{E}	set of edges.
\mathcal{A}	extended timed automaton.
Inv	assigns an invariant to each location.
l	location.
L	set of locations in an extended timed automaton.

BIBLIOGRAPHY

- [A⁺14] Waheed Ahmad et al. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In *ACSD*, pages 72–81, 2014. (Cited on page 87).
- [ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993. (Cited on page 24).
- [AD90] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *International Colloquium on Automata, Languages, and Programming*, pages 322–335. Springer, 1990. (Cited on page 21).
- [AD94] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994. (Cited on pages 10 and 21).
- [AFM⁺03] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *Int'l Conf. on Formal Modeling and Analysis of Timed Systems*, pages 60–72. Springer, 2003. (Cited on pages 8, 9, and 108).
- [B⁺92] François Baccelli et al. *Synchronization and linearity: an algebra for discrete event systems*. John Wiley & Sons Ltd, 1992. (Cited on page 89).
- [B⁺95] Greet Bilsen et al. Cyclo-static dataflow. In *ICASSP*, volume 5, pages 3255–3258, 1995. (Cited on page 87).
- [B⁺96] G. Bilsen et al. Cyclo-static dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996. (Cited on pages 30 and 38).
- [BB]So8] 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 *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPEs)*, pages 33–42. ACM, 2008. (Cited on pages 37 and 45).
- [BBS11] T. Bijlsma, M. J. G. Bekooij, and G. J. M. Smit. Circular buffers with multiple overlapping windows for cyclic task graphs. In *Int'l Conf. on High-Performance Embedded Architectures and Compilers (HiPEAC)*, volume 5, 2011. (Cited on page 45).
- [BHM08] Aske Brekling, Michael R Hansen, and Jan Madsen. Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(1-2):1–19, 2008. (Cited on page 108).
- [BKLo8] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008. (Cited on pages 17, 18, and 20).
- [BL93] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1993. (Cited on page 30).

- [BPvM05] M. J. G. Bekooij, S. Parmar, and J. L. van Meerbergen. Performance guarantees by simulation of process networks. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPEs)*, pages 10–19. ACM, 2005. (Cited on pages 57 and 80).
- [CE81] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981. (Cited on page 16).
- [Čer92] Kārlis Čerāns. Decidability of bisimulation equivalences for parallel timer processes. In *International Conference on Computer Aided Verification*, pages 302–315. Springer, 1992. (Cited on page 23).
- [CGMZ95] Edmund M Clarke, Orna Grumberg, Kenneth L McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 427–432. ACM, 1995. (Cited on page 16).
- [CSG99] D.E. Culler, H.P. Singh, and A. Gupta. *Parallel Computer Architecture: a hardware/software approach*. Morgan Kaufmann, 1999. (Cited on page 37).
- [CTCG⁺98] Jean Cochet-Terrasson, Guy Cohen, Stéphane Gaubert, Michael McGettrick, and Jean-Pierre Quadrat. Numerical computation of spectral elements in max-plus algebra. *IFAC Proceedings Volumes*, 31(18):667–674, 1998. (Cited on page 31).
- [Das04] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *TODAES*, 9(4):385–418, 2004. (Cited on page 8).
- [dG⁺12] Robert de Groote et al. Max-plus algebraic throughput analysis of synchronous dataflow graphs. In *SEAA*, pages 29–38. IEEE, 2012. (Cited on pages 31 and 86).
- [dGHKB13] Robert de Groote, Philip K F Hölzenspies, Jan Kuper, and Hajo Broersma. Back to basics: Homogeneous representations of multi-rate synchronous dataflow graphs. In *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, pages 35–46. IEEE, 2013. (Cited on page 119).
- [DIG99] Ali Dasdan, Sandy S Irani, and Rajesh K Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 37–42. ACM, 1999. (Cited on page 31).
- [Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965. (Cited on page 15).
- [Dij72] Edsger W. Dijkstra. Information streams sharing a finite buffer. *Information Processing Letters*, 1(5):179 – 180, 1972. (Cited on page 15).
- [Dil89] David L Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Conference on Computer Aided Verification*, pages 197–212. Springer, 1989. (Cited on pages 21 and 24).
- [DILS09] Alexandre David, Jacob Illum, Kim G Larsen, and Arne Skou. Model-based framework for schedulability analysis using UPPAAL 4.1. *Model-based design for embedded systems*, 1(1):93–119, 2009. (Cited on pages 16, 86, and 108).

- [DSB⁺13] Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. Schedule-extended synchronous dataflow graphs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (CADICS)*, 32(10):1495–1508, 2013. (Cited on pages 8 and 38).
- [FKH⁺08] Joachim Falk, Joachim Keinert, Christian Haubelt, Jürgen Teich, and S.S. Bhat-tacharyya. A generalized static data flow clustering algorithm for mp soc scheduling of multimedia applications. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*. ACM, 2008. (Cited on page 105).
- [FMPY06] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354(2):301–317, 2006. (Cited on page 108).
- [G⁺07] Amir Hossein Ghamarian et al. Latency minimization for synchronous dataflow graphs. In *DSD*, pages 189–196, 2007. (Cited on page 87).
- [GGS⁺06] Amir Hossein Ghamarian, M. C. W. Geilen, Sander Stuijk, Twan Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and MohammadReza Mousavi. Throughput analysis of synchronous data flow graphs. In *Int'l Conf. on Application of Concurrency to System Design (ACSD)*, pages 25–36. IEEE, 2006. (Cited on pages 26 and 31).
- [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 *Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2013. (Cited on pages 30, 35, 36, 38, 53, 63, 70, and 78).
- [GHB14] 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 *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2014. (Cited on pages 30, 52, and 63).
- [GS02] Zonghua Gu and Kang G Shin. Analysis of event-driven real-time systems with Time Petri Nets. In *Design and Analysis of Distributed Embedded Systems*, pages 31–40. Springer, 2002. (Cited on page 87).
- [GS10] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *IEEE/ACM/IFIP Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2010. (Cited on pages 35, 62, and 63).
- [Gua09a] Qian Guangming. An earlier time for inserting and/or accelerating tasks. *Real-Time Systems*, 41(3):181–194, 2009. (Cited on pages 35 and 37).
- [Gua09b] Qian Guangming. An earlier time for inserting and/or accelerating tasks. *Real-Time Systems*, 41(3):181–194, 2009. (Cited on page 63).
- [H⁺13a] Joost P H M Hausmans et al. Two parameter workload characterization for improved dataflow analysis accuracy. In *RTAS*, pages 117–126, 2013. (Cited on pages 98 and 100).
- [H⁺13b] J.P.H.M. Hausmans et al. Two parameter workload characterization for improved dataflow analysis accuracy. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2013. (Cited on pages 37 and 39).

- [H⁺14] Joost P H M Hausmans et al. Unified dataflow model for the analysis of data and pipeline parallelism, and buffer sizing. In *MEMOCODE*, pages 12–21. IEEE, 2014. (Cited on page 86).
- [H⁺16] Joost P H M Hausmans et al. A refinement theory for timed-dataflow analysis with support for reordering. In *EMSOFT*, page 20. ACM, 2016. (Cited on pages 32 and 86).
- [Hau15] J.P.H.M. Hausmans. *Abstractions for aperiodic multiprocessor scheduling of real-time stream processing applications*. PhD thesis, Centre for Telematics and Information Technology, University of Twente, 2015. (Cited on pages 5 and 28).
- [HB16] Joost P H M Hausmans and Marco J G Bekooij. A refinement theory for timed-dataflow analysis with support for reordering. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, page 20. ACM, 2016. (Cited on page 119).
- [Her88] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. ACM Symp. on Principles of Distributed Computing (PODC)*, 1988. (Cited on page 37).
- [HGGM01] M González Harbour, JJ Gutiérrez García, JC Palencia Gutiérrez, and JM Drake Moyano. MAST: Modeling and analysis suite for real time applications. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 125–134. IEEE, 2001. (Cited on pages 8, 106, and 107).
- [HGWB14] 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 *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 108–117. ACM, 2014. (Cited on pages 9, 62, 63, and 66).
- [HHJ⁺05] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis—the SymTA/S approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005. (Cited on pages 8, 106, and 107).
- [HHK01] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Embedded Software*, pages 166–184. Springer, 2001. (Cited on page 37).
- [HV06] Martijn Hendriks and Marcel Verhoef. Timed automata based analysis of embedded system architectures. In *Proc. Parallel & Distributed Processing Symp.*, pages 8–pp. IEEE, 2006. (Cited on pages 8, 9, 97, and 107).
- [HWGB13] J. P. H. M. Hausmans, M. H. Wiggers, S. J. Geuns, and M. J. G. Bekooij. Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2013. (Cited on pages 9, 63, and 66).
- [JPTY08] Bengt Jonsson, Simon Perathoner, Lothar Thiele, and Wang Yi. Cyclic dependencies in modular performance analysis. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 179–188. ACM, 2008. (Cited on pages 8 and 107).

- [K⁺16] Peter Koek et al. CSDF^a: A model for exploiting the trade-off between data and pipeline parallelism. In *SCOPES*, pages 30–39. ACM, 2016. (Cited on pages 28, 30, and 87).
- [Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings IFIP Congress*, pages 471–475, 1974. (Cited on page 32).
- [KB17] P Kurtin and M Bekooij. An abstraction-refinement theory for the analysis and design of real-time systems (extended version). *CTIT Technical Report Series*, (TR-CTIT-17-06), 2017. (Cited on pages 9 and 10).
- [KHB16a] P. S. Kurtin, J. P. H. M. Hausmans, and M. J. G. Bekooij. Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*. IEEE, 2016. To appear. (Cited on pages 8 and 72).
- [KHB16b] P. S. Kurtin, J. P. H. M. Hausmans, and M. J. G. Bekooij. HAPI: An event-driven simulator for real-time multiprocessor systems. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2016. (Cited on pages 57 and 80).
- [KHB16c] Philip S Kurtin, Joost P H M Hausmans, and Marco J G Bekooij. Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 1–12. IEEE, 2016. (Cited on pages 99, 101, 106, 108, 114, 117, 119, and 122).
- [KY04] Pavel Krčál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 236–250. Springer, 2004. (Cited on page 108).
- [Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8), 1974. (Cited on page 37).
- [Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), 1979. (Cited on page 37).
- [Lew90] Harry R Lewis. A logic of concrete time intervals. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on*, pages 380–389. IEEE, 1990. (Cited on page 21).
- [LM⁺87] Edward Lee, D. G. Messerschmitt, et al. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. (Cited on pages 29 and 31).
- [LMB⁺14] Alok Lele, Orlando Moreira, Joao Bastos, Ricardo Almeida, Paulo Pedreiras, and Kees van Berkel. Analyzing preemptive fixed priority scheduling of data flow graphs. In *IEEE Symp. on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 50–59. IEEE, 2014. (Cited on pages 9 and 106).
- [LMC12] Alok Lele, Orlando Moreira, and Pieter JL Cuijpers. A new data flow analysis model for tdm. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 237–246. ACM, 2012. (Cited on page 9).

- [LMvB15] Alok Lele, Orlando Moreira, and Kees van Berkel. FP-scheduling for mode-controlled dataflow: a case study. In *Design, Automation and Test in Europe (DATE)*, pages 1257–1260. EDA Consortium, 2015. (Cited on page 63).
- [LP95] E.A. Lee and T.M. Parks. Dataflow process networks. In *Proc. of the IEEE*, May 1995. (Cited on pages 27, 32, and 38).
- [MB07] Orlando M. Moreira and Marco J. G. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007(1):1–14, 2007. (Cited on pages 85 and 87).
- [MDA09] Gabor Madl, Nikil Dutt, and Sherif Abdelwahed. A conservative approximation method for the verification of preemptive scheduling using timed automata. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 255–264. IEEE, 2009. (Cited on page 108).
- [Mil80] Robin Milner. A calculus of communicating systems. 1980. (Cited on page 19).
- [MLR⁺10] Marius Mikučionis, Kim Guldstrand Larsen, Jacob Illum Rasmussen, Brian Nielsen, Arne Skou, Steen Ulrik Palm, Jan Storbak Pedersen, and Poul Houggaard. Schedulability analysis using uppaal: Herschel-planck case study. In *Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation*, pages 175–190. Springer, 2010. (Cited on page 107).
- [N⁺02] A. Nieuwland et al. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems (DAES)*, 7(3), 2002. (Cited on page 37).
- [Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical computer science*, pages 167–183. Springer, 1981. (Cited on page 19).
- [PBL95] Jose Luis Pino, Shuvra S Bhattacharyya, and Edward A Lee. A hierarchical multiprocessor scheduling system for dsp applications. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 122–126. IEEE, 1995. (Cited on page 30).
- [PWT⁺07] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 193–202. ACM, 2007. (Cited on pages 8 and 107).
- [RC04a] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-time systems*, 2004. (Cited on page 63).
- [RC04b] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, 2004. (Cited on pages 35 and 37).
- [Rei68] Raymond Reiter. Scheduling parallel computations. *Journal of the ACM (JACM)*, 15(4):590–599, 1968. (Cited on pages 31 and 114).

- [S⁺89] Lui Sha et al. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989. (Cited on pages 35, 37, and 63).
- [S⁺10] N. Stoimenov et al. Resource adaptations with servers for hard real-time systems. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*. ACM, 2010. (Cited on pages 35, 37, and 38).
- [S⁺13] F. Siyoum et al. Automated extraction of scenario sequences from disciplined data-flow networks. In *Int'l Conf. on Formal Methods and Models for Codesign (MEM-OCODE)*, 2013. (Cited on pages 35 and 38).
- [SB09] Sundararajan Sriram and S. S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2009. (Cited on pages 8, 25, 29, and 31).
- [SBW09] M. Steine, M.J.G. Bekooij, and M.H. Wiggers. A priority-based budget scheduler with conservative dataflow model. In *Euromicro Conf. on Digital System Design Architectures, Methods and Tools (DSD)*. IEEE, 2009. (Cited on pages 36 and 38).
- [Szy88] B.K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. Int'l Conference on Supercomputing*, 1988. (Cited on page 37).
- [T⁺06] B.D. Theelen et al. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2006. (Cited on page 38).
- [TBW92a] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority preemptively scheduled systems. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 100–109. IEEE, 1992. (Cited on page 63).
- [TBW92b] K.W. Tindell, A. Burns, and A.J. Wellings. Mode changes in priority preemptively scheduled systems. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 100–109, Dec 1992. (Cited on pages 35 and 37).
- [TBW94] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994. (Cited on pages 9 and 66).
- [TCG⁺01] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, Alexander Maxiaguine, and Jonas Greutert. Embedded software in network processors—models and algorithms. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 416–434. Springer, 2001. (Cited on page 107).
- [TS09] Lothar Thiele and Nikolay Stoimenov. Modular performance analysis of cyclic data-flow graphs. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 127–136. ACM, 2009. (Cited on pages 8 and 107).
- [vdBB07] J.-W. van den Brand and M.J.G. Bekooij. Streaming consistency: a model for efficient MPSoC design. In *Euromicro Conf. on Digital System Design Architectures, Methods and Tools (DSD)*, 2007. (Cited on page 37).
- [vKSG17] Reinier van Kampenhout, Sander Stuijk, and Kees Goossens. Programming and analysing scenario-aware dataflow on a multi-processor platform. In *Design, Automation and Test in Europe (DATE)*, pages 876–881. European Design and Automation Association, 2017. (Cited on page 63).

- [WBS07a] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 11–22. ACM, 2007. (Cited on page 62).
- [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 *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2007. (Cited on page 40).
- [WBS09] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Monotonicity and run-time scheduling. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, 2009. (Cited on pages 6, 9, 28, 36, 38, and 62).
- [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. (Cited on pages 35, 38, and 62).
- [WGHB15] Philip S Wilmanns, Stefan J Geuns, Joost P H M Hausmans, and Marco J G Bekooij. Buffer sizing to reduce interference and increase throughput of real-time stream processing applications. In *IEEE Symp. on Real-Time Computing (ISORC)*, pages 9–18. IEEE, 2015. (Cited on pages 8, 105, 106, and 109).
- [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 *Euromicro Conf. on Digital System Design Architectures, Methods and Tools (DSD)*. IEEE, 2014. (Cited on pages 63, 67, 70, and 74).

LIST OF PUBLICATIONS

- [GK:1] Guus Kuiper, Berend H J Dekens, Stefan J Geuns, Philip S Wilmanns, Joost P H M Hausmans, and Marco J G Bekooij. Compiler for real-time multiprocessor systems with shared accelerators. In *Design, Automation and Test in Europe (DATE)*, 2015.
- [GK:2] Guus Kuiper, Stefan J Geuns, and Marco J G Bekooij. Utilization improvement by enforcing mutual exclusive task execution in modal stream processing applications. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2015.
- [GK:3] Guus Kuiper, Stefan J Geuns, Joost P H M Hausmans, and Marco J G Bekooij. Compositional temporal analysis method for fixed priority pre-emptive scheduled modal stream processing applications. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 98–107. ACM, 2016.
- [GK:4] Guus Kuiper and Marco J G Bekooij. Latency analysis of homogeneous synchronous dataflow graphs using timed automata. In *Design, Automation and Test in Europe (DATE)*. IEEE, 2017.
- [GK:5] Guus Kuiper, Philip S Kurtin, and Marco J G Bekooij. Hybrid latency minimization approach using model checking and dataflow analysis. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2017.

THIS THESIS

```
@phdthesis{kuiper2019:thesis,  
  author={Kuiper, Guus},  
  title={Accurate analysis of real-time stream processing applications --  
        using dataflow models and timed automata},  
  school={University of Twente},  
  address={PO Box 217, 7500 AE Enschede, The Netherlands},  
  year={2019},  
  month=mar,  
  day={8},  
  number={IDS Ph.D. Thesis Series No. 18-463},  
  issn={2589-4730},  
  isbn={978-90-365-4541-9},  
  doi={10.3990/1.9789036545419}  
}
```

BIBTEX of this thesis

INDEX

Symbols

Cyber-Physical System, 3

A

abstraction
 bounding, *see also*
 bounding abstraction
 inclusion, *see also*
 inclusion abstraction
actor, 25
actuator, 3
admissible schedule, 38
auto-concurrency, 28, 30

B

BIBTEX of this thesis, 147
bisimulation, 19
bounding abstraction, 9
buffer, 4

C

clock region, 23
complexity class, 24
concurrency, 5
conditional execution, 4
configuration, 109
consistency, 28
continuous-time, 3
contributions, 130
CSDF, 30
CSDF^a, 30
cyclic dependencies, 5
cyclic dependency, 105

D

data-driven, 25
Dataflow models, 25
dataflow models
 BDF, *see also* BDF
 CSDF, *see also* CSDF

CSDF^a, *see also* CSDF^a
HSDF, *see also* HSDF
SDF, *see also* SDF
SVPDF, *see also* SVPDF

deadlock, 27
direct predecessors, 17
direct successors, 17
discrete-time, 3
dynamic application, 4

E

edge, 25
embedded system, 2
enabled, 25
end-to-end latency, 89
execution time, 5
external enabling time, 66

F

firing duration, 25
firing rule, 25
fixed-point, 9
functional firing, 27
functionally deterministic, 27

H

handshaking actions, 24
HSDF, 25

I

interference, 6

L

latency, 5
lock, 35
lock implementation, 46

M

modal application, 4
mode, 4

mode changes, 35
 model checking, 16
 monotonic, 9
 monotonicity, 28
 mutually exclusion
 inter-iteration, 43
 intra-iteration, 43

P

parallelism, 5
 periodic, 3

R

rate, 29
 reachability analysis, 18
 backward reachability, 18
 forward reachability, 18
 real-time system, 4
 region graph, 23
 repetition vector, 28, 29
 replenishment interval, 7, 40
 research objective, 129
 response time, 6

S

scheduler, 5, 6
 scheduling
 FPP, *see also* FPP
 RR, *see also* RR
 TMDA, *see also* TDMA
 SDF, 29
 self-edge, 28
 self-timed execution, 25
 sensor, 3
 sequence constraints, 116
 sequential firing rule, 27
 shared resource, 6
 static application, 4
 static schedule, 29
 static-order schedule, 110, 119
 stream processing applications, 3
 SVPDF, 30
 synthesis, 130
 system
 CPS, *see also* Cyber-Physical System
 embedded, *see also* embedded system
 real-time, *see also* real-time system

T

task, 3
 terminal state, 17
 throughput, 5
 timed automata, *see also* timed automaton
 token
 timed automaton, 22
 token, 25
 topology matrix, 28
 transition
 delay transition, 23
 discrete transition, 22
 transition system, 17
 quotient transition system, 20
 timed transition system, 22

U

uppaal
 guard, 91
 select, 91
 synchronization, 91
 update, 91

V

verification, 130



ISBN 978-90-365-4541-9



9 789036 545419