Philip-Sebastian Kurtin

An Abstraction-Refinement Theory for the Analysis and Design of Concurrent Real-Time Systems

Members of the graduation committee:

Prof. dr.	J. N. Kok	University of Twente (chairman and secretary)
Prof. dr. ir.	M. J. G. Bekooij	University of Twente (promotor)
Prof. dr. ir.	G. J. M. Smit	University of Twente (member)
Assoc. prof.	R. Langerak	University of Twente (member)
Prof. dr. ir.	R. Ernst	Technische Universität Braunschweig (member)
Prof. dr.	S. Chakraborty	Technische Universität München (member)
Dr. ir.	M. C. W. Geilen	Eindhoven University of Technology (special expert)

UNIVERSITY OF TWENTE.

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

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

DSI Ph.D. Thesis Series No. 18-017
Digital Society Institute
P.O. Box 217, 7500 AE Enschede, The Netherlands

This research has been conducted within the Integrated Design Approach for Safety-Critical Real-Time Automotive Systems project (project number 12698). This research is supported by the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs.

Copyright © 2018 Philip-Sebastian Kurtin, Eindhoven, The Netherlands. This work is licensed under the Creative Commons Attribution 4.0 International License.
<http://creativecommons.org/licenses/by/4.0/>.

Cover design by Sabrina Hörberg and Philip-Sebastian Kurtin.

This thesis was typeset using $\LaTeX$, Ipe, TikZ and Kile.

This thesis was printed by ProefschriftMaken.
<http://www.proefschriftmaken.nl>.

ISBN 978-90-365-4646-1
ISSN 2589-7721 (DSI Ph.D. Thesis Series No. 18-017)
DOI 10.3990/1.9789036546461

AN ABSTRACTION-REFINEMENT THEORY

FOR THE

ANALYSIS AND DESIGN OF CONCURRENT REAL-TIME
SYSTEMS

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 9 november 2018 om 12.45 uur

door

Philip-Sebastian Kurtin

geboren op 21 juni 1986
te Paderborn, Duitsland

Dit proefschrift is goedgekeurd door:

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

Abstract

Concurrent real-time systems with shared resources belong to the class of safety-critical systems. For such systems it is required to determine both temporally and functionally conservative guarantees, whereas estimates and approximations are usually not sufficient.

However, the growing complexity of real-time systems makes it more and more challenging to apply standard techniques for their analysis. Especially the presence of both cyclic data dependencies (which can occur due to feedback loops, but also due to the usage of FIFO buffers) and cyclic resource dependencies (i.e. resource dependencies that are opposed to the flow of data) makes many related analysis approaches inapplicable. Finally, also the usage of Static Priority Pre-emptive (SPP) scheduling and its accompanying scheduling anomalies (temporal non-monotonicities) further impede the employment of many “classical” analysis techniques.

To address this growing complexity and to be able to give guarantees nevertheless we consequently need to reduce complexity in a both temporally and functionally conservative manner. We have to be careful, however, because every complexity reduction entails a reduction of accuracy, they are two sides of the same medal.

We identify two different methods to reduce complexity in a conservative way. First, we can reduce complexity by reducing the “*entropy*” of analysis models by means of abstraction (as opposed to refinement), such that the amount of information contained within the models is lowered. And second, we can also reduce complexity by a reduction of analysis technique “*effectiveness*”, i.e. we apply simpler, but coarser techniques on the same models. Given these two levers, we aim to seamlessly trade off between complexity and accuracy.

However, there are two significant restrictions that prevent us from making such trade-offs. For one, there is no abstraction-refinement theory that supports the abstraction of non-deterministic, non-monotone, cyclic real-time systems to deterministic, monotone and (if required) acyclic analysis models. And given

suitable analysis models, there is only a very coarse analysis technique for the analysis of such models, which prevents trading off accuracy and complexity on the effectiveness scale.

For that matter we present an abstraction-refinement theory for real-time systems in the first part of this thesis. We introduce a timed component model that is defined in such a generic way that both real-time system implementations and any kinds of analysis models for such applications can be expressed therein. The two main constructs of the component model are streams, which are finite or infinite sequences of events (i.e. pairs of timestamps and values), as well as the components themselves that transform input streams to output streams via mathematical relations. We prove various properties for this component model, such as the automatic lifting of input acceptance (i.e. the acceptance of input streams) from component to graph level.

Thereafter, we devise three different abstraction-refinement theories for the timed component model, exclusion, inclusion and bounding. Exclusion can be used to remove unconsidered corner cases such as potential malfunctions that are outside the scope of analysis, resulting in a “*model of reality*”. Inclusion abstraction allows for the substitution of uncertainty in the model of reality with non-determinism and inclusion refinement enables to derive implementations for which no specific events need to occur, as long as the events take place within certain ranges. In contrast, bounding abstraction permits to replace non-determinism with determinism, enabling the creation of efficiently analyzable models that can be used to give temporal or functional guarantees on non-deterministic and non-monotone implementations, whereas bounding refinement allows to create implementations that adhere to temporal or functional properties of deterministic specification models. We further differ between best-case and worst-case bounding, which is required for applications in which jitter plays a major role.

In the second part of the thesis we use exclusion, inclusion and bounding abstractions to construct several analysis models from concurrent real-time systems with shared resources and SPP scheduling. For SPP scheduling it is required to determine so-called enabling rate characterizations of tasks, as otherwise the interference of higher priority tasks on lower priority ones cannot be bounded. Moreover, the presence of cyclic data and resource dependencies creates a mutual dependency between the schedules of tasks (their enabling and finish times) and the interference due to resource dependencies, which is the reason why all our presented analysis approaches are iterative.

For the determination of schedules we make use of two dataflow models that we define as best-case and worst-case bounding abstractions of the analyzed real-time systems. With the best-case models we compute periodic lower bounds on the schedules of tasks, while upper bounds are determined with the worst-case models. These lower and upper bounds can be used in so-called response time analysis models, which are inclusion abstractions of the underlying real-time systems, to calculate upper bounds on the jitters of higher priority tasks that

are suitable to determine so-called period-and-jitter interference characterizations. With these characterizations we can compute maximum response times of tasks, i.e. differences between maximum enabling and finish times considering interference, which results in the aforementioned coarse analysis approach.

To improve effectiveness and therewith accuracy of the period-and-jitter analysis we first propose to combine the aforementioned interference characterization with an explicit consideration of cyclic data dependencies that are the result of both feedback loops and FIFO buffers. We show that the consideration of such dependencies to limit interference results in a significantly higher analysis accuracy. Based on this observation, we attempt to exploit this interference-limiting effect even more by introducing an iterative buffer sizing which does not only analyze, but actually optimizes the underlying real-time systems. On top of that, we discuss the introduction of so-called synchronization edges that can be seen as non-functional FIFO buffers and are placed directly between higher and lower priority tasks, enabling a further exploitation of cyclic dependencies to restrict interference. Lastly, we replace the coarse period-and-jitter interference characterization with execution intervals, resulting in an even higher analysis accuracy.

In our last analysis approach we do not only aim at increasing analysis accuracy, but also at increasing analysis applicability by enabling the support of real-time systems with tasks consisting of multiple phases and operating at different rates. We show that this approach is not only suitable for inherently multi-rate, multi-phase applications, but also for single-rate, single-phase ones because the modeling with phases can be also used to relax the rather strict requirements on the underlying hardware platforms imposed by the aforementioned approaches. With a modification of this approach we further enable the analysis of applications with multiple shared resources.

Finally, we also discuss simulation in the context of our modeling framework. We present the so-called HAPI simulator, which is capable of simulating any kinds of concurrent real-time systems with shared resources. Among other use-cases, we also show that HAPI can be used to falsify erroneously determined analysis results. The latter is enabled by the fact that we can apply HAPI on the same models that we also use for analysis, which is a significant distinction to related approaches.

In the first part of this thesis we show with various case studies the applicability of our abstraction-refinement theory and the underlying component model for many kinds of discrete-event systems and analysis models. In the second part we use a WLAN transceiver application as an ongoing case study to demonstrate that all our improvements, i.e. consideration of cyclic data dependencies, iterative buffer sizing, synchronization edges, execution intervals and multi-phase analysis, result in a significant improvement of analysis accuracy.

Samenvatting

Parallele real-time systemen waarin meerdere software taken van dezelfde hardware componenten gebruik maken, worden steeds vaker ingezet voor applicaties die niet mogen falen omdat anders de veiligheid van gebruikers in het geding komt. Voor dergelijke systemen is het vereist om te garanderen dat de resultaten correct zijn en op tijd worden geleverd. Het is dan ook voor dit soort systemen onvoldoende om een schatting of benaderingen te geven waarvan niet gegarandeerd is dat ze pessimistisch zijn.

De groeiende complexiteit van parallele real-time systemen maakt het echter steeds problematischer om standaard analyse technieken toe te passen. Deze standaard analyse technieken zijn vaak niet toepasbaar indien er cyclische afhankelijkheden zijn. Deze cyclische afhankelijkheden ontstaan ten gevolge van regellussen in de applicaties waarbij de uitgangresultaten opnieuw geconsumeerd worden. Ze kunnen ook ontstaan wanneer software taken, die data met elkaar communiceren, uitgevoerd worden op dezelfde processoren. Dit gebeurt ook als ze uitgevoerd worden op processoren die hardware componenten zoals geheugens en communicatiebussen delen. Ten slotte, zijn standaard analyse technieken vaak ook niet geschikt voor systemen die gebruik maken van de Static Priority Preemptive (SPP)-scheduling techniek omdat dit gedrag veroorzaakt dat niet monotoon is, wat ook wel bekend staat als scheduling anomalieën.

Om de groeiende complexiteit aan te pakken en om toch garanties te kunnen bieden, moeten we daarom de complexiteit verminderen door gebruik te maken van gegarandeerd pessimistische overapproximatie technieken voor zowel het temporele als het functionele gedrag. Hierbij moeten we echter voorzichtig te werk gaan omdat elke complexiteitsverlaging onvermijdelijk een vermindering van de nauwkeurigheid met zich meebrengt, het zijn namelijk twee kanten van dezelfde medaille.

We identificeren twee verschillende methoden om de complexiteit van de analyse te verminderen waarbij de resultaten gegarandeerd pessimistisch blijven.

Ten eerste kunnen we de complexiteit verminderen door de “*entropie*” van analysemodellen te verminderen door middel van abstractie (het tegenovergestelde van verfijning), zodat de hoeveelheid informatie die de modellen bevatten wordt verlaagd. En ten tweede kunnen we ook de complexiteit verminderen door een reductie van de “*effectiviteit*” van een analysetechniek, d.w.z. we passen eenvoudigere, maar grofstoffelijkere technieken toe op dezelfde modellen. Met deze twee methodes streven we ernaar om een traploze afweging te kunnen maken tussen complexiteit en nauwkeurigheid.

Er waren echter twee belangrijke belemmeringen die hebben belet om dergelijke compromissen te maken. Ten eerste was er geen abstractie-verfijningstheorie die de abstractie van onzekere, niet-monotone, cyclische real-time systemen mogelijk maakt waarbij deterministische, monotone en (indien nodig) acyclische modellen gecreëerd worden. En voor de beschikbare analysemodellen was er slechts één heel grofstoffelijke analysetechniek beschikbaar, waardoor er geen traploze afweging gemaakt kon worden tussen nauwkeurigheid en complexiteit.

Daarom presenteren we in het eerste deel van dit proefschrift een abstractie-verfijningstheorie voor real-time systemen. We introduceren een model dat zodanig generiek gedefinieerd is dat zowel real-time systeem implementaties er in uitgedrukt kunnen worden als allerlei soorten analysemodellen. De twee hoofdconstructies van het component-model zijn stromen, die zowel eindige als oneindige reeksen van gebeurtenissen zijn (dat wil zeggen paren van tijdstippen en waarden), en de componenten die via mathematische relaties ingangsstromen transformeren in uitgangsstromen. We bewijzen verschillende eigenschappen van dit component-model, zoals het automatisch optillen van acceptatie van de ingangsstromen door de componenten naar acceptatie van de ingangsstromen door een graaf van deze componenten.

Daarna presenteren we drie verschillende abstractie-verfijningstheorieën voor het component model: uitsluiting, insluiting en beperking. Uitsluiting kan worden gebruikt om grensgevallen te verwijderen, zoals potentiële verstoringen die we niet willen beschouwen tijdens de analyse, resulterend in “*een model van de realiteit*”. Insluitings-abstractie maakt de vervanging van onzekerheid in het model van realiteit door zogeheten “onbepaald” gedrag mogelijk, terwijl het insluitings-verfijning mogelijk maakt om implementaties af te leiden waarvoor gebeurtenissen alleen maar binnen bepaalde grenzen hoeven plaats te vinden. In tegenstelling hiermee kan beperkings-abstractie onbepaaldheid vervangen door deterministisch/uniek gedrag, waardoor efficiënte analysemodellen kunnen worden gemaakt die kunnen worden gebruikt om temporele of functionele garanties te geven voor implementaties met een onbepaald / of niet-monotone gedrag. Daarentegen maakt beperkings-verfijning het mogelijk om implementaties te maken die zich houden aan temporele en/of functionele eigenschappen van deterministische specificatiemodellen. Ten slotte, maken we ook onderscheid tussen het slechts mogelijke en het best mogelijke temporele gedrag zodat we de zogeheten jitter kunnen bepalen voor toepassingen waarbij dit een belangrijke metriek is.

In het tweede deel van het proefschrift gebruiken we uitsluiting-, insluitings-, en beperkings-abstracties om verschillende analysemodellen te construeren voor real-time systemen met gedeelde hulpbronnen en SPP-scheduling. Voor de SPP-scheduling is het nodig om een activeringskarakterisatie van taken te bepalen, omdat anders de invloed op het temporele gedrag van andere taken niet begrensd kan worden. Bovendien zorgt de aanwezigheid van cyclische data-, en hulpbronafhankelijkheden voor een wederzijdse afhankelijkheid tussen de scheduling van taken (hun activering- en eindtijden) en de momenten waarop andere taken kunnen worden uitgevoerd en de noodzakelijk input data kunnen produceren. Deze cyclische afhankelijkheid heeft geresulteerd in het iteratief zijn van onze analysetechnieken.

Voor de bepaling van schedules maken we gebruik van twee dataflow modellen die we definiëren als best-case en worst-case beperkings-abstracties van de geanalyseerde real-time systemen. Bij de best-case modellen berekenen we periodieke ondergrenzen op de uitvoeringsmomenten van taken, terwijl bovengrenzen worden bepaald met de worst-case modellen. Deze onder-, en bovengrenzen kunnen worden gebruikt in zogenaamde reactietijd analysemodellen, die insluitings-abstracties zijn van de onderliggende real-time systemen, om de bovengrenzen van de jitters van taken met een hogere prioriteit te berekenen, die geschikt zijn om de zogenaamde periode-en-jitter interferentie-karakterisering te bepalen. Met deze karakterisering kunnen we maximale reactietijden van taken berekenen, d.w.z. verschillen tussen maximale activerings-, en eindtijden met inachtneming van interferentie, wat resulteert in de hiervoor genoemde grofstoffelijke analysebenadering.

Om de effectiviteit en daarmee de nauwkeurigheid van de periode-en-jitter analyse te verbeteren, stellen we voor om de bovengenoemde interferentie-karakterisering te combineren met een expliciete overweging van cyclische data-afhankelijkheden die het resultaat zijn van zowel terugkoppellussen als FIFO-buffers. We laten zien dat de overweging van dergelijke afhankelijkheden om interferentie te beperken resulteert in een significant hogere nauwkeurigheid van de analyse. Op basis van deze waarneming proberen we dit interferentiebeperkende effect nog meer te benutten door het introduceren van een iteratieve buffergrootteberekening die niet alleen de onderliggende real-time systemen analyseert, maar eigenlijk optimaliseert. Tevens bespreken we de introductie van zogenaamde synchronisatie-signalen die kunnen worden gezien als niet-functionele FIFO-buffers welke direct worden geplaatst tussen taken met een hogere en een lagere prioriteit, waardoor een verdere exploitatie van cyclische afhankelijkheden mogelijk wordt om interferentie te beperken. Ten slotte vervangen we de grofstoffelijke periode-en-jitter interferentie-karakterisering door zogeheten uitvoeringsintervallen, wat resulteert in een nog hogere nauwkeurigheid van de analyse.

In onze laatste analyse methode willen we niet alleen de nauwkeurigheid van de analyse vergroten, maar tevens de toepasbaarheid vergroten door de ondersteuning van real-time systemen mogelijk te maken met taken die uit meerdere

uitvoeringsfasen bestaan en op verschillende doorstroomsnelheden werken. We laten zien dat deze benadering niet alleen geschikt is voor multi-rate en meerfasige applicaties, maar ook voor single-rate en enkelfasige applicaties, omdat de modellering met fasen ook kan worden gebruikt om de zeer strikte vereisten van de onderliggende hardware platformen te verminderen, welke zijn opgelegd door de bovengenoemde benaderingen. Met een aanpassing van deze aanpak maken we verder de analyse van applicaties die meer dan één hulpbron gelijktijdig gebruiken mogelijk.

Ten slotte bespreken we ook simulatie in de context van ons modelleerraamwerk. We presenteren de zogenaamde HAPI-simulator, die in staat is om elk soort real-time systeem met gedeelde hulpbronnen te simuleren. Naast andere toepassingen laten we ook zien dat HAPI kan worden gebruikt om analyseresultaten te falsificeren. Deze toepassing wordt mogelijk gemaakt door het feit dat we HAPI kunnen toepassen op dezelfde modellen als die we gebruiken voor analyse, wat een belangrijk verschil is t.o.v. gerelateerde aanpakken.

In het eerste deel tonen we met verschillende toepassingsstudies de inzetbaarheid aan van onze abstractie-verfijningstheorie en het onderliggende component model. De inzetbaarheid wordt getoond voor vele soorten discrete-gebeurtenissystemen en analysemodellen. In het tweede deel gebruiken we een WLAN-transceiver-applicatie als een doorlopende toepassingsstudie om aan te tonen dat al onze verbeteringen, d.w.z. met inachtneming van cyclische data-afhankelijkheden, iteratieve buffering, synchronisatiesignalen, uitvoeringsintervallen en multifase-analyse, resulteren in een significante verbetering van de nauwkeurigheid van de analyse.

Acknowledgements

As with all things in life, there was no straight line from the beginning of my promotion to the finalization of the thesis lying in front of you, but rather a winding path with many hurdles and crossings to overcome. And a good thing that is. To my opinion, it is the obstacles that one needs to vanquish which make a conclusion not merely the logical consequence of a beginning, but a true achievement. And it is the choices one makes, which do not only stipulate the outcome of a task, a project or an assignment. It is the choices one makes that make who one is.

Without you, my colleagues, friends and family, I could not have possibly overcome any of these challenges. Your support has helped me to surpass all the obstacles I had to face. And you gave me the possibility to make choices in the first place, not based on external circumstances, but on what I deemed right.

For that I want to say thank you.

First I would like to mention Marco. You have been my supervisor over more than seven years, and never have I regretted the decision to work with you. You have always found the right balance between challenging me and helping me to move forward, you have always encouraged me, inspired me, motivated me. You were not merely my supervisor, but indeed a true mentor. I thank you for giving me the opportunity to work with you.

I would also like to acknowledge Stefan and Joost, my former colleagues in our little Ph.D. enclave at NXP Research. Stefan, you have been the one who made our group an actual team, you have always extended a helping hand whenever needed, especially in the writing of my first papers. Joost, the discussions with you have always been a source of inspiration, most notably also for my very first paper, and helped in solving countless theoretical problems. And you have introduced me to the very useful concept of proof by intimidation provided me with the template that this thesis is based on. Thank you both for your help.

Of course I also want to mention the CAES group where I always felt welcome and where I have spent some truly memorable moments, particularly christmas

dinners. I would especially like to thank Gerard for offering me the opportunity to pursue my promotion in his group, as well as Thelma, Nicole and Marlous who were always helpful, in every possible way. And I also want to thank everyone at the group within NXP Research, who have not only given me a place to work, but who have always made me feel as part of the team.

Finally, I would also like to thank my friends and family.

Nemanja, ti si mi kroz cijeli moj studij bio drug, i u toku godina i postao pravi prijatelj. Hvala ti da si mi uvijek dao dobar savjet i da si uvijek bio spreman pomoći, bilo to u vezi sa doktoratom, sa našim zajedničkim projektom, sa mačkama ili kod bilo čega drugog. Okan, auch Dir vielen Dank für Deine langjährige Freundschaft. Insbesondere unsere interessanten Diskussionen waren für mich oftmals dringend benötigte Zerstreuung und auch nicht selten Inspiration. Yi-Chen, vielen Dank, dass Du immer ein offenes Ohr und einen guten Rat für mich hattest, als auch natürlich für Deinen Teil an den Abenteuern der *“Drei reiselustigen Drei”*. Bilge, you have quickly moved from being my friend's girlfriend to being a true friend on your own. I thank you for your support and look forward to continue ~~hanging out~~ working with you, as well as to the many ~~gin-tonic sessions~~ business meetings to come. Und Barbara, auch bei Dir möchte ich mich für Deine Hilfsbereitschaft und die interessanten Gespräche, vor allem auch zu den Freuden und Leiden der Promotion, bedanken.

Mama, ti si me uvijek podržavala, ti si me uvijek uhvatila kad sam pao, pa mi opet pomogla na noge. Ti si mi omogućila da studiram, da promoviram, da vodim jedan život po mojoj predodžbi, da si ispunim zbilja sve moje snove. Hvala ti, da si uvijek vjerovala u mene. Hans-Otto, Dirk, Sezinando und Irina, auch bei Euch möchte ich mich für all die Unterstützung bedanken, die Ihr mir über die Jahre habt zukommen lassen. Toni, hvala i tebi, da si mi uvijek ponudio pomoć i dobar savjet, da si uvijek bio uz mene. Sretan sam, i ponosan, da mi nisi samo brat, nego i pravi prijatelj. Dida, Baka, Lida i Natali, hvala i vama da ste me uvijek podržavali, da ste mi uvijek i na bilo koji način pomogli. Vi ste mi pokazali što riječ *“familija”* zbilja znaci. Gdje god i kad god smo se sreli, kod vas sam se uvijek mogao opustiti, kod vas sam se uvijek osjećao doma. Ne mogu vam reći kako sam sretan biti član naše male familije. Hvala vam za sve.

Zuletzt möchte ich mich auch bei Dir, Yi-Chin, von Herzen bedanken. Du hast die letzten Jahre so unglaublich viel Verständnis entgegen gebracht und warst mir die größte Stütze die ich mir je hätte erhoffen können. Ich habe es Dir wirklich nicht immer leicht gemacht und doch warst Du immer für mich da, hast mich nach Rückschlägen immer wieder aufgebaut, hast es noch immer geschafft mich zum Lachen zu bringen, hast mich in allem was ich mir vornahm unterstützt. Du bist meine Vergangenheit, meine Gegenwart und Zukunft. Auf letztere freue ich mich ungemein, auch weil ich hoffe, dass ich Dir in dieser zumindest ein wenig davon zurückgeben kann, was Du mir über all die Jahre gegeben hast. Vor allem aber freue ich mich auf ein großes Mehr an gemeinsamer Zeit.

Philip Kurtin
Eindhoven, October 2018

Contents

1	Preface	1
2	Overview	3
2.1	The Significance of Safety-Critical Systems & Guarantees	3
2.2	A Generic Classification of Analysis Approaches	8
2.3	Abstraction & Refinement for the Analysis & Design of Safety-Critical Systems	11
2.4	Analysis & Design of Concurrent Real-Time Systems	14
2.5	Simulation	18
2.6	Problem Statement	21
2.7	Contributions	22
2.8	Thesis Outline	24
I	An Abstraction-Refinement Theory	25
3	Introduction	27
3.1	Informal Description & Discussion	29
3.2	Related Work	36
3.3	Relation to Paper Versions	41

4	Timed Component Model	43
4.1	Ports & Streams	43
4.2	Stream Traces & Interfaces	46
4.3	Components	47
4.4	Component Monotonicity & Continuity	48
4.5	Component Graphs	49
4.6	Input Acceptance & Replaceability	51
4.7	Timed Dataflow Models	60
5	Abstraction & Refinement	65
5.1	Bounding	65
5.2	Inclusion & Exclusion	74
6	Case Studies	83
6.1	Reordering	83
6.2	Input Set Widening on Abstraction	84
6.3	Value Bounding	85
6.4	Removing Uncertainty & Non-Determinism	88
6.5	Removing Dependencies	90
6.6	Practical Example	93
7	Conclusion	97
II	Evaluation of Concurrent Real-Time Systems	99
8	Introduction	101
8.1	Basic Ideas	103
8.2	Related Work	117
8.3	Relation to Paper Versions	123
9	Modeling Concurrent Real-Time Systems	125
9.1	Applications & Application Models	125
9.2	Applications in the Timed Component Model	129
9.3	Constructing Analysis Models by means of Abstraction	133

10 Period-and-Jitter Analysis	141
10.1 Analysis Flows	142
10.2 Determining Bounds on Task Schedules	144
10.3 Computing Maximum Response Times	146
10.4 Buffer Sizing	152
10.5 Using Synchronization Edges for the Optimization of Schedules .	157
10.6 Proof of Conservativeness	159
10.7 Case Study	164
11 Execution Interval Analysis	175
11.1 Analysis Flows	175
11.2 Determining Bounds on Task Schedules	178
11.3 Computing Maximum Response Times	180
11.4 Buffer Sizing	192
11.5 Proof of Conservativeness	192
11.6 Case Study	194
12 Task Phase Analysis	203
12.1 Analysis Flow	203
12.2 Expanding CSDF Graphs to HSDF Graphs	205
12.3 Determining Bounds on Task Phase Schedules	207
12.4 Computing Maximum Response Times	210
12.5 Proof of Conservativeness	221
12.6 Multiple Shared Resources	225
12.7 Case Study	230
13 Simulation	239
13.1 Simulation & Abstraction	239
13.2 The HAPI Simulator	241
13.3 Case Study	243
14 Conclusion	251

15 Postface	255
III Appendix	257
A An Abstraction-Refinement Theory for Cyber-Physical Systems	259
A.1 Informal Description	259
A.2 Signal Component Model	260
A.3 Generalized Abstraction & Refinement	265
B Additional Proofs for Real-Time System Analysis Techniques	275
B.1 Termination of the Period-and-Jitter Analysis	275
B.2 Termination of the Execution Interval Analysis	279
B.3 Validity of the Multi-Phase Stop Criterion	280
List of Acronyms	289
List of Symbols	291
Bibliography	295
List of Publications	301
Propositions	303

chapter 1

Preface

When little Anna touched the stove she screamed. And then she learned.

Still drenched in tears, seven year old Anna vowed to never touch a stove again. Later, she elaborated this little piece of knowledge by learning that not the whole stove, but only the cooking plates can get hot, and that only if the little red lights next to them are burning. And then she had another revelation in realizing that the temperatures of the plates depend on the positions of the knobs below.

In her late twenties, now an engineer, Anna remembered her bad childhood experience and decided to do her duty to mankind by saving millions of other children from the same fate. At first, she simply imagined a stove that could not burn a child's hands by construction. After months of research, after many sleepless nights in the lab and after resolving hundreds of issues, she had finally transformed her initial idea to a real product. She had invented the induction cooker, a stove capable of heating specially designed vessels without the plates becoming hot themselves.

Albeit this little anecdote is fictional, you may have thought that it happened all the same. This is because you can understand Anna, you are able to follow her line of thought. You can comprehend her initial reaction, her generalization of a particular experience with one stove to all stoves, as well as her subsequent steps of detailing. Likewise, you can grasp the process of her invention, from her first idea of something entirely fictional to a physically real object.

Apparently, the reasoning in generalization, classification and simplification on the one hand and specification, itemization and concretion on the other is a fundamental feature of our way of thinking. And our language allows us to convey such thoughts from one to another.

Durant argues that it is the common noun allowing us to transform the specific to the general. He describes such nouns as the “*symbols of civilization*” that “*seem to grow in a reciprocal relation of cause and effect with the development of thought*”. As such, he rightly raises the question whether “*any other invention ever equaled, in power and glory, the common noun*” [Dur97].

It is paramount to notice that only specific things are real, whereas all generalizations are fictional. There is not *a* stove, there is only *the* stove. And grown-up Anna's idea of the cold stove is fictional all the more, as it has no correspondence in reality at the time of its conception. The abilities to imagine fictional things, independent of whether the fiction has a counterpart in reality or not, as well as to use language for the transmission of imaginations are two of the most powerful accomplishments of mankind. As Harari puts it, they are even distinctive features that separate man from beast:

"Yet the truly unique feature of our language is not its ability to transmit information about men and lions. Rather, it's the ability to transmit information about things that do not exist at all. As far as we know, only Sapiens can talk about entire kinds of entities that they have never seen, touched or smelled. Legends, myths, gods and religions appeared for the first time with the Cognitive Revolution. ... Ever since the Cognitive Revolution, Sapiens have thus been living in a dual reality. On the one hand, the objective reality of rivers, trees and lions; and on the other hand, the imagined reality of gods, nations and corporations." [Har15]

But what is it that enables us to construct generalizations from specific objects? And what allows us to derive specifications from generic imaginations? It is *abstraction* that paves the way from the specific to the general, as it is *refinement* that allows us to travel back from the general to the specific. Throughout our whole lives we abstract from reality to learn, to understand, to explain, while we refine from imaginations to build, to implement and to invent.

We require both abstraction and refinement to address the sheer complexity of reality. Without abstraction, we would not be able to learn anything at all, we would be stuck with Socrates' *"I know that I know nothing"* [Pla17]. And without refinement, we would not have a different means to invent than randomly assembling the existing. But every reduction in complexity also comes with a reduction in accuracy, with a detachment of reality. Anna, for instance, has initially abstracted too much when deciding to never touch a stove again and only subsequent refinements allowed her to come back to terms with the kitchen appliance. Hence in every abstraction and every refinement we must carefully trade off between accuracy and complexity.

Now you may ask what all these remarks have to do with the content of this thesis. Above all, they stress that the concepts of abstraction and refinement are not only innate to our human way of thinking, but a necessity for both the acquisition of knowledge from and the application of knowledge on reality. Whenever we reason about reality we implicitly apply abstraction and refinement. For real-time systems for which we need to derive guarantees instead of estimates, however, such an implicit reasoning is usually too error-prone and vague.

To that end, we present a new theory, a new *"language"* in the first part of this thesis that allows us to reason about abstraction and refinement for real-time systems in a very explicit, formalized way. In the second part we apply this theory on a relevant subclass of such applications, highlighting both its practical applicability and capability to seamlessly trade off accuracy and complexity.

chapter 2

Overview

In this overview chapter we first discuss the significance of safety-critical systems and the notion of guarantees in Section 2.1. Thereafter we introduce a method of categorizing different analysis approaches in Section 2.2 that can be used to derive guarantees for such applications, with the categorization being conducted in terms of accuracy and complexity. A framework that enables to seamlessly move on the trade-off between accuracy and complexity is introduced in Section 2.3 with respect to analysis models and in Section 2.4 with respect to analysis techniques. The latter also discusses the lack of accurate analysis methods for a very specific, yet important subclass of safety-critical systems and describes the basic idea of filling this gap. Section 2.5 examines simulation in the context of safety-critical systems.

We conclude this overview chapter with a problem statement in Section 2.6, a contribution overview in Section 2.7 and the outline of the remainder of this thesis in Section 2.8.

2.1 THE SIGNIFICANCE OF SAFETY-CRITICAL SYSTEMS & GUARANTEES

Safety-critical systems can be defined as applications for which the cost of a malfunction may be considered infinite (e.g. the cost may be death, severe destruction, irreparable damage to the environment or similar). Such malfunctions can be either functional (e.g. generation of wrong results) or temporal (e.g. deadline misses). For the latter, the temporal aspect of safety-criticality, the term hard real-time is coined. It is widely recognized that safety-critical systems are one of the most challenging types of applications when it comes to both their design and analysis. But what is it that makes analysis and design of such applications so tough?

First, imagine a simple text-processing application executed on a personal computer. If you type a text using this application then it is obviously desirable

that the typed text appears on screen immediately and exactly corresponds to what you have typed. But if it takes half a second for the text to appear or if occasionally, e.g. every 1000 letters typed, a sporadic error occurs, then this is merely an inconvenience, but nothing that cannot be coped with, nothing that cannot be easily corrected. Regarding the temporal aspect, one can consider such an application in the broadest sense as soft real-time, as the cost of a delay may be described as “*annoying*”, but certainly not a catastrophe.

Now consider an Anti-Lock Braking System (ABS), a system that is standard equipment in all modern cars. An ABS monitors the rotational speed of car wheels to detect whether the wheels have sufficiently good contact to the surface during a braking action. If this is not the case, the ABS relaxes the braking on the wheels to prevent locking. In contrast to the text-processing application, an ABS is clearly a safety-critical system: Here, a half-second delay or a sporadic error are not only inconvenient, but could decide over life and death. For that reason it is not sufficient to have an estimate on temporal behavior and functionality, but guarantees must be given instead.

Nowadays, ABS's are implemented using a single Electronic Control Unit (ECU), alongside with some sensors and actuators. Due to the low **complexity** of the ABS a rather simple micro-controller can be used. While analyzing and subsequently giving temporal and functional guarantees on such a system is not entirely trivial, it is certainly also not too difficult, given state-of-the-art analysis techniques.

However, for applications like autonomous driving systems things look quite different. Such applications are safety-critical in the same sense as the ABS, but have a much higher complexity: They have to account for much more variables (position of the own car from GPS, speed of the car, speed of surrounding cars, other surrounding objects, identified using radar or lidar systems, as well as some complex image processing), can initiate much more actions on different levels (follow route from work to home, platoon with some cars in front or change lanes, prevent a crash by increasing or reducing speed, as well as turning left or right) and make use of sophisticated decision-making processes, not rarely involving artificial intelligence and the training of neural networks.

Another source of complexity is the fact that such applications are usually executed in a **concurrent** fashion. The reasons for concurrency are mainly that many parts of safety-critical systems are naturally concurrent, such as the parallel processing and merging of multiple sensor inputs, and that complex applications cannot achieve the required performance if executed in a non-concurrent, i.e. single-threaded, fashion. In general, concurrency can be realized in two different ways: In a **time-driven** and a **data-driven** fashion.

Time-driven means that tasks are started strictly periodically and it is up to analysis to ensure that the periods and offsets between tasks are chosen in such way that new data is always available when a task is started. The advantage of this method is that it is easy to analyze, as tasks can be treated mostly in separation, and easy to implement if a suitable infrastructure is present. However, the latter

“if” must be emphasized, as time-driven applications require a global clock that is available in all subparts of an application. This may be quite easy to realize for single Systems-on-Chip (SoCs), but if you think about applications being distributed over e.g. all parts of a plane you get a different picture. Moreover, time-driven applications are not very robust, as deadline misses can result in the skipping or double-reading of data. And for applications in which tasks have varying execution times, a time-driven implementation can be highly inefficient as the period must be chosen larger than the largest possible execution time.

In contrast, tasks in a data-driven application are not started strictly periodically, but whenever data is available. Managing the availability of data throughout different tasks and different parts of an application requires sophisticated methods of **synchronization**, which can result in a significant overhead, and demands more advanced analysis methods as the behavior of tasks is more interdependent than for time-driven applications. On the other hand, data-driven applications are usually more efficient than their time-driven counterparts (if synchronization overhead is well-managed), because they allow for a mutual compensation of varying execution times and thus smaller periods. Moreover, they are more robust due to the same compensating effects and have less requirements on the underlying hardware platform as they do not require constructs like global clocks. In the following, the focus is mainly on data-driven applications.

Complexity in applications like autonomous driving systems is further increased by the presence of **cycles**, i.e. an application cannot be seen as an acyclic graph of separable tasks, but contains cycles, which can appear due to the following reasons: First, an application can contain feedback loops, that are for instance used in control, creating cyclic data dependencies. Second, multiple tasks of an application can access the same shared resource in a counter-dataflow order, resulting in cyclic resource dependencies. And third, an implemented application only has a limited amount of memory available. If two tasks communicate in a data-driven fashion over e.g. a First-In First-Out (FIFO) buffer, then the writing task only has a certain amount of buffer space available, i.e. it must stop writing if the buffer is full. This creates another type of cyclic data dependency between tasks communicating over buffers.

Giving guarantees on the temporal or functional behavior of such safety-critical, complex, concurrent and cyclic applications is anything but trivial. In fact, many analysis approaches are either not applicable for such applications (as discussed later, especially cyclicity is a major problem for many approaches), produce overly conservative results or have analysis run-times that make their usage for complex applications impractical.

Due to these reasons the problem of giving guarantees is regularly circumvented by so-called **over-provisioning**, a concept that can be intuitively explained as follows: Suppose that you have to attend a really important meeting at 9 am and you usually need 45 minutes to get to work. Starting from home at 8.15 am would be likely a bad idea because you could not be sure to arrive on time. Taking off half an hour earlier, i.e. at 7.45 am, appears to be a more appropriate

measure given the significance of the meeting. And this is just the essence of over-provisioning: Instead of replacing uncertainty of events by certainty, additional time or space are reserved to build up a margin of safety, effectively reducing uncertainty. Applied on computer systems, over-provisioning becomes over-dimensioning, which means that if e.g. a task has to meet a deadline and usually takes just about the deadline time to complete on a certain processor, a much faster processor is chosen. Or it means that if a bus barely satisfies a required minimum throughput, a much wider and / or faster bus is chosen, and so on.

At first glance, over-provisioning seems to be a reasonable measure to take. After all, we apply the very same concept in our everyday lives. However, over-provisioning has two significant drawbacks: Remaining uncertainty and cost.

For over-provisioning it is quite common to use rules of thumb as “*take a two times faster processor*”. But such measures are very questionable when it comes to safety-critical systems. Or would you like to fly with a plane whose engines are designed such that “*they should usually work*”? While over-provisioning reduces the likelihood of failure, it still does not give guarantees that failure cannot occur. For example, you could start at 7.45 am for your meeting and the unlikely event of a breaking train could still make you come too late (note that we are simplifying here, for many railways such as e.g. the Dutch it is rather inappropriate to speak of “*unlikely events*” with respect to malfunctions). Of course, you could try to dig deeper, make a probabilistic analysis on how likely you will arrive to your meeting on time, taking all kinds of factors like breaking trains into account. Conducting such an analysis, however, is usually even more complex than giving binary guarantees.

Or you could start even earlier, much earlier. But this would increase the second factor, cost, even more. For instance, if you went to work half an hour earlier than usual, you would effectively lose half an hour of sleep, i.e. your cost would increase the earlier you start. In computing terms, this means that if you employed a faster processor, your production cost would increase, as would chip area and power consumption.

As we are approaching the post-Moore era, in which scaling down cannot be relied upon anymore as a means to increase performance while keeping cost bounded, the cost factor becomes even more significant. In addition, safety-critical systems are seldom used in controlled environments with reliable high power sources, efficient cooling and unlimited space. This restricts the amount of over-provisioning that can be applied, even if monetary cost were negligible.

The latter observation becomes even more significant if the emerging Internet of Things (IoT) is taken into account. While the complexity of the subsystems that are employed in IoT applications is usually low, the overall system complexity can be tremendous. Moreover, IoT applications are designed to operate “*in the wild*”, using devices that are ultra-low-power, do not rarely depend on energy-harvesting, must often be unobtrusive (i.e. small) and communicate over unreliable channels such as bluetooth, Wireless Local Area Network (WLAN) or

even infrared. Nevertheless, such applications are bound to work in a collaborative, yet reliable (i.e. safety-critical) fashion. These aspects severely limit the applicability of over-provisioning. For example, it is much easier to replace a processor with a faster one if the device is wall-plugged and not powered via e.g. solar panels and if the dimension of the device is not a concern, it is much easier to replace an on-chip bus with a wider one than increasing the bandwidth of radio communication, and so on. As a consequence, giving guarantees instead of using excessive over-provisioning gains even more importance.

Given these relations between over-provisioning, uncertainty, cost and safety-criticality another facet deserves mentioning: The treatment of inherently non-safety-critical systems as safety-critical for economic reasons. For instance, consider two H.264 decoders that are used in TV systems. The first decoder is designed for meeting an average throughput, but sometimes misses a deadline, such that a flicker occurs once an hour on average. In contrast, the second decoder is treated as a safety-critical system, such that it always meets the required throughput and consequently never flickers. From a functional point of view, you would likely prefer a TV with the second decoder. But for consumers usually not only functionality, but also price matters. If the second system required a large amount of over-provisioning (a much faster processor, 64 bit instead of 16 bit bus, and so on) to prevent flickering, then the TV system with the better decoder would likely also have a significantly higher cost. The at first glance apparent choice would suddenly become a trade-off between cost and functionality, such that the first TV system with the worse decoder may be the better choice for most consumers after all.

The point to be taken here is that lots of nowadays' non-safety-critical systems would benefit from being treated as safety-critical systems, but the associated cost due to over-provisioning regularly renders such a treatment economically unjustifiable. By finding approaches that allow to give guarantees on applications instead of rule-of-thumb over-provisioning one could however close this cost gap, significantly increasing the class of applications for which safety-critical analysis methods would be applicable and thus relevant.

After having defined the class of complex safety-critical systems and having evaluated the significance of giving guarantees on the behavior of such applications, it is only appropriate to dedicate a few words to the significance and meaning of the term guarantee itself.

The term guarantee with respect to real (implemented) applications should be taken with a grain of salt. After all, there is no such thing as a guarantee in the real world. For instance, a computer system that is guaranteed to provide results within a certain time can only satisfy this guarantee if it is plugged to a fitting power source, if there is no failure of circuits due to overheating, no wear-and-tear, no significant amount of cosmic rays, if it does not burst into flames together with the building in which it is located, if the world is not destroyed by an alien attack, etc. Consequently, all guarantees that can be given are not given with respect to reality itself, but actually only with respect to a **model of**

reality, i.e. a model that conforms to reality within certain bounds. Such a conformance relation between reality and a model of reality can be established using hypotheses like a fail hypothesis (the system in question is working correctly) or a load hypothesis (the load on certain components of the system does not exceed a certain boundary).

Now you may question whether it even makes sense to try to give guarantees on something that is inherently uncertain. The point to be taken here is simply that giving guarantees on a model of reality that conforms to reality within bounds removes a degree of uncertainty compared to giving estimates on a model of reality. Or in other words, a guarantee on an estimate is better than an estimate on an estimate. Taking additionally into account that the estimates usually given for over-provisioned applications rather adhere to rules-of-thumb than being accurate stochastic assessments, one can see that a guarantee on a carefully designed model of reality (in the sense that no devious assumptions are made) is of an entirely different quality.

That being said, we can now move on to discuss the several challenges and obstacles that lie ahead on the long and winding path to a conservative, efficient and nevertheless accurate analysis of complex safety-critical systems. But first we need to define what terms such as conservative, efficient and accurate actually mean. Precisely defining these terms and thereby specifying a concise framework for the classification and relation of different analysis approaches is consequently subject of the following section.

2.2 A GENERIC CLASSIFICATION OF ANALYSIS APPROACHES

Any analysis approach can be divided into two components: An underlying analysis model that contains all the information used by the analysis approach and an analysis technique working on this information (see Figure 2.1). In the following, we classify different analysis approaches with respect to the type of the analysis approach, the entropy of the analysis model and the effectiveness and efficiency of the analysis technique. Roughly speaking, the type relates to the kind of guarantees given by an approach, the entropy to the amount of information contained in the underlying analysis model, the effectiveness determines how well this information is exploited by the analysis technique to provide guarantees and the efficiency to the resource-usage of the technique to obtain its effectiveness.

With respect to **analysis approach types** we can differ between approaches giving temporal guarantees, such as guarantees on maximum end-to-end latency or minimum throughput, and functional guarantees, such as result accuracy, control stability, etc.

In the end of the last section we have assessed that there are no guarantees in reality. This assessment can be also interpreted such that reality has an infinite **analysis model entropy**, as a potential analysis model representing reality as a whole would contain an infinite amount of information. On the other side of

Figure 2.1: Accuracy and complexity of analysis approaches.

the spectrum, we can define an analysis model labeled “*undecided*”, which can be characterized by the sole fact that it contains no information. Consequently, reality and undecided mark the endpoints of an imaginary entropy scale (as indicated by the vertical axis in Figure 2.1) on which we can position all analysis models discussed in the following.

That a certain amount of information is contained in an analysis model does not mean that this information is also fully exploited by the analysis technique working on this model. Thus, we can further classify analysis approaches by the **effectiveness of the applied analysis techniques**. Given a certain analysis model, let us call an analysis technique effective if it determines the tightest possible guarantees that can be given for the model. Analogously, let us call a technique ineffective if it derives the loosest possible guarantee for the model (for instance, the loosest possible guarantee on an end-to-end latency would be that “*the latency is smaller or equal to infinite time*”). This allows to position all analysis techniques on an imaginary effectiveness scale (as indicated by the horizontal axis in Figure 2.1).

And now comes the tricky part: No matter how effective an analysis technique, i.e. how tight or loose the provided guarantee is, it must still provide a guarantee on the underlying analysis model, as it is required by the type of analysis approach. This implies that the only guarantee (no matter whether it is a functional or a temporal one) that can be given by an analysis technique working directly on reality is the loosest possible one. For instance, for an analysis approach that shall provide a guarantee on end-to-end latency and that uses reality as analysis model, it does not matter which technique is used by the model, as the only guarantee that can be given would be the loosest one, i.e. “*the latency is smaller or equal to infinite time*”. This is due to the fact that the amount of information that must be considered by the technique to provide any guarantee is infinite. Interestingly, if we consider the other side of the entropy spectrum, i.e. an analysis technique working on the “*undecided*” analysis model, we obtain the same result: For an

analysis approach that attempts to determine an end-to-end latency guarantee on the “*undecided*” analysis model the tightest guarantee that can be given is again the loosest one, i.e. “*the latency is smaller or equal to infinite time*”.

The bottom line here is that analysis techniques working on either reality or the “*undecided*” model are bound to be ineffective. Consequently, any analysis approaches that we consider in the following can be positioned on the entropy scale in between reality and “*undecided*”.

Finally, the **efficiency of an analysis technique** reflects the amount of resources that is consumed by a technique to achieve its effectiveness. For instance, a technique can make use of fixed point arithmetics to obtain its results, or it can use floating points. If the results are the same, then the fixed point one will likely be more efficient than the floating point one because the same guarantees can be given using less resources (in this case less processing time).

In the following we use these metrics, type, entropy, effectiveness and efficiency, to relate different analysis approaches to each other. First, we only compare analysis approaches with respect to the same guarantees. This means that if for example an analysis approach X gives guarantees on both end-to-end latencies and throughput and an approach Y only guarantees on throughput, we only compare these approaches with respect to their throughput guarantees. And if an analysis approach Z gives guarantees on control performance, we do not compare it to the other two approaches. Second, for two analysis approaches of the same type, we call an analysis approach X more entropic than an analysis approach Y if the analysis model used by X has a higher entropy than the model used by Y. Third, for two analysis approaches of the same type working on the same analysis model, we call an analysis approach X more effective than an analysis approach Y if the respective analysis technique used by X is more effective than the technique of Y. And fourth, for two analysis approaches with the same type, model and effectiveness, we call an approach X more efficient than an approach Y if it consumes less resources than Y to obtain the same results.

Directly following from entropy and effectiveness are two other key metrics that can be used to relate different analysis approaches: **Accuracy** and **complexity**. In general, accuracy assesses the tightness of guarantees not only for analysis approaches working on the same analysis model, but also for analysis approaches using different models. Similarly, complexity assesses the difficulty to derive such guarantees and thus indirectly also the required computational time, again for analysis approaches using either the same or different models. The actual required computational time is determined by the efficiency of the approach, which, according to above definition, reflects the resource usage of an approach given a certain level of complexity.

For two analysis approaches that provide the same guarantees and work on the same analysis model, an approach X is both more accurate and more complex than an approach Y if approach X is more effective than Y (e.g. an approach X using model 3 and technique A as opposed to an approach Y using model 3 and technique B in Figure 2.1). Likewise, for two approaches with the same

effectiveness, an approach X is both more accurate and more complex than an approach Y if approach X is more entropic than approach Y (e.g. an approach X using model 3 and technique A as opposed to an approach Y using model 4 and technique A in Figure 2.1). And if two approaches have the same complexity because they work on the same models with techniques of the same effectiveness, it holds that an approach X is more efficient than an approach Y if X uses less resources (e.g. processing time or memory) to obtain the same results as Y.

These relationships illustrate that accuracy and complexity are essentially two sides of the same medal (see diagonal axis in Figure 2.1). It is impossible to increase accuracy without increasing complexity as well. Consequently, a trade-off between the two must be made. As already indicated in the preface, providing methods and techniques to enable a seamless choice of this trade-off is the goal of this thesis.

In light of this, the next section answers the question how models can be constructed that are more or less entropic than others, while the section thereafter focuses on the derivation of more or less effective analysis techniques for the same models.

2.3 ABSTRACTION & REFINEMENT FOR THE ANALYSIS & DESIGN OF SAFETY-CRITICAL SYSTEMS

Given that a useful analysis on reality is impossible, it is paramount to derive analysis models that have a sufficiently high entropy (that are close enough to reality) such that accurate analysis results can be provided and that likewise have a sufficiently small entropy such that complexity does not explode, i.e. that analysis results can be provided in a sufficiently small time frame. To determine such analysis models, a process called **abstraction** needs to be applied. Intuitively, an abstraction can be seen as a step from one model to another, such that the latter model contains less information than the former. A **refinement** is the opposite of an abstraction, i.e. it represents a step from one model to another, such that the latter model contains more information than the former. Note that with this definition reality can be also seen as a model.

In the following we differ between three different types of abstraction: **Exclusion**, **inclusion** and **bounding**. Now this differentiation may seem counterintuitive at first, as exclusion and inclusion look like opposing concepts. The key point here is, however, that all three can be used to remove information, i.e. to reduce entropy.

To illustrate the various types of abstraction let us reuse the appointment example from the first section. In an effort to make the problem manageable, we can break down the problem into subproblems, e.g. into eating breakfast, walking to the departing train station, taking the train, walking from the arriving train station to work. Let us now only focus on the “*taking the train*” part.

As “*taking the train*” is a real-world action, there is an infinite number of scenarios that can occur: The train can be on time, it can be delayed due to more

people than usual trying to board the train, there can be construction works on the track, the train could break due to a technical defect, it could get hit by an asteroid, and so on. As discussed in the first section, we consequently have to create a model of reality to make the problem analyzable. For that purpose, we remove all unlikely scenarios such as construction works, mechanical defects, asteroid hits, etc. And this is just the essence of exclusion abstraction: The entropy of the problem is reduced by excluding scenarios (see model 1 in Figure 2.1).

After this pruning step we are left with a reduced number of scenarios. In more detail, such scenarios could be *“if x people are waiting to board the train, then the train will take y minutes and z seconds from station to station”*. For brevity, let us use tuples $(x|y|z)$ to describe such scenarios. Now suppose that only the following scenarios exist in reality: $(97|27:25)$, $(55|25:43)$ and $(13|26:12)$ (note that usually a lot more scenarios exist in reality which we omit here for simplicity).

To further reduce entropy we can as well add additional scenarios, such as $(97|25:43)$, $(97|26:12)$, $(55|27:25)$, $(55|26:12)$, $(13|27:25)$ and $(55|25:43)$. These scenarios obviously do not occur in reality, but because all original scenarios are considered as well we can be sure that if we did an analysis on this model no real scenarios would be ignored. After the addition of these scenarios, we can describe all possible scenarios in a single scenario $(97 \text{ or } 55 \text{ or } 13|27:25 \text{ or } 25:43 \text{ or } 16:12)$, which translates to *“if either 97, 55 or 13 people are waiting to board the train, then the train will take either 27 minutes and 25 seconds, 25 minutes and 43 seconds or 16 minutes and 12 seconds from station to station”*.

So what has just happened here? In short, we have added additional behaviors and thereby replaced uncertainty (it is unknown how many passengers will wait for the train, thus the time station-to-station is also unknown) by non-determinism (for any number of passengers there can occur any time station-to-station). By doing that, we have vastly reduced the entropy of the problem, only this time not by excluding scenarios, but by including scenarios. This is inclusion abstraction (see model 2 in Figure 2.1).

Lastly, we must take into account that many analysis approaches cannot be applied on non-deterministic analysis models. For that reason we can make use of the third type of abstraction, bounding. Since we are interested in an upper bound on the time needed from station to station, we should use upper-bounding. Applied on our example, the scenario $(97 \text{ or } 55 \text{ or } 13|27:25 \text{ or } 25:43 \text{ or } 16:12)$ could be upper-bounded by $(97 \text{ or } 55 \text{ or } 13|27:25)$ (as illustrated with model 3 in Figure 2.1), or, if integers are preferred, even by $(97 \text{ or } 55 \text{ or } 13|28:00)$. Apparently, none of the original scenarios is reflected anymore on the level of abstraction corresponding to the latter bounding (as indicated with model 4 in Figure 2.1). But this does not matter since the remaining behavior is an upper bound on all the underlying behaviors.

Doing a temporal analysis on this now deterministic analysis model is easy, much easier than an analysis on the other, uncertain or non-deterministic abstraction layers. If the analysis technique used were effective, it would give us a station-to-station time of 28:00 minutes. This is a conservative upper bound on

all considered scenarios, i.e. we can finally give the guarantee that “*if the train is working normally (exclusion abstraction) then the train will not take more than 28:00 minutes from station to station (inclusion and bounding abstraction)*”.

What we have described here in a rather intuitive fashion is the essence of conducting analyses based on abstraction. For this simple example, it was quite easy to derive the necessary abstractions and to show that these abstractions are indeed conservative in the sense that the station-to-station time did not get underestimated after each abstraction step. However, showing the same for more complex scenarios is usually not so straightforward.

For instance, we have already initially assumed that the **component** that describes the train-ride can be separated from other components of the **component graph** describing the whole way to work, such as “*walking from home to departing station*” or “*walking from arrival station to work*”, and thereafter treated the abstraction of the train-ride in isolation. But can we be sure that an increased station-to-station time does not result in a shorter overall time from home to work and vice versa? E.g. could it not be that if the train arrived late (more than 27:30), you would rather take a taxicab (5:00 travel time) instead of walking to work (10:00 travel time)? In this case it would follow that if we “*connected*” the upper-bounded train-ride component to the original “*walking from train station to work*” component we would obtain an overall result of $28:00 + 5:00 = 33:00$ minutes, which is clearly less than the maximum time of $27:25 + 10:00 = 37:25$ minutes that can occur in reality (after exclusion).

Now one may ask whether it would not make more sense to treat the appointment example as a whole and reduce entropy solely by abstraction. However, given that each abstraction step results in a loss of accuracy, i.e. in a widening of the gap between reality and model, it appears inevitable to separate an application into components as another measure of entropy reduction. Moreover, applications are usually already built in a modular fashion and it should be possible to connect existing applications to others, without having to start with abstraction and analysis on each new connection all-over again.

On top of that, we have so far only talked about the analysis of existing applications, which corresponds to a bottom-up perspective from reality to an abstract analysis model. However, when it comes to the design of new applications, one would like to conduct a top-down approach from a specification model to reality in the same way. In terms of our appointment example this would mean to start with a fairly abstract requirement like “*arrive at work on time*”. From thereon, we would then aim to split the problem into subproblems and to refine the different subproblems via exclusion, inclusion and bounding down to reality. However, neither analysis nor design approaches are usually strictly bottom-up or top-down, but more often a mix of the two. For instance, it could happen that a certain specification cannot be realized, such as e.g. getting to work in 10 minutes while already the train part always takes more than 25 minutes. In this case, one would like to move up again, using abstraction and analysis, to determine a more realistic specification. These observations advocate to derive a theory that is equally suitable for bottom-up analysis and top-down design approaches.

Lastly, another problem with existing analysis approaches is their incompatibility. There may be a good analysis approach to analyze the train part of the way to work, and another approach better suitable to analyze the walking parts. Each of such approaches has its own analysis model that is usually incompatible to others. Without a common framework that allows to combine different analysis models it would be impossible to give guarantees on the whole application. For example, one method may find a 28 minutes upper bound on the train-ride part and another a 5 minutes upper bound on the “*walking to work*” part. How could one possibly guarantee that a total upper bound on the way to work is a combination of the two, without a common denominator of the respective analysis models?

All these considerations suggest the necessity to develop an **abstraction-refinement theory** that is equally suitable for the analysis and the design of safety-critical systems. For that matter a **component model** would be needed that is expressive enough to allow both the representation of a variety of analysis models, as well as reality itself (and that both in the functional and temporal domains) and that supports composition, hierarchy and hiding to enable the breakdown of complex problems into subproblems. On top of that, an **abstraction-refinement theory** would be required that allows for exclusion, inclusion and bounding abstraction and refinement on the component model, in such a way that abstraction and refinement on component level are automatically lifted to graph level.

As discussed later, existing works fail to combine these requirements in a single theory. The derivation of a component model and a corresponding abstraction-refinement theory satisfying all aforementioned requirements are consequently subject to the first part of this thesis.

2.4 ANALYSIS & DESIGN OF CONCURRENT REAL-TIME SYSTEMS

So far we have only discussed the various ways of abstraction that can be used to derive analysis models suitable for the application of analysis approaches. But the best analysis model is of no use if no suitable techniques exist to analyze these models. In the following we discuss the analysis of a particularly challenging type of applications: Cyclic concurrent hard real-time streaming applications with shared resources.

Let us go through the different characteristics of such applications. **Cyclic** can be understood in the way introduced in the first section, i.e. it means that one or more of the following three conditions apply: First, the application contains functional cyclic data dependencies like feedback loops. Second, the application operates on a limited amount of memory, for instance via FIFO buffers of limited capacities, such that a reading task does not only have to wait if a buffer is empty, but also a writing task has to wait if the buffer is full. And third, the application may contain cyclic resource dependencies, such that there are shared resources

between different tasks that are granted to the tasks in an order opposite to the flow of data.

The term **concurrent** implies that the application is not necessarily executed in a sequential manner, but may contain parts that are executed in parallel. As discussed before, we adopt a data-driven point of view, assuming that tasks communicate via constructs such as FIFO buffers and are executed on the arrival of data.

Hard real-time is also a term that we have already discussed in the first section, denoting that the application is used in a safety-critical context that must be treated accordingly. In particular, hard real-time implies that it is required to give guarantees on temporal properties of the application, such as a guarantee on the minimum throughput or a guarantee on the maximum end-to-end latency.

The term **streaming** refers to the fact that the considered applications are not only passing single values from one task to another, but entire streams. A stream can be defined as a finite or infinite sequence of events, with the events being characterized by a timestamp and a value (and possibly an index if a reordering of data should be accounted for). This notion of streams allows to characterize literally any type of event-driven applications.

Lastly, the explicit consideration of **shared resources** implies that the tasks of an application are not solely dependent on the data that is transferred from one task to another, but also dependent on shared resources. A shared resource can be for instance a processor on which multiple tasks are competing for processing time, but also a shared memory, a bus or a peripheral. The manner in which a shared resource is granted or denied to a task is determined by a so-called **scheduling policy**.

For some scheduling policies such as Time-Division Multiplexing (TDM) or Round-Robin (RR) it holds that the interference of different tasks competing for the same resource can be abstracted away without determining an enabling rate characterization of the competing tasks, i.e. without analyzing how often the competing tasks are accessing the resource. For other scheduler types, such as **Static Priority Preemptive (SPP) scheduling**, this is not possible. The reason that enabling rate characterizations of tasks are required for SPP is that a scheduler belongs to the class of “*non-starvation-free schedulers*”, which is defined by the characteristic that a rogue high priority task could access the resource forever, resulting in starvation of all lower priority tasks. Because of this circumstance SPP resembles one of the most challenging scheduler types, which is why most of the second part of this thesis focuses on this particular type of scheduler.

With the increasing complexity of safety-critical systems, the increasing necessity of flexibility, which favors Software-Defined Radio (SDR) implementations over custom chips, and the wide usage of SPP schedulers, especially in the automotive domain, this challenging class of applications becomes more and more relevant. One particular example of such an application is the decoding mode of a WLAN 802.11p transceiver that is used for car-to-car or car-to-base communication. It contains a feedback loop that is used for channel estimation and uses

FIFO buffers with limited capacities, which both make it cyclic, it is executed on a multiprocessor system with shared resources for performance reasons and it is hard real-time as it is used as part of safety-critical systems such as automated braking systems. In short, it combines all aforementioned challenges, making it a suitable reoccurring case study for the second part of this thesis.

Interestingly, none existing related works are able to provide a fast and accurate analysis of such applications.

First, there are analysis approaches that model such applications in detail and use model checking to provide analysis results. However, the major downside of such approaches is the high complexity resulting from the huge state space that needs to be analyzed, such that even for simple applications analysis run-times are often in the magnitude of days or weeks. Especially when it comes to the design of complex applications, in which analysis is not only required once, but must be repeated on each modification, such analysis approaches appear to be impractical in most cases.

Second, there exist many approaches from the hard real-time domain that are capable to provide an accurate and fast analysis of applications with shared resources and SPP scheduling. But these approaches all have in common that they lack the ability to handle cycles in an accurate manner, making them inapplicable for the described type of applications.

Third, dataflow analysis approaches natively support the analysis of concurrent and cyclic applications, but only have limited support for resource dependencies. In fact, they only support starvation-free scheduler types such as TDM or RR for which interference can be bounded without knowledge of the enabling rate characterizations of competing tasks. This circumstance also makes solely dataflow-based approaches inapplicable for applications such as the WLAN 802.11p transceiver.

Finally, a recently introduced combination of dataflow and real-time analysis appears to be more promising, as it enables the fast analysis of cyclic real-time applications scheduled via SPP schedulers. However, the approach is very coarse in the sense that the used analysis technique is highly ineffective, resulting in a low analysis accuracy. Increasing the effectiveness of this approach is consequently to a large extent subject to the second part of this thesis.

The methods that are presented later in more detail can be subsumed as follows: The first and largest improvement of analysis accuracy must be attributed to the exploitation of the effect that **cycles** do not only make analysis more challenging, but that they also **limit interference**. To give a rather intuitive example, assume that two tasks are executing on the same processor and are at the same time communicating over a FIFO buffer with the capacity one. This buffer prevents the reading task to execute while the writing one is executing because the buffer is empty until the writing task finishes. Likewise, the writing task cannot execute as long as the reading one is executing because it has to wait until the buffer becomes free again at the end of the reading task's execution. The buffer makes the two tasks effectively mutually exclusive. Taking this mutual exclusivity into

account allows to conclude that interference between these two tasks can never occur.

In a second step, this observation is taken to another level by considering that the interference limiting effect breaks the usual monotonicity between buffer capacities and throughput, i.e. that larger buffer capacities result in a larger throughput. So far, analysis and buffer sizing were always treated in separation, i.e. first analysis was conducted assuming infinite capacities. Thereafter, buffer sizes were determined that were large enough such that the analysis results were not affected. Instead of this two-step approach, we introduce an **iterative buffer sizing** that begins with the smallest possible buffer capacities and allows to exploit the aforementioned interference limiting effect. This enables larger throughput guarantees at smaller buffer capacities, i.e. a larger worst-case performance at smaller area cost - a win-win situation for application design.

Moreover, the original approach makes use of a rather crude method called period-and-jitter to estimate interference. In a third step, this method is thus replaced with a more accurate interference characterization that makes use of so-called **execution intervals**, intervals that allow to describe interference between different tasks with a significantly higher accuracy.

Note that these steps just describe the essence of increasing effectiveness: The used analysis models are the same throughout all steps, but more information is taken from the models to increase effectiveness and thereby accuracy, at the cost of an increased complexity (as with techniques A and B both applied on model 4 in Figure 2.1).

In a last step, we change this paradigm by moving away from the notion of inseparable tasks. Instead, the specification of several **task phases** of varying lengths and multiple rates is allowed. The approach does not only change the analysis technique, but also makes use of different, less abstract and thus more entropic analysis models, enabling the use of more complex, but also more accurate analysis approaches. This comes with a variety of advantages: It allows to specify tasks in a more relaxed fashion (with the previous approaches it was required that all tasks synchronize on their buffers atomically at the beginnings and ends of their executions, this restriction is removed with the introduction of phases), it allows to capture interference more accurately and it allows the consideration of tasks that inherently consist of multiple phases and that operate with multiple rates.

For the latter approach we first use a period-and-jitter interference characterization, in combination with an explicit consideration of cyclic data dependencies. This version of the approach has the same effectiveness as the first improvement step (the period-and-jitter analysis for inseparable tasks), but operates on a more entropic analysis model (a model considering task phases specifically instead of inseparable tasks). Hence the period-and-jitter analysis for task phases is more accurate than the period-and-jitter analysis for inseparable tasks, but not necessarily more accurate than the analysis approaches for inseparable tasks with iterative buffer sizings and / or execution intervals, as the latter use both more

effective analysis techniques and less entropic analysis models (this relation corresponds to e.g. technique A being applied on model 4 vs. technique B being applied on model 3 in Figure 2.1).

Thereafter we also introduce execution intervals with precedence constraints to characterize interference in the analysis of multi-phase applications.

Finally, also the **joint consideration of multiple shared resources** such as shared processors, shared memories, shared peripherals, etc. is enabled by the multi-phase treatment, allowing the specification of entire applications without requiring accuracy-reducing abstractions that remove some or all resource dependencies.

As illustrated in the accompanying case studies, all these approaches have in common that they increase applicability and analysis accuracy significantly, while keeping complexity sufficiently low to provide analysis results quickly. This makes them all suitable candidates to trade-off accuracy and complexity in a seamless manner, depending on the particular prerequisites specified by the user.

2.5 SIMULATION

Up to this point we have only discussed analysis as a means to determine the temporal and / or functional behavior of existing or newly designed applications. Besides, another widely used concept also deserves mentioning: The **simulation** of applications.

Like analysis, simulation makes use of application models to capture certain aspects of reality. The main difference between analysis and simulation lies in the way the respective models are treated: For analysis, a deep understanding of the model is required to obtain useful results. In contrast, simulation treats a model like a black box, it does not need nor have a view on the internals of a model, but only determines the reactions of the model on its inputs. A model is thus not analyzed, but **executed**, resulting in **simulation traces** that consist of the inputs and outputs of a single simulation run. Note that unlike analysis, simulation can be also applied directly on reality, only that one does not call this simulation anymore, but **testing**.

The obvious advantage of simulation over analysis is the speed at which results can be obtained: Instead of analyzing a model thoroughly, which does not rarely come with an exponential computational complexity, one can simply build a model, thereafter execute this model one or more times and therewith obtain simulation traces that ideally reflect reality.

Ideally, that is. But this perspective entails multiple issues.

The fundamental problem of simulation-based approaches is that only **exhaustive simulation**, i.e. a repetitive execution of simulation that ensures that all possible traces for a model are determined, can be used to give guarantees. Otherwise it can never be ruled out that some corner cases exist which are not captured

by an estimate based on traces. Depending on the entropy of the model an exhaustive simulation can take a very long time, or even infinite time if the number of possible traces is infinite. For instance, models that make use of continuous time inherently have an infinite number of temporal traces because already the number of possible inputs is infinite. The same can hold for models that have internal states. On top of that, exhaustive simulation is generally only applicable for deterministic models. This is due to the fact that a deterministic model can be fed all possible inputs, which ensures that all possible outputs are determined. For a non-deterministic one, however, even feeding all possible inputs does not guarantee observation of all outputs, i.e. it is never guaranteed that a simulation is indeed exhaustive.

Moreover, just like analysis results, also simulation results do not represent results with respect to reality, but only results with respect to the underlying model. This further impedes interpretation of results if the model is not constructed based on clearly defined mechanisms such as exclusion, inclusion and bounding abstraction. This is unfortunately often the case. And because simulation is often used as an alternative to analysis, the models used for simulation are rarely the same or even related to the models used in analysis, preventing comparability between simulation and analysis results.

Taking these observations into account, it appears that simulation is quite suitable to determine the typical behavior of an application, but less to obtain guarantees with respect to best-case or worst-case behaviors. With that in mind, you may ask whether simulation even make sense in a safety-critical context.

In the “classical” usage - as an alternative to analysis - it does not. As exhaustive simulation is inapplicable in most cases, simulation can seldom be used to derive the guarantees required for safety-critical systems. Moreover, the methods discussed in the previous sections enable a seamless trade-off between analysis accuracy and complexity, such that the main advantage of simulation, speed, becomes less relevant. The alternatives are not anymore accurate guarantees after a long time with analysis vs. estimates after a short time with simulation, but inaccurate guarantees after a short time with analysis vs. no guarantees with simulation.

Now you may ask: Why do you devote an entire chapter to a topic that you consider useless in the context of this thesis? Rest assured that it’s not ~~only~~ to squeeze in yet another paper. The main reason is that while simulation is not a viable alternative to analysis, it still has its strengths in combination with analysis. These strengths are threefold.

First, it must be acknowledged that both the construction of analysis models and their actual analysis are error-prone processes, even though the formally defined methods of abstraction and refinement help to mitigate this issue.

And this is where simulation steps in: For simpler models, i.e. models with a rather low entropy, exhaustive simulation can be used for the **verification** of analysis results. But as the class of models for which exhaustive simulation is applicable in a reasonable time frame is limited, another application of simulation

in the context of analysis becomes even more relevant: The **falsification** of analysis results. This means that simulation is not applied exhaustively, but only a limited number of times, such that flaws in the analysis can be recognized before an application is actually implemented and deployed in its safety-critical environment. If no flaws are found after extensive (not exhaustive) simulation, this can consequently increase **confidence** in the used analysis.

Second, given that exhaustive simulation is applicable it cannot only be used for validation, but also to measure effectiveness of analysis techniques. This is due to the fact that exhaustive simulation always determines the most accurate result possible for a given model, i.e. it always has maximum effectiveness, thereby defining a benchmark for all analysis techniques applied on the same model (see simulation in Figure 2.1).

And third, it must be also recognized that while exhaustive simulation often cannot be used for entire applications, this does not have to be the case for its parts, or components. To illustrate this concept, let us exert our well-known appointment example one last time.

Without going into too much detail, it can be assumed that the appointment example as a whole is time-continuous, stateful and non-deterministic. To apply exhaustive simulation, the only kind of simulation that can be used to derive guarantees, it would be necessary to abstract the complete application such that it becomes time-discrete, stateless and deterministic. While this is possible, it would very likely come with a significant loss in accuracy.

A more elegant usage of exhaustive simulation would be to apply it only selectively, on parts of the problem. For instance, one could apply analysis for the “*taking-the-train*” part and also use analysis to consider the combination of all parts, but assess the walking parts using exhaustive simulation.

As an example, let us consider the “*walking to the train station*” part. Assume that there are three traffic lights on the “*walking to the train station*” part. Further assume that it has been determined that you are walking with a constant speed and that there are no other obstacles on your way to the train station than traffic lights, such that it takes you 5 minutes to the train station if all lights are green. Moreover, assume that a bounding abstraction step has been taken to discretize the originally continuous time to one second intervals.

Determining an accurate upper bound on the time from home to station is not trivial, but the alternatives that can occur in the described model are certainly limited, they solely depend on the durations of red and green phases, as well as the arrival times at the respective traffic lights. Now one could of course derive a very effective analysis technique for this particular model, which would even make sense if lots of walking-with-traffic-lights applications had to be analyzed. But if it is only about determining an upper bound for this particular application, an exhaustive simulation of this model might be the much faster alternative - not fast in the sense that exhaustive simulation is faster than the analysis technique, but applying simulation is faster than deriving a novel, specialized technique.

Given exhaustive simulation, a derived upper bound on the walking time of e.g. maximally 7 minutes would be a guarantee and thus be equivalent to a result determined via analysis. Consequently, the simulation result could be thereafter used in combination with other simulation or analysis results, like the upper bound of 30 minutes determined for the train part, such that eventually a guarantee on the whole way-to-work application could be given.

When it comes to computing, an equivalent example would be the determination of bounds on task execution times that are later used in a model of the whole application, taking communication and interference effects between tasks into account. Note that in this case not even bounding abstraction has to be used for discretization, as unlike an entire application which may be embedded in a physical environment or at least may contain multiple processors running asynchronously, single tasks on single processors are usually not defined in the continuous time domain, but in discrete time - with the smallest unit of time being a processor tick.

These three points justify the usage of simulation in a safety-critical context, only not as an alternative, but complementary to analysis. Finally, let us try to put simulation into perspective by answering the following question: How does simulation fit into the context of trading off accuracy and complexity?

As discussed above, only exhaustive simulation can give guarantees and if exhaustive simulation is conducted then these results are the same as of an analysis technique with maximum effectiveness. For the same model, simulation consequently cannot be used to trade-off accuracy and complexity. But if simulation can be applied on different models, with different entropies, the trade-off can still be drawn - only with one instead of two degrees of freedom as in the case of analysis.

However, this requires that simulation can be applied on the same models as analysis and that these models adhere to the same concepts of exclusion, inclusion and bounding abstraction and refinement. We present a simulation framework by the end of this thesis that just conforms to these requirements.

2.6 PROBLEM STATEMENT

The problems addressed in this thesis can be subsumed as follows:

- As deduced in Section 2.3, there is a need for a component model that allows to express temporal and functional behavior of a wide range of analysis and design models, as well as of reality itself. Besides, such a component model should also allow for the combination of different models with each other and should support the lifting of component level properties to graph level. Lastly, there should be a theory based on such a component model that equally supports abstraction and refinement between models of different

entropies and that consequently allows to relate different levels of exclusion, inclusion and bounding abstraction or refinement to each other. However, neither such a component model nor such an abstraction-refinement theory exist.

- Following the discussion in Section 2.4 we established that analysis of the important class of cyclic concurrent hard real-time streaming applications with shared resources is limited to a small number of approaches. Many approaches have difficulties with the combination of cyclicity and shared resources, such that only few methods are applicable. But these methods are either based on highly entropic analysis models, preventing a fast analysis, or have such a low effectiveness that analysis results rarely have a sufficiently high accuracy. This imposes the need for analysis approaches that are applicable for such applications, that are reasonably fast to support short analysis and design cycles and that provide a sufficiently high analysis accuracy.
- Finally, in Section 2.5 we have argued that it is often desirable to conduct simulation as a means to provide faster evaluations than achievable with analysis, to falsify wrong analysis results, to sometimes even verify analysis techniques and to measure analysis technique effectiveness. However, a simulation framework that embeds into a component model as described in the first bullet point and that is applicable for a wide range of safety-critical systems does not exist.

2.7 CONTRIBUTIONS

In the first part of this thesis we focus on the derivation of a general abstraction-refinement theory for event-driven applications, which establishes the entropy relation between different analysis approaches:

- We introduce a component model that is:
 - Generic, i.e. it allows to represent the temporal and functional behavior of all kinds of event-driven applications.
 - Modular, i.e. it allows for composition, hierarchy and hiding.
- For that component model, we introduce an abstraction-refinement theory that:
 - Defines exclusion, inclusion and bounding abstraction and refinement.
 - Guarantees automated lifting of abstraction and refinement on composition, i.e. guarantees lifting from components to graphs of components.

- We define methods to reduce entropy on abstraction by the removal of data or resource dependencies in a uniform way.
- We discuss the representation of the important subclass of dataflow models in the component model.

In the second part of the thesis we address the derivation of analysis approaches suitable for cyclic concurrent hard real-time streaming applications with shared resources and SPP scheduling. For such applications we improve state-of-the-art, which is a combination of dataflow and classical real-time analysis, as follows:

- We illustrate that the methods of the first part can be used to derive strong relations between applications and several analysis models. Given these relations, we prove conservativeness, i.e. we prove that our analysis results are not only valid within the analysis models, but are also valid bounds on the actual events within the respective real, implemented applications.
- We describe the interference-reducing effect of data cycles and how it can be used to increase effectiveness and thereby accuracy.
- We introduce a technique called iterative buffer sizing. This technique allows to exploit aforementioned effect even more by not only analyzing, but actually optimizing the considered applications, resulting in both better temporal properties and lower memory requirements. Additionally, we introduce so-called synchronization edges that can be perceived as non-functional buffers solely used to optimize applications even more.
- We replace the rather crude period-and-jitter interference characterization by so-called execution intervals, again increasing effectiveness and analysis accuracy.
- We present an analysis approach that is capable of considering task phases with varying execution times and varying rates, which enables the following improvements:
 - Possibility to use a more fine-grained task model, resulting in a higher accuracy.
 - Removal of the strict condition that synchronization must occur atomically at task execution boundaries, relaxing hardware and operating system requirements.
 - Capability to express multi-rate applications, increasing the scope of these analyses to a broader class of applications.
 - Consideration of multiple shared resources in a uniform way, reducing the necessity to abstract away certain resources, which effectively further increases accuracy.

- Lastly, we present a simulation framework that allows the simulation of a large subclass of models expressed in the component model. This allows for a fast evaluation of safety-critical systems and gives support for verification, falsification and effectiveness measurement of analysis approaches.
- We use the decoding mode of a WLAN 802.11p transceiver application as an ongoing case study, which allows to study and relate the practical benefits and weaknesses of the different approaches to each other.

2.8 THESIS OUTLINE

The outline of the remainder of this thesis is as follows.

Part I of this thesis has the following structure: Chapter 3 gives a more detailed introduction to the generic component model and the related abstraction-refinement theory, provides an informal description of the theory and discusses related work. In Chapter 4 the generic component model is introduced formally and in Chapter 5 the accompanying theory applicable for exclusion, inclusion and bounding abstraction and refinement. Chapter 6 presents several case studies for the theory and Chapter 7 concludes the first part.

The outline of Part II of this thesis is as follows: Chapter 8 introduces the different analysis techniques and simulation in more detail and presents related work. Chapter 9 describes the representation of concurrent real-time systems in the timed component model introduced in the first part, as well as the derivation of analysis models by means of abstraction. In Chapter 10 analysis techniques based on a combination of period-and-jitter and cyclic data dependencies are discussed, both with and without the interference limitation using cycles, and in Chapter 11 the techniques using execution intervals. Chapter 12 presents different analysis techniques that allow for an accurate analysis of multi-phase applications and ultimately also multiple shared resources. Chapter 13 discusses simulation and Chapter 14 concludes the second part.

Chapter 15 finally wraps up this thesis.

PART I

An Abstraction-Refinement Theory

chapter 3

Introduction

We have already seen on several examples that the ever-increasing complexity of real-time systems imposes a need for both component-based and model-based reasonings. Component-based reasonings allow to break down larger problems into easier manageable subproblems, whereas model-based reasonings allow to reduce complexity by working with models that are less entropic than reality. Deriving a timed component model that allows for such component-based reasonings, as well as an abstraction-refinement theory that enables the construction of models with varying entropies is consequently subject of the following first part of this thesis.

In Chapter 4 we introduce a timed component model that is laid out in such way that it can serve as a meta-model for many existing analysis models relevant in the context of real-time systems, such as timed dataflow, timed petri nets or Kahn Process Networks (KPNs). Our timed component model is denotational and asynchronous, which allows for a concise expression of both synchronous and asynchronous applications. Its components relate discrete-event streams between input and output interfaces using mathematical relations. The streams transferred between components consist of indexed pairs of timestamps and values, which allows for the expression of both temporal and functional behaviors.

To maximize expressibility, our timed component models does not explicitly forbid any kind of temporal or functional component (inter)dependence. However, complex dependencies between components prevent to consider components in isolation, effectively disabling component-based reasonings. For instance, reconsider our appointment example from the preface: There we established that the “*taking the train*” and the “*walking from arriving train station to work*” components were not separable, as the outcome of the “*taking the train*” part influenced the behavior of the walking part (on a late train arrival you would have considered taking a taxicab instead of walking). Consequently, the components could not be analyzed in isolation.

We address this issue from two different angles: First, we derive several conditions under which components can be treated in separation, not generally, but with respect to certain properties such as time or input acceptance. And second, we introduce several abstraction-refinement theories that allow to abstract dependencies between components away, such that an abstract component graph becomes analyzable on a per-component basis. For our appointment example, this could e.g. mean that we make an abstraction of the “*walking from arriving train station to work*”, such that you would always walk to work, and that at the same speed, independent of when the train arrived. Such an abstraction would allow to analyze temporal behavior per component.

In Chapter 5 we introduce three different abstraction-refinement theories, namely the aforementioned exclusion, inclusion and bounding abstractions and refinement theories. All these theories have in common that they allow to reduce or increase entropy of a model, depending on whether abstraction or refinement is applied. Moreover, the theories enable an automatic lifting of abstraction or refinement from component to graph level, without any additional restrictions, besides that graphs must be constructed according to the rules of our timed component model. The latter property is critical, as without it, it would be required to do a holistic analysis of a model or of reality just to enable abstraction, defeating the very purpose of abstraction.

As assessed before, only bounding abstraction can be used to make deterministic abstractions of non-deterministic models. However, this comes with the downside that either the abstract model or the model for which the abstraction is derived must be monotone with respect to the bounding relation. This means that if a model is not monotone with respect to this relation, every bounding abstraction inevitably comes with a loss of accuracy. In contrast, inclusion and exclusion abstractions do not bound behaviors, but rather contain behaviors of the model for which the abstractions are derived. Consequently, they are not necessarily prone to accuracy loss.

Chapter 6 discusses several applications of our theory and bridges the gap to the second part of this thesis.

Finally, note that the presented component model and abstraction-refinement theory revolve around real-time systems (or more generally, around computer systems), i.e. applications in which time is discrete. However, we can also extend our theory to an even broader context. For that matter we introduce in Appendix A an extension of our theory that makes it applicable for Cyber-Physical Systems (CPS's) as well, i.e. applications that integrally combine discrete-event and continuous time systems. To this end we replace our notion of streams by a notion of signals and generalize the bounding abstraction-refinement theory such that it does not only allow for bounding with respect to timestamps or values, but bounding with respect to any relation between signals.

As to the remainder of this introductory chapter, in Section 3.1 we give a more in-depth, yet informal description of our theory and highlight several of its key aspects. Section 3.2 discusses related work and Section 3.3 highlights differences to the paper on which the content of the first part of this thesis is based on.

3.1 INFORMAL DESCRIPTION & DISCUSSION

In this section we give an informal description of our theory, which serves the purpose of a basic idea and defines the terminology used in the remainder of this thesis. Moreover, we discuss several key aspects of our theory and establish the relation to the The-Earlier-the-Better (TETB) refinement theory that laid the foundation for this work.

3.1.1 TIMED COMPONENT MODEL

In our theory we reason in **streams** that map **indices** to tuples of **timestamps** and **values**. Technically, every stream has an infinite amount of events, but the smallest index from which onwards all timestamps and values are equal to infinity is used to mark the **length of a stream**. Streams are transferred over **ports**, with each port being characterized by a **value domain** that specifies which values the indices of a transferred stream can take, as well as an **ordering relation** that specifies an order on the value domain. Multiple ports form an **interface** and streams being transferred over the ports of an interface form a **trace**.

A **component** consists of an **input interface**, an **output interface** and a **relation** that translates traces on the input interface to traces on the output interface. As the relation of a component does not necessarily have to be a function, a component can also be **non-deterministic**, i.e. it can produce different output traces for the same input trace. We require that all components accept the **empty trace**, a trace consisting of zero length streams (all timestamps and values are infinite), and that the empty trace on the input interface always results in the empty trace on the output interface.

In the following we call pairs of input and output traces that are valid with respect to a component relation **behaviors** of the component.

The **input set** of a component is defined by its relation and contains all traces that are accepted by the component. Likewise, the **output set** of a component is also defined by its relation and contains all traces that can be produced.

Components can be **composed** to form new components. We differ between **parallel compositions** in which the input and output interfaces of the original components are united, **serial compositions** in which (a part of) the output interface of one component is connected to (a part of) the input interface of another, and **feedback compositions** in which (a part of) the output interface is connected to (a part of) the input interface of the same component. Interface connections on serial and feedback compositions adhere to one-to-one **port mappings**.

Composed components have input and output interfaces containing all ports not connected in the respective compositions. Thus we differ between **internal interfaces** whose ports are connected and **external interfaces** that remain

unconnected. Consequently, a parallel composition does not have internal interfaces, whereas the external interfaces of serial and feedback compositions contain less ports than the individual components.

While the relations of parallelly composed components remain unchanged (as consequently also the input and output sets), the relations of serially composed components are reduced to the combinations of output traces that can be produced by the first and the input traces that can be accepted by the second component. This implies that any serial composition of two components is valid, however, both the input set and output set of the composition can contain less traces that can be accepted or produced, respectively. For feedback composition, an external input trace is valid if for this input a fixed point is always reachable, starting from the empty trace on the internal interface.

A component that consists of multiple composed components is called a **component graph**.

3.1.2 BOUNDING ABSTRACTION & REFINEMENT

Our bounding abstraction-refinement theory relies on two key concepts: **Bounding** and **input acceptance preservation**, which we consider separately in our theory. In this section we focus on the bounding part. We begin with the bounding of streams, then traces, components and lastly component graphs. Finally, we discuss the relation of bounding to abstraction and refinement.

We say that a stream upper-bounds another stream if for all indices the timestamps are equal or larger and if for all indices the values are equal or larger according to the port-specific ordering relation. Likewise, a trace upper-bounds another trace if all the streams of the former upper-bound all streams of the latter. A lower bound is the reverse of an upper bound.

Based on trace bounding we define two different variants of component bounding. Let it hold for two input traces of two components that the input trace of the first is an upper bound on the second. And let it further hold that there exists a corresponding output trace of the first that is an upper bound on all corresponding output traces of the second. If this holds for any input traces of the two components that are upper-bounding each other we say that the first component is a **worst-case upper bound** on the second (and the second a worst-case lower bound on the first).

Analogously, let it hold for an input trace of one component that is a lower bound on an input trace of another that for these input traces there exists an output trace of the first component that is a lower bound on all output traces of the second. If this holds for any input traces of the two components that are lower-bounding each other we say that the first component is a **best-case lower bound** on the second (and the second a best-case upper bound on the first).

We prove that both worst-case and best-case bounding are preserved on component composition, without any further requirements. This allows us to conclude

that two graphs bound each other if all their components bound each other, i.e. bounding is automatically lifted from component to graph level.

Finally we define that a component graph is a **worst-case (best-case) abstraction** of another graph if the first accepts at least all inputs of the second and if for the same input the first has at least one output that upper-bounds (lower-bounds) all outputs of the second. Likewise, a graph is a **worst-case (best-case) refinement** of another graph if the first accepts at least all inputs of the second and if for the same input the second has at least one output that upper-bounds (lower-bounds) all outputs of the first. We show that bounding abstraction and refinement can be concluded from component level bounding and graph level input acceptance preservation. We prove that bounding abstraction and refinement are transitive.

3.1.3 RELATION BETWEEN BOUNDING & TETB REFINEMENT

Our bounding abstraction-refinement theory is mainly inspired by the TETB refinement theory [GTW11], but generalizes it in multiple ways. While in TETB streams are only temporal, our notion of streams is based on indices, timestamps and values. This enables a consideration of components that produce values out of timestamp order like in [HB16], as well as both temporal and functional bounding. But the key difference compared to [GTW11, HB16] lies in the combination of bounding and input acceptance preservation. In TETB, bounding and input acceptance preservation are inseparably linked via the component refinement relation $\sqsubseteq$: The relation implies that a component A^{impl} only refines a component A^{wc} if it is worst-case temporally bounded by A^{wc} , i.e. if component A^{impl} is never slower than the worst-case of A^{wc} , and if A^{impl} preserves input acceptance of A^{wc} , i.e. A^{impl} accepts at least all inputs that A^{wc} also does. In contrast, our theory separates bounding and input acceptance preservation, which resolves several shortcomings:

First, in application design it is desirable that a refined implementation accepts at least all inputs of a design model, whereas in application analysis it is desirable that an analysis model accepts at least all inputs of an implementation. As depicted on the left side of Figure 3.1, abstraction and refinement are symmetric in TETB, which implies that an abstraction cannot accept more, but only less inputs than an implementation. This means that in TETB the only way to realize analysis models which hold for all cases of an implementation is to construct them such that they accept exactly the same inputs as the respective implementation. However, if one considers that many implementations are input-restricted (such as components that are only laid out for inputs with a certain period and jitter), while many analysis models are by definition input-complete (i.e. accept any inputs, such as dataflow models), it appears that equal input acceptance is an unrealistic assumption in many cases. As depicted on the right side of Figure 3.1, in our theory we consider input acceptance preservation and bounding separately. This allows for a combination of the two as needed, i.e. refinement for application design with input acceptance preservation from model to implementation,

Figure 3.1: Input acceptance preservation and bounding.

as well as abstraction for application analysis with input acceptance preservation from implementation to model. Effectively, this means that TETB is mainly a refinement theory, while our theory is an abstraction-refinement theory.

Second, if for instance interference effects due to resource sharing or buffer allocation are in the scope of analysis, determining worst-case upper bounds on the temporal behavior of components is not sufficient. Instead, upper bounds on their jitters are needed [9, 10], which additionally requires to construct models that are best-case lower-bounding the temporal behavior of implementations, i.e. no behaviors of an implementation are allowed to be earlier than the earliest behavior of the model. As depicted in Figure 3.1, TETB with its refinement relation $\sqsubseteq$ only supports worst-case models. We introduce two bounding relations, $\trianglelefteq$ for worst-case bounding and $\trianglelefteq$ for best-case bounding, such that our theory supports both worst-case and best-case models.

And third, in TETB automatic lifting from component to graph level is only discussed with respect to refinement, which implies that input acceptance preservation must already hold on component level. To understand the impact of this requirement, consider two serially connected components, of which the latter is input-complete, but only receives strictly periodic inputs of the former. In TETB, it would be required that any refinement of the latter component were also input-complete, although it would actually suffice if the refinement of the latter accepted the strictly periodic inputs from the former. On top of that, in most cases it does not even suffice that input acceptance preservation holds on component level, but both abstract and refined components must be additionally input-complete. This requires complex workarounds for cases in which e.g. data streams do not arrive in order [HB16], while for other cases even no such workarounds exist (an example of this can be found in our case study). These implications show that input acceptance preservation on component level is a both severe and unnecessary restriction. Our theory circumvents this restriction by separately discussing lifting of bounding and input acceptance. Consequently, bounding is automatically lifted from component to graph level without any restrictions on component input acceptance.

3.1.4 INCLUSION & EXCLUSION ABSTRACTION & REFINEMENT

As discussed in the introduction, the main advantage of bounding abstraction and refinement over inclusion is that bounding can be used to remove behaviors, such that e.g. deterministic models can be created for non-deterministic implementations. The single downside of bounding is, however, that if an implementation component is not monotone with respect to the bounding relation, any valid component bounding comes with a loss of accuracy. The benefits of deterministic models for non-deterministic implementations usually outweigh a minor loss in accuracy. Nevertheless, for cases in which accuracy is the main concern we present an inclusion abstraction and refinement theory for the same component model.

In addition, we also derive an exclusion abstraction and refinement theory that uses the same building blocks as inclusion and that can be used to remove unconsidered behaviors.

Analogous to bounding abstraction and refinement, our inclusion and exclusion abstraction-refinement theories are based on the concepts of **inclusion** and **input acceptance preservation**. We begin with trace equality and based on that develop inclusion of components, as well as inclusion of component graphs. Lastly, we discuss the relation of component inclusion to inclusion and exclusion abstraction and refinement.

We define that a trace equals another trace if both are defined on the same interfaces and if it holds for all ports of these interfaces that the respective streams are equal.

Based on trace equality we define component inclusion as follows. Let it hold for a trace that it is a valid input trace of two different components. Furthermore, let it hold that for this input trace the first component can match all output traces of the second, i.e. the first component can produce at least all output traces that the second component can also produce. If this holds for any input traces of the two components that are equal we say that the first component **includes** the second (and the second is included by the first).

We prove that inclusion is preserved on component composition, without any further requirements. This allows us to conclude that two graphs include each other if all their components include each other, i.e. inclusion is automatically lifted from component to graph level.

We define that a component graph is an **inclusion abstraction** of another graph if the first accepts at least all inputs of the second and if for the same inputs the first can match all outputs of the second. Likewise, a graph is an **inclusion refinement** of another graph if the first accepts at least all inputs of the second and if for the same inputs the second can match all outputs of the first.

Analogously, we define that a component graph is an **exclusion abstraction** of another graph if the first accepts at least all inputs of the second and if for the same inputs the second can match all outputs of the first. Likewise, a graph is an **exclusion refinement** of another graph if the first accepts at least all inputs

of the second and if for the same inputs the first can match all outputs of the second.

We discuss that both inclusion and exclusion abstraction and refinement can be concluded from component level inclusion and graph level input acceptance preservation. We prove that inclusion and exclusion abstraction and refinement are transitive.

3.1.5 INPUT ACCEPTANCE LIFTING & REPLACEABILITY

Unlike the TETB refinement relation, our bounding relations do not imply input acceptance preservation. This generalization comes at the cost that graph level input acceptance preservation, which is needed for abstraction and refinement, cannot be concluded from component level bounding. To address this, we derive properties which imply automatic lifting of input acceptance from component to graph level. If such an automatic lifting is given, one can conclude input acceptance preservation between graphs from input acceptance preservation between individual components.

We say that a component is **input-independent** if it holds for the streams accepted according to the component relation that all combinations of streams are accepted, i.e. if two streams are accepted on one port and two other on another, then all four combinations must be accepted. Moreover, a component is **empty-continuous** if it holds that the relation between input and output traces is continuous with respect to a certain trace ordering relation for which the empty trace is the infimum of all traces.

It can be seen that a graph consisting of components that are both input-independent and empty-continuous is also input-independent and empty-continuous, given that the input sets of all connected components are supersets of the respective connected output sets. If these properties hold it can be further seen that input acceptance is automatically lifted from component to graph level, i.e. that a graph accepts all inputs that are accepted by its respective components before composition.

Furthermore we discuss that all **input-complete components** (components that accept any inputs) are input-independent, as well as that all **operational components** (components that produce extended outputs for extended inputs) are empty-continuous. This lets us conclude that a graph of input-complete, operational components is also input-complete and operational, which is for instance the case for the important subclass of **timed dataflow models**.

Based on the same properties, conditions can be derived for which components in a graph can be **replaced** without reducing the input acceptance of the graph.

3.1.6 PREVENTION OF DEAD STATES

Consider the two serially connected components depicted in Figure 3.2. Let us further assume that component *A* is a very simple component, such that it only

Figure 3.2: Dead states and non-determinism.

accepts three input streams, x_0 , x_1 and x_2 , and produces for these input streams the following output streams: y_0 for x_0 , y_1 for x_1 and y_2 or y'_2 for x_2 . Note that A is non-deterministic as it can produce both y_2 and y'_2 for x_2 . Further let B be of similar simplicity, such that it only accepts the streams y_0 and y_2 as inputs and that it produces the output streams z_0 and z_2 for these inputs.

If the two components are serially connected, as indicated in Figure 3.2, **dead states** can occur, that is, component A can produce outputs such as y_1 that B does not accept. To prevent such dead states, different measures can be taken.

The most conservative approach would be to disallow the composition of such components altogether. But this would severely restrict expressiveness of our component model and prevent application for many relevant use-cases. Consequently, it appears more reasonable to allow the composition of the two components and to prevent the occurrence of dead states by different means.

For instance, we can allow the composition of any two components in principle, but explicitly define that on composition inputs that can result in dead states are disallowed. For our example, this would mean that if components A and B are serially composed, the serial composition does accept x_0 as an input (as the corresponding output y_0 is accepted by B), but not x_1 (as y_1 is not accepted by B). With respect to the non-deterministic case that occurs on an input x_2 it can be further differed between two interpretations.

According to the so-called **angelic interpretation of non-determinism** it is assumed that components always behave well with respect to their surroundings. For our example, this would mean that component A always produces y_2 on an input of x_2 , as only y_2 can be accepted by B . This approach would result in maximal compatibility between components. However, a major disadvantage would be that components must internally adapt to the acceptance of other components. For instance, A would have to be aware of its surroundings, i.e. the input acceptance of B , and internally restrict its outputs to satisfy the requirements of B . While such an approach may appear suitable if considering the component model in isolation, we should keep in mind that the purpose of the component model is to represent actual, i.e. real, components, for which an internal adjustment of behaviors is not straightforward and sometimes even infeasible.

This is the reason why we adopt a **demonic interpretation of non-determinism** in our model, which assumes that components can behave, or even more drastic, always behave bad with respect to their surroundings. Applied on our

example, this means that if A receives an input of x_2 it always produces an output y'_2 that is not accepted by B . To prevent the occurrence of dead states we consequently have to disallow the input of x_2 to A altogether. While this approach comes at the cost that compatibility is reduced compared to the angelic interpretation, it also allows to compose any components with each other without the need to modify component behavior: The input of x_2 to A is simply prevented, without adapting the internal behavior of A .

In consequence, we define serial and feedback compositions such that an external input is only accepted by the respective compositions if not only one, but all outputs for that input are also accepted by the connected components that take these outputs as inputs.

One important implication of such definitions of compositions is obviously that input acceptance can be reduced on compositions. This is the reason why it is generally not sufficient to evaluate input acceptance of single components in order to determine input acceptance of an entire graph of components. As the comparison of graph level input acceptance is a prerequisite for both abstraction and refinement, we dedicate the entire Section 4.6 to this discussion.

Finally, note that even inclusion is not safe from a reduction of input acceptance: Consider that component A includes a component A' , i.e. A has more behaviors than A' , such that A' can only produce y_2 for an input of x_2 whereas A can produce both y_2 and y'_2 . If A' is composed with B then the composition accepts both x_0 and x_2 as inputs because all respective outputs are accepted by B . But as discussed above, for the case that A is composed with B it holds that only x_0 is accepted. This demonstrates that replacing a component by another that has more behaviors than the original one can result in overall less behaviors of the composition. Nevertheless, this is not problematic in our theory as also for inclusion abstraction we clearly distinguish between inclusion, which only needs to hold for accepted inputs, and input acceptance preservation, i.e. an input acceptance reduction is tolerable with respect to inclusion.

3.2 RELATED WORK

In this section we give an overview of related work, the so-called interface refinement theories. The main scope of interface refinement is the replacement of components with refined components without violating certain graph level properties. In this context, component interfaces are abstractions of the components themselves, containing sufficient information to be representative for the underlying components, but not more information than needed to assert the adherence or violation of graph level properties. With respect to the graph level properties that are to be preserved we divide existing interface refinement theories into three classes.

With **type refinement** [LS15] it is merely ensured that refined components accept at least the same data types and produce no other data types than abstract components. **Inclusion refinement** subsumes all theories that involve

inclusion of behaviors, meaning that abstract components can match at least all behaviors (all outputs given the same inputs) of refined components. Examples of such theories are **language refinement** [LS15], that is also called trace containment [Mil71], and the stronger **simulation refinement** [LS15]. Lastly, **bounding refinement** differs from inclusion refinement in that it is not required for abstract components to match the exact behaviors of refined components, but it is sufficient that abstract components upper- or lower-bound behaviors of refinements. This is of special importance for analysis as, unlike inclusion refinement, bounding refinement allows to create deterministic and monotone abstractions of non-deterministic, non-monotone implementations, simplifying analysis drastically as for each input only one behavior remains to be analyzed.

Note that in state machine terms, simulation refinement can be used to determine state machines that have less states than others, but otherwise adhere to the same transitions (the same behaviors). In contrast, bounding refinement can be used to not only remove states, but also to remove transitions between states, which represents an entirely different dimension of complexity reduction.

Besides this classification we further differ between theories considering the **temporal** and **functional** behavior of components and between theories for **synchronous** and **asynchronous** components. Synchronous components allow for an implicit notion of time by assuming a certain duration of synchronous rounds, while asynchronous components are more general, but require an explicit notion of time to consider temporal behavior.

In [dAH01b] a functional language refinement relation is introduced for synchronous components. Interfaces of components are defined in a relational manner using state machines, allowing to capture input-output dependencies of components. The theory defines refinement only for a subclass of relational interfaces. Moreover, the theory does not contain proofs for the preservation of refinement on composition. These shortcomings are amended in [TLHL09] that allows for any kind of relational interfaces (with the restriction that feedback composition is not allowed to be combinatorial) and that proves the automatic lifting of refinement from component to graph level. In contrast to these works, our theory assumes an asynchronous component model, resulting in a higher expressibility. Instead of the implicit notion of time that is inherent to synchronous theories we make use of an explicit notion in the form of timestamps.

A functional language refinement relation for concurrent asynchronous components is presented in [dAH01a]. The interfaces are described using automata in an operational manner. This theory of interface automata is extended in [dAHS02] for timed automata [AD94], allowing an explicit specification of progress of time. [dAHS02] lacks a definition of refinement, which is added in [D⁺10].

In comparison to the aforementioned theories, our theory is asynchronous, but also denotational and it operates on entire streams instead of single values, which allows for a concise representation of complex interface relations. Besides, our theory makes use of a relational component model and as such entirely abstracts

Related inclusion refinements:

Inclusion refinement in our theory:

The-earlier-the-better refinement $\sqsubseteq$:

Worst-case refinement in our theory:

Figure 3.3: A graph G and its refinement graph G' .

from the notion of states (albeit state machines can be seen as one means to define the relations of components).

We define three types of refinement, namely inclusion, exclusion and bounding refinement. The similarities and differences of our theory to related works are illustrated in Figure 3.3, in which a component graph G is refined to a graph G' .

Our inclusion refinement theory is very similar to the aforementioned theories in that our inclusion theory also requires matching behaviors, i.e. inclusion only holds if all outputs of the components A' and B' can be matched by the respective outputs of components A and B , given the same inputs. However, the related inclusion theories additionally demand input acceptance preservation on component level, such that the refined components A' and B' accept at least all inputs of the components A and B (the simulation refinement as defined in [LS15] even requires equality of both input and output sets). In contrast, our inclusion refinement theory relaxes this requirement by separating inclusion from input acceptance preservation. Hence input acceptance preservation only needs to hold on graph level, i.e. only between G and G' in the figure.

Furthermore, note that both language refinement and simulation refinement are defined via state machines whose transitions process single input to single output values. For refinement both require that one component can match all sequences of input-output values of another. The difference between the two is that simulation refinement additionally requires that one component can always match the behaviors of the other without knowledge of future inputs, i.e. the component cannot make non-deterministic choices leading to states from which the matching cannot be resumed. Our inclusion refinement requires a matching of entire streams of values and thereby abstracts from single values and state machines altogether. As such, the stronger requirement of simulation refinement also cannot be expressed in our theory, in that respect it corresponds to language refinement instead.

The requirement of matching behaviors inherent to inclusion refinement theories further implies that if G' is non-deterministic also G must be non-deterministic. For TETB refinement as well as our bounding refinement theory, the latter is not required, but it is only needed that the behaviors of A and B upper-bound the behaviors of A' and B' instead. Consequently, G is allowed to be both

Figure 3.4: Comparison of our theory to abstract interpretation (f_γ, f_α : any functions, $R_\triangleright, R_\triangleleft$: bounding relations, R_γ, R_α : any relations, $f_=$: equality function).

deterministic and monotone even if G' is neither. This is of special importance for analysis purposes as deterministic and monotone analysis models enable the application of efficient analysis techniques [SB09a], as well as the usage of algebraic techniques on closed form expressions, which allow for deep insights into the respective problems (e.g. enabling a quick identification of bottlenecks).

According to our classification, the abstract interpretation theory [CC77] is a functional bounding refinement theory. The key differences between abstract interpretation and our inclusion and bounding theories are as follows (see Figure 3.4): First, abstract interpretation allows for variable mappings between different domains, whereas our bounding abstraction requires a particular relation, the bounding relation, to hold between streams in the same domain. In Appendix A we extend our bounding theory to an approximation theory that allows to use any kinds of relations, making the approximation theory a generalization of abstract interpretation. Inclusion requires equality of streams within the same domain and since equality can be expressed as a function, it is a subset of abstract interpretation in that respect. Second, abstract interpretation is limited to mappings between input and output values, whereas both bounding and inclusion allow for any kinds of relations between input and output streams, hence allowing for non-determinism. Third, our theory supports a component-based reasoning (e.g. by an automatic lifting of value bounding from component to graph level), which abstract interpretation does not. Consequently, abstract interpretation also does not support a one-by-one replacement of components that is especially useful for refinement-based design processes. And fourth, our approach allows to combine temporal and functional (value) bounding, whereas abstract interpretation is limited to functional bounding.

A bounding refinement relation is introduced in [TWS06] for modular real-time systems. The theory is asynchronous and temporal as it makes use of arrival and service curves to characterize traffic, as well as provided and remaining service. This allows to express both data and resource dependencies in a single model. However, the events that form the curves are specified in the time interval instead of the time domain, which prevents a correlation of events from different

curves. In consequence, this disallows cycles, i.e. the application of the theory for components with feedback. Moreover, the service curves only denote time, but no values, which renders an application for use-cases in which reordering can occur impossible. In contrast, both feedback composition and reordering can be expressed and handled with our theory.

In [WBS09] the concept of creating deterministic abstractions that upper-bound the temporal behavior of a non-deterministic implementation is introduced using deterministic timed dataflow models. But the work lacks the definition of a transitive refinement relation that can be used to create multiple refinement layers. This is amended with the temporal bounding refinement relation presented in [GTW11], the aforementioned TETB refinement theory.

In all aforementioned theories, refinement and abstraction are **symmetric**, i.e. if a component refines another, the latter is an abstraction of the former. As in these theories a refinement must accept at least the inputs of an abstraction, the symmetry of abstraction and refinement implies that an analysis model may well be a valid abstraction of an implementation although it only accepts a small part of the implementation's inputs. But usually just the opposite is desirable, i.e. that an analysis model accepts at least all inputs of an implementation. To account for this requirement we define our notions of abstraction and refinement **asymmetric**, such that a refinement must accept at least the inputs of a model and an abstraction at least the inputs of an implementation. This makes our theory equally suitable for both application design and analysis.

Refinement in TETB further implies both worst-case lower-bounding and input acceptance preservation. This allows for the creation of non-deterministic refinements of deterministic worst-case models, which is fundamentally different to both language and simulation refinement and similar to the worst-case refinement in our theory. As illustrated in Figure 3.3, input acceptance preservation does not only need to hold between two refining graphs as in our theory, but between all components of the graphs individually. A consequence of this limitation is that for preservation of refinement on serial composition the components must be in most cases input-complete and on feedback composition even always input-complete, which renders the whole notion of input-restricted components in many cases useless. We only require bounding on component level, which removes any requirement on input acceptance preservation. TETB further requires refinement monotonicity of the individual components on feedback and serial compositions, as well as refinement continuity on feedback composition, which we do not (a short discussion on these differences can be found after the respective proofs in Chapter 5.1.5). Lastly, we introduce the notion of value refinement, making our refinement theory both temporal and functional, and we define both worst-case and a best-case bounding relations, enabling the creation of both worst-case and best-case models.

3.3 RELATION TO PAPER VERSIONS

The remainder of the first part of this thesis is based on [4], which is itself an extension of [3]. The main difference to the paper versions is that we additionally introduce support for the removal and addition of ports to the process of abstraction and refinement, which is required for the approaches presented in the second part of this thesis. Note that this modification further complicates the abstraction-refinement theory. If the removal of ports is not in your scope of interest, kindly be advised to refer to [4] for a more approachable version of the theory.

With respect to notation, we use Ξ instead of Θ to denote the mapping of interfaces in serial and feedback component compositions, in order to prevent confusion with the number of dataflow actor phases Θ .

On top of that, we also formalize exclusion abstraction and refinement, which is not discussed in the paper versions and extend our real-time system theory to a theory applicable for CPS's in Appendix A.

Timed Component Model

In this chapter we first give a formal definition of streams on ports, which is subsequently generalized to traces on interfaces consisting of multiple ports. Using such interfaces we define components relating traces on input interfaces to traces on output interfaces. For such components, we introduce appropriate definitions of monotonicity and continuity. Finally we define parallel, serial and feedback compositions of components, which enable the construction of component graphs.

4.1 PORTS & STREAMS

We define streams as infinite sequences of events, with each event mapping an index to a timestamp and a value. Subsequently we define the length of streams, as well as so-called ports that are used to transfer streams from a specific value domain. Finally we specify the connectivity of ports and define the prefix and earlier-than relations for streams on the same ports.

Definition 1 (Stream). *A stream x on a port p is an infinite sequence of indexed events, with each event consisting of the production time of the event in the form of a timestamp and a value from the value domain of the port. Formally x can be described as a total mapping $x : \mathbb{N} \rightarrow \mathcal{T} \times O$, with $\mathcal{T}$ a continuous time domain and O a value domain. We require that the time domain $\mathcal{T}$ is a lattice with respect to an ordering relation $\leq$ and that $\mathcal{T}$ has an infimum $0 \in \mathcal{T}$, as well as a supremum $\infty \in \mathcal{T}$, such that $\forall_{\tau \in \mathcal{T}} : 0 \leq \tau \leq \infty$. Analogously we require that the value domain O is a lattice with respect to an ordering relation $\models$ and that O has an infimum $\vartheta^0 \in O$, as well as a supremum $\vartheta^\infty \in O$, such that $\forall_{\vartheta \in O} : \vartheta^0 \models \vartheta \models \vartheta^\infty$. We use $\tau_x : \mathbb{N} \rightarrow \mathcal{T}$ and $\vartheta_x : \mathbb{N} \rightarrow O$ to retrieve timestamps and values of events by their indices, respectively.*

Although streams are formally defined as infinite sequences of events, streams can also be seen as finite. We define the length of streams as follows:

Definition 2 (Stream Length). We define an event of a stream x at index i to be absent iff $\tau_x(i) = \infty$ and $\vartheta_x(i) = \vartheta^\infty$. The length of a stream can then be defined as the smallest index from which onwards all events are absent, i.e. (with $\min \emptyset \equiv \infty$):

$$|x| = \min\{i \in \mathbb{N} \mid \forall i' \geq i: \tau_x(i') = \infty \wedge \vartheta_x(i') = \vartheta^\infty\}$$

In our timed component model we transfer streams over so-called ports, which are specified as follows:

Definition 3 (Port). A port p is characterized by a tuple $(x, O_p, \models_p)$ and contains a stream $x \in St(p)$, with $St(p)$ the set of all valid streams on port p . The set $St(p)$ is constructed based on a port-specific value domain O_p , such that $St(p) = \{x \mid x : \mathbb{N} \rightarrow \mathcal{T} \times O_p\}$. The port-specific value domain O_p must adhere to the requirements on the value domains of streams, i.e. it must be a lattice with respect to an ordering relation $\models_p$, with ϑ_p^0 the infimum and ϑ_p^∞ the supremum. Based on the ordering relations $\leq$ of $\mathcal{T}$ and $\models_p$ of O_p we further define the null stream $x_p^0 \in St(p)$, such that $\forall i \in \mathbb{N}: \tau_{x_p^0}(i) = 0 \wedge \vartheta_{x_p^0}(i) = \vartheta_p^0$, and the empty stream $x_p^\infty \in St(p)$, such that $\forall i \in \mathbb{N}: \tau_{x_p^\infty}(i) = \infty \wedge \vartheta_{x_p^\infty}(i) = \vartheta_p^\infty$, respectively.

In the following we construct interfaces from multiple ports and connect such interfaces to other interfaces by connecting the underlying ports. However, not all ports can be connected as they may have different value domains. Consequently we define a sufficient requirement on the connectibility of ports as follows:

Definition 4 (Port Connectibility). Let q and p be two different ports, i.e. $q \neq p$. Then port p is connectible to a port q , i.e. $q \rightarrow p$, iff it holds that all valid streams on port q are also valid streams on port p , i.e. $St(q) \subseteq St(p)$.

We define prefix and earlier-than ordering relations for streams on the same port as follows:

Definition 5 (Stream Order). Let $x, x' \in St(p)$ be two streams on a port p . The prefix ordering relation $\leq$ and the smaller-than ordering relation $\leq$ for streams are defined as:

$$\begin{aligned} x \leq x' &\equiv |x| \leq |x'| \wedge \forall i < |x| : \tau_x(i) = \tau_{x'}(i) \wedge \vartheta_x(i) = \vartheta_{x'}(i) \\ x \leq x' &\equiv |x| = |x'| \wedge \forall i < |x| : \tau_x(i) \leq \tau_{x'}(i) \wedge \vartheta_x(i) \models_p \vartheta_{x'}(i) \end{aligned}$$

It can be seen that both the prefix and the smaller-than ordering relations of streams have the same properties as in [GTW11], which implies that we can reuse the results from [GTW11] that are based on these properties.

In the following we need to compute limits of sequences of streams. For that matter we require a distance function for streams that converges towards zero if the distances between all timestamps and values approach zero.

First, let us define a suitable distance functions for timestamps. The most straightforward distance function between two timestamps τ and τ' would be to simply compute the absolute value of the difference, i.e. $d_{\mathcal{T}}(\tau, \tau') = |\tau - \tau'|$. However, this function has the disadvantage that if one of the timestamps is infinity (which is allowed as explicitly $\infty \in \mathcal{T}$) and the other timestamp approaches infinity, the distance never converges to zero, as long as the second timestamp only approaches, but does not become infinity. For that reason we make use of the following distance function that does not have this disadvantage:

Definition 6 (Timestamp Distance). *The distance between two timestamps $\tau, \tau' \in \mathcal{T}$ is a function $d_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \rightarrow \mathbb{R}_0^+$ with:*

$$d_{\mathcal{T}}(\tau, \tau') = \left| 2^{-\tau} - 2^{-\tau'} \right|$$

As timestamps can take values between 0 and ∞ this function gives distances between 0 and 1, which is maximal if one of the timestamps is infinity and the other 0.

For values we cannot define a distance function directly, as the value domain is port-specific and can be defined as needed. Nevertheless, we can define the requirements that a value distance function must adhere to as follows:

Definition 7 (Value Distance). *Let O be a lattice with respect to an ordering relation $\models$, such that O has an infimum $\vartheta^0 \in O$, as well as a supremum $\vartheta^\infty \in O$ with $\forall \vartheta \in O : \vartheta^0 \models \vartheta \models \vartheta^\infty$. Furthermore let $n : O \rightarrow \mathbb{R}_0^+$ be a partial mapping such that $\forall \vartheta, \vartheta' \in O : \vartheta \models \vartheta' \Rightarrow n(\vartheta) \leq n(\vartheta') \wedge \vartheta = \vartheta' \Rightarrow n(\vartheta) = n(\vartheta')$, $n(\vartheta^0) = 0$ and $n(\vartheta^\infty) = \infty$. Then the distance between two values ϑ, ϑ' can be defined as a function $d_O : O \times O \rightarrow \mathbb{R}_0^+$ with:*

$$d_O(\vartheta, \vartheta') = \left| 2^{-n(\vartheta)} - 2^{-n(\vartheta')} \right|$$

A distance function for values as specified above has the same properties as the distance function for timestamps. This allows us to combine the timestamp and value distances to a distance between events of streams as follows:

Definition 8 (Event Distance). *Let $x, x' \in St(p)$ be two streams on a port p . The distance between two events of such streams is a function $d_{St(p)} : St(p) \times St(p) \times \mathbb{N} \rightarrow \mathbb{R}_0^+$ with:*

$$d_{St(p)}(x, x', i) = \max(d_{\mathcal{T}}(\tau_x(i), \tau_{x'}(i)), d_O(\vartheta_x(i), \vartheta_{x'}(i)))$$

With the event distance we can define a suitable distance function $D_{St(p)}$ for streams on a port p adhering to the property that if two streams converge to each other, such that all timestamps and values converge to each other, the distance function approaches zero. Formally, this means that for two streams $x, x' \in St(p)$ on a port p the following property must hold:

$$\forall i : d_{St(p)}(x, x', i) \rightarrow 0 \Rightarrow D_{St(p)}(x, x') \rightarrow 0$$

The Cantor metric [Liu05] does not adhere to this property as it requires equality of timestamps and values for convergence. In contrast, it can be seen that the following distance function satisfies the requirement:

Definition 9 (Stream Distance). *The distance between two streams $x, x' \in St(p)$ on a port p is a function $D_{St(p)} : St(p) \times St(p) \rightarrow \mathbb{R}_0^+$ with:*

$$D_{St(p)}(x, x') = \max_{i \in \mathbb{N}} \left(\frac{1}{i+1} \cdot (1 - 2^{-d_{St(p)}(x, x', i)}) \right)$$

The first factor $(1+i)^{-1}$ ensures that $d_{St(p)}(x, x')$ converges to 0 if a stream of a finite length converges to a stream of an infinite length, whereas the second factor $1 - 2^{-d_{St(p)}(x, x', i)}$ ensures convergence per timestamp and value. The maximum finally ensures that the distance only approaches 0 if convergence of timestamps and values is given for all indices.

Using this stream distance function we can finally define the limit of stream sequences as follows:

Definition 10 (Stream Sequence Limit). *Let $x_k \in St(p)$ be a sequence of streams and $x \in St(p)$ a stream on a port p . The sequence x_k converges to x for $k \rightarrow \infty$, i.e. $\lim_{k \rightarrow \infty} x_k = x$, iff it holds that for all $\varepsilon > 0$ there exists a $K \in \mathbb{N}$ such that for all $k \geq K$ it holds that $D_{St(p)}(x_k, x) < \varepsilon$.*

4.2 STREAM TRACES & INTERFACES

We generalize the concept of streams on ports to traces on interfaces, with interfaces being sets of ports and traces being sets of streams on the ports of interfaces. Subsequently we define connectibility and connection of interfaces and lift the prefix and earlier-than ordering relations of streams to traces.

Definition 11 (Interface). *An interface P is a set of $|P|$ different ports.*

Definition 12 (Trace). *A trace X is a set of $|X| = |P|$ streams on an interface P , such that each stream $x \in X$ is on a different port $p \in P$. In the following we use the shorthand notation $X[p]$ to retrieve the stream on port $p \in P$. The set of all valid traces on an interface P is then defined as $Tr(P) = \{X \mid |X| = |P| \wedge \forall_{p \in P}: X[p] \in St(p)\}$, with $X_p^0 \in Tr(P)$ the null trace and $X_p^\infty \in Tr(P)$ the empty trace, such that $\forall_{p \in P}: X_p^0[p] = x_p^0 \wedge X_p^\infty[p] = x_p^\infty$.*

These definitions allow us to lift connectibility of ports to connectibility of interfaces:

Definition 13 (Interface Connectibility). *Let Q and P be two disjoint interfaces of the same numbers of ports, i.e. $Q \cap P = \emptyset$ and $|Q| = |P|$. Furthermore let $\Xi : Q \rightarrow P$ be a bijective mapping, i.e. $\forall_{q \in Q} \exists_{p \in P}: p = \Xi(q)$ and $\forall_{q, q' \in Q}: q \neq q' \Rightarrow \Xi(q) \neq \Xi(q')$. Then interface P is connectible to interface Q given mapping Ξ , i.e. $Q \rightarrow_\Xi P$, iff it holds that all mapped ports are connectible, i.e. $\forall_{q \in Q}: q \rightarrow \Xi(q)$.*

Given connectivity of interfaces we can also define the connection of interfaces as follows:

Definition 14 (Interface Connection). *Let Q and P be two disjoint interfaces of the same numbers of ports. Furthermore let $\Xi : Q \rightarrow P$ be a bijective mapping. If P is connectible to Q given mapping Ξ , i.e. $Q \rightarrow_{\Xi} P$, then the interfaces Q and P can be connected by an interface connection C^{Ξ} .*

An interface connection C^{Ξ} is characterized by a tuple (Q, P, Ξ) and assigns the streams on P to streams on Q , according to mapping Ξ . Formally, C^{Ξ} can be described as a function $C^{\Xi} : Tr(Q) \rightarrow Tr(P)$ with $X = C^{\Xi}(Y) \equiv \forall_{q \in Q} : X[\Xi(q)] = Y[q]$.

We can lift the prefix and earlier-than ordering relations of streams to prefix and earlier-than ordering relations of traces as follows:

Definition 15 (Trace Order). *Let $X, X' \in Tr(P)$ be two traces on an interface P . The prefix ordering relation $\leq$ and earlier-than ordering relation $\leq$ for traces are defined as:*

$$\begin{aligned} X \leq X' &\equiv \forall_{p \in P} : X[p] \leq X'[p] \\ X \leq X' &\equiv \forall_{p \in P} : X[p] \leq X'[p] \end{aligned}$$

The limit of stream sequences can be lifted to a limit of trace sequences as follows:

Definition 16 (Trace Sequence Limit). *Let $X_k \in Tr(P)$ be a sequence of traces and $X \in Tr(P)$ a trace on an interface P . The sequence X_k converges to X for $k \rightarrow \infty$, i.e. $\lim_{k \rightarrow \infty} X_k = X$, iff all streams of the sequence X_k converge to the respective streams of X for $k \rightarrow \infty$, i.e. iff it holds that $\forall_{p \in P} : \lim_{k \rightarrow \infty} X_k[p] = X[p]$*

Lastly, we formally define the merging and splitting of traces as follows:

Definition 17 (Trace Merging & Splitting). *Let X_0 be a trace on an interface P_0 and X_1 a trace on an interface P_1 . If $P_0 \cap P_1 = \emptyset$ then these traces can be merged to a trace $X = X_0 \cup X_1$ on an interface $P = P_0 \cup P_1$. In the following we call the reverse of trace merging trace splitting and traces such as $X_0 \subseteq X$ and $X_1 \subseteq X$ subtraces of a trace X .*

4.3 COMPONENTS

Instead of the term actor that is used in [GTW11] we use the term component to prevent confusion with actors from the timed dataflow theory, of which Synchronous Dataflow (SDF) actors are an example. A component is defined as follows:

Definition 18 (Component). A component A is characterized by a tuple (P_A, Q_A, R_A) and assigns the traces on output interface Q_A to traces Y that are derived according to relation $R_A \subseteq \text{Tr}(P_A) \times \text{Tr}(Q_A)$ from traces X on input interface P_A . We use XAY to denote $(X, Y) \in R_A$, with $X \in \text{in}_A$ and $Y \in \text{out}_A$. The input and output sets of A are defined as:

$$\begin{aligned} \text{in}_A &= \{X \in \text{Tr}(P_A) \mid \exists Y \in \text{Tr}(Q_A): XAY\} \\ \text{out}_A &= \{Y \in \text{Tr}(Q_A) \mid \exists X \in \text{Tr}(P_A): XAY\} \end{aligned}$$

We require that the empty input trace $X^\infty \in \text{Tr}(P_A)$ is a valid trace with respect to R_A , i.e. $X^\infty \in \text{in}_A$, and that for the empty input trace the relation R_A always results in the empty output trace $Y^\infty \in \text{Tr}(Q_A)$, i.e. $X^\infty AY \Rightarrow Y = Y^\infty$. Initially (before a trace X with $|X| > 0$ is assigned to P_A) the input interface P_A contains the empty input trace X^∞ , which implies that initially the respective output interface Q_A contains the empty output trace Y^∞ .

4.4 COMPONENT MONOTONICITY & CONTINUITY

We define component monotonicity with respect to any trace ordering relation as follows:

Definition 19 (Monotonicity). Let A be a component and α a relation on traces. Component A is α -monotone iff it holds $\forall X, X' \in \text{in}_A$:

$$(X \alpha X' \Rightarrow \forall XAY, X'AY': Y \alpha Y')$$

Moreover we call component A best-case α -monotone iff it holds $\forall X, X' \in \text{in}_A$:

$$(X \alpha X' \Rightarrow \forall X'AY' \exists XAY: Y \alpha Y')$$

Analogously we call component A worst-case α -monotone iff it holds $\forall X, X' \in \text{in}_A$:

$$(X \succsim X' \Rightarrow \forall X'AY' \exists XAY: Y \succsim Y')$$

Note that a α -monotone component is both best-case and worst-case α -monotone.

Furthermore, we define continuity of components as follows:

Definition 20 (Continuity). Let A be a component with input interface P and output interface Q and α a relation on traces. Moreover let $\sup_\alpha$ be the supremum and $\inf_\alpha$ be the infimum of a set of traces with respect to α and let for any sub-relation $R'_A \subseteq R_A$ the respective sub-input and sub-output sets:

$$\begin{aligned} \text{in}'_A &= \{X \in \text{Tr}(P) \mid \exists Y \in \text{Tr}(Q): (X, Y) \in R'_A\} \\ \text{out}'_A &= \{Y \in \text{Tr}(Q) \mid \exists X \in \text{Tr}(P): (X, Y) \in R'_A\} \end{aligned}$$

Then component A is α -continuous iff it holds that:

$$\forall R'_A \subseteq R_A: (\sup_\alpha(\text{in}'_A), \sup_\alpha(\text{out}'_A)) \in R_A$$

Monotonicity and continuity are closely related, as can be deduced from the following corollary:

Corollary 4.1 (Monotonicity vs. Continuity). *If a component A is ∞ -continuous then component A is also ∞ -monotone.*

4.5 COMPONENT GRAPHS

Composing components by connecting interfaces yields new components. In the following we define the components resulting from parallel, serial and feedback compositions, as well as component graphs as components composed of other components.

Definition 21 (Parallel Composition). *Let A and B be two components with disjoint input interfaces P_A and P_B and disjoint output interfaces Q_A and Q_B . Then the parallel composition of A and B is a component $A||B$ with input interface $P_{A||B} = P_A \cup P_B$, output interface $Q_{A||B} = Q_A \cup Q_B$ and the relation between input and output interfaces as follows:*

$$R_{A||B} = \{(X_A \cup X_B, Y_A \cup Y_B) \in Tr(P_{A||B}) \times Tr(Q_{A||B}) \mid X_A A Y_A \wedge X_B B Y_B\}$$

For serial and feedback compositions we need the following notion of component connectivity:

Definition 22 (Component Connectivity). *Let A and B be two components with input interfaces P_A and P_B and output interfaces Q_A and Q_B , respectively. Furthermore let $Q_A^* \subseteq Q_A$ and $P_B^* \subseteq P_B$ be two disjoint interfaces with the same numbers of ports and let $\Xi : Q_A^* \rightarrow P_B^*$ be a bijective mapping. Then it holds that component B is connectible to component A given mapping Ξ , i.e. $A \rightarrow_{\Xi} B$, iff it holds that P_B^* is connectible to Q_A^* given mapping Ξ , i.e. $Q_A^* \rightarrow_{\Xi} P_B^*$.*

This allows us to define serial composition as follows:

Definition 23 (Serial Composition). *Let A and B be two components with disjoint input interfaces P_A and $P_B = P_B^{\diamond} \cup P_B^*$ and disjoint output interfaces $Q_A = Q_A^* \cup Q_A^{\diamond}$ and Q_B , respectively, with $P_B^{\diamond} \cap P_B^* = Q_A^* \cap Q_A^{\diamond} = \emptyset$. Furthermore let Q_A^* and P_B^* be two disjoint interfaces with the same numbers of ports and let $\Xi : Q_A^* \rightarrow P_B^*$ be a bijective mapping.*

If B is connectible to A given mapping Ξ , i.e. $A \rightarrow_{\Xi} B$, then the serial composition of A and B given mapping Ξ is obtained by connecting interface P_B^ to interface Q_A^* via an interface connection C^{Ξ} . This results in a component $A \Xi B$ with input interface $P_{A \Xi B} = P_A \cup P_B^{\diamond}$, output interface $Q_{A \Xi B} = Q_A^{\diamond} \cup Q_B$ and the relation between input and output interfaces as follows:*

$$R_{A \Xi B} = \{(X_A \cup X_B^{\diamond}, Y_A^{\diamond} \cup Y_B) \in Tr(P_{A \Xi B}) \times Tr(Q_{A \Xi B}) \mid \\ \exists_{X_A A (Y_A^* \cup Y_A^{\diamond})} : \exists_{(X_B^{\diamond} \cup C^{\Xi}(Y_A^*)) B Y_B} \wedge \forall_{X_A A (Y_A^* \cup Y_A^{\diamond})} : \exists_{(X_B^{\diamond} \cup C^{\Xi}(Y_A^*)) B Y_B}\}$$

The first part of the relation ensures that an input is only accepted by the composition $A \Xi B$ if there exists a corresponding output of A to that input which is also accepted by B . And the second part addresses potential non-determinism of A , such that an input is only accepted by $A \Xi B$ if all possible outputs of A for that input are also accepted by B . This prevents the occurrence of dead states.

Lastly, we define feedback composition as follows:

Definition 24 (Feedback Composition). *Let A be a component with input interface $P_A = P_A^\circ \cup P_A^*$ and output interface $Q_A = Q_A^* \cup Q_A^\circ$, with $P_A^\circ \cap P_A^* = Q_A^* \cap Q_A^\circ = \emptyset$. Furthermore let Q_A^* and P_A^* be two disjoint interfaces with the same numbers of ports and let $\Xi : Q_A^* \rightarrow P_A^*$ be a bijective mapping.*

If A is connectible to A given mapping Ξ , i.e. $A \rightarrow_\Xi A$, then the feedback composition of A given mapping Ξ is obtained by connecting interface P_A^ to interface Q_A^* via an interface connection C^Ξ . This results in a component $A \Xi A$ with input interface $P_{A \Xi A} = P_A^\circ$, output interface $Q_{A \Xi A} = Q_A^\circ$ and the relation between input and output interfaces as follows (with $Y^{*,\infty}$ the empty trace on interface Q_A^*):*

$$R_{A \Xi A} = \{(X^\circ, Y^\circ) \in Tr(P_{A \Xi A}) \times Tr(Q_{A \Xi A}) \mid$$

$$Y_{-1}^* = Y_{-1}^{*\bullet} = Y^{*,\infty} : \exists (X^\circ \cup C^\Xi(Y_{k-1}^{*\bullet}))A(Y_k^* \cup Y_k^\circ) : \exists Y^{* \bullet \bullet} \cup Y^\circ = \lim_{k \rightarrow \infty} Y_k^{* \bullet \bullet} \cup Y_k^\circ \wedge$$

$$\forall (X^\circ \cup C^\Xi(Y_{k-1}^{*\bullet \bullet}))A(Y_k^{* \bullet \bullet} \cup Y_k^\circ) : \exists Y^{* \bullet \bullet} \cup Y^\circ = \lim_{k \rightarrow \infty} Y_k^{* \bullet \bullet} \cup Y_k^\circ\}$$

For a component A without feedback the relation of the component is only applied once for each input trace X , resulting in one output trace Y . For a component $A \Xi A$ with feedback, however, the relation of the component is applied multiple times for each external input trace X° until a fixed point is reached, as the internal input trace on interface P_A^* of the underlying component A depends on the internal output trace on interface Q_A^* of the same component. According to the definition of components, such a sequence of multiple applications always begins with the empty trace on the internal input interface. The second line of the relation captures this by ensuring that an input is only accepted by $A \Xi A$ if A has a fixed point for this input that is reachable, starting from the empty trace on the internal interface. Note that this makes our feedback composition fundamentally different to the one in TETB, which only requires existence of fixed points. Compared to TETB, the reformulation facilitates an expression of operational components in our denotational timed component model and relaxes the conditions under which automatic lifting of bounding and input acceptance is given. The third line again addresses potential non-determinism of A , preventing dead states by ensuring that an input is only accepted by $A \Xi A$ if not only one, but any sequence of internal traces, starting from the empty trace, converges to a fixed point.

Based on these compositions we can finally define graphs of components as follows:

Definition 25 (Component Graph). *A component graph G is itself a component composed of other components via parallel, serial and / or feedback compositions.*

In the TETB refinement theory the definition of refinement implies that given component refinement the input acceptance of components is automatically lifted from component to graph level. Moreover, the refinement relation is defined in such way that it is always possible to replace components by their refinements without reducing the input acceptance of the entire graph. In our theory both these implications do not hold in general, which is due to the fact that on component level we only require bounding, inclusion or exclusion instead of abstraction or refinement. In the following we consequently derive additional component properties for which both automatic lifting of input acceptance from component to graph level and replaceability can be shown. For that matter we introduce two important subclasses of components, input-complete and operational components, which are defined as follows:

Definition 26 (Input-Complete Component). *A component A with input interface P is input-complete iff it holds that it accepts all input traces, i.e. $in_A = Tr(P)$.*

Definition 27 (Operational Component). *A component A is operational iff it is continuous with respect to the prefix relation, i.e. it is $\leq$ -continuous.*

4.6.1 INPUT ACCEPTANCE LIFTING

In this section we derive conditions under which an automatic lifting of input acceptance from component to graph level applies, i.e. under which input acceptance is preserved on parallel, serial and feedback composition. This is of particular importance for the verification of input acceptance preservation on abstraction or refinement, as an automatic lifting of input acceptance allows to deduce graph level from component level input acceptance preservation. Afterwards we show that on compositions of input-complete and operational components the resulting components are also input-complete and operational. First we define some notation that is used in the following.

Definition 28 (Partial trace sets). *Let $Tr \subseteq Tr(P)$ be a set of traces on an interface P . We denote the set of streams on a port $p \in P$ that are part of traces in Tr as:*

$$Tr[p] = \{x \in St(p) \mid \exists_{X \in Tr}: x = X[p]\}$$

Likewise, we denote the set of traces on an interface $P' \subseteq P$ that are part of traces in Tr as:

$$Tr[P'] = \{X' \in Tr(P') \mid \exists_{X \in Tr}: \forall_{p \in P'}: X'[p] = X[p]\}$$

Definition 29 (Output Set Connection). *Let A be a component whose output interface $Q_A^* \subseteq Q_A$ is connected to another component interface P_B^* via an interface connection C^Ξ . We denote the set of valid output traces of A with respect to interface P_B^* as:*

$$C^\Xi(out_A^*) = \{X \in Tr(P_B^*) \mid \exists_{Y \in out_A^*}: X = C^\Xi(Y)\}$$

Definition 30 (Compositional Trace Relation). *A relation $\propto$ on traces is compositional iff it is also defined for streams, such that it holds on any interface P that $X \propto X' \Leftrightarrow \forall p \in P: X[p] \propto X'[p]$.*

Corollary 4.2 (Compositional Extrema). *Let $\propto$ be a compositional trace relation. Then it holds that extrema with respect to this relation are preserved on merging and splitting of interfaces, i.e. it holds for any set of traces Tr on an interface $P = P' \cup P''$ that:*

$$\begin{aligned} \sup_{\propto}(Tr) &= \sup_{\propto}(Tr[P']) \cup \sup_{\propto}(Tr[P'']) \\ \inf_{\propto}(Tr) &= \inf_{\propto}(Tr[P']) \cup \inf_{\propto}(Tr[P'']) \end{aligned}$$

Next we introduce the subclasses of input-independent and empty-continuous components and subsequently establish their relations to input-complete and operational components:

Definition 31 (Input-Independent Component). *A component A with input interface P_A is input-independent iff it holds that input acceptance per port is independent, i.e. $in_A = \bigotimes_{p \in P_A} in_A[p]$.*

Corollary 4.3 (Input-Complete vs. Input-Independent). *An input-complete component is input-independent.*

Definition 32 (Empty-Continuous Component). *A component A with output interface Q_A is empty-continuous iff it holds that the component is continuous with respect to a reflexive, compositional trace relation $\propto$ for which the empty trace X^∞ on any interface P is a global infimum according to $\propto$, i.e. $X^\infty = \inf_{\propto}(Tr(P))$.*

Corollary 4.4 (Operational vs. Empty-Continuous). *An operational component is empty-continuous.*

Using these subclasses we define sufficient conditions for the automatic lifting of input acceptance. We first prove that under certain conditions input acceptance is preserved on parallel, feedback and serial composition and afterwards discuss input acceptance lifting from component to graph level.

Lemma 4.1 (Parallel Input Acceptance Preservation). *Let A and B be two components and $A||B$ a parallel composition according to Definition 21. Then the input acceptance of A and B is preserved on parallel composition $A||B$, i.e. it holds that $in_{A||B} = in_A \times in_B$. Moreover, the parallel composition $A||B$ is input-independent / empty-continuous with respect to a relation $\propto$ if both A and B are also input-independent / empty-continuous with respect to the same relation.*

Proof. Trivial. □

Lemma 4.2 (Serial Input Acceptance Preservation). *Let A and B be two components and $A \Xi B$ a serial composition according to Definition 23. Then the input acceptance of A and B is preserved on serial composition $A \Xi B$, i.e. it holds that*

$in_{A \Xi B} = in_A \times in_B^\diamond$, if B accepts all outputs of A , i.e. $C^\Xi(out_A^*) \subseteq in_B^*$, and if B is input-independent. Moreover, the serial composition $A \Xi B$ is input-independent if A is also input-independent. Finally, $A \Xi B$ is empty-continuous with respect to a relation α if both A and B are empty-continuous with respect to the same relation.

Proof Idea. From the input-independence of B and $C^\Xi(out_A^*) \subseteq in_B^*$ it follows that B accepts all combinations of external inputs and outputs of A , i.e. input acceptance of A and B is preserved on serial composition $A \Xi B$. Showing that $A \Xi B$ is input-independent if A is additionally input-independent is trivial.

To prove empty-continuity of $A \Xi B$ if both A and B are additionally empty-continuous we have to show that for any sub-relation of the relation of $A \Xi B$ the supremum of the respective sub-input set results in the supremum of the respective sub-output set. For that matter we derive the sub-relations of A and B corresponding to the sub-relation of $A \Xi B$ and the respective sub-input and sub-output sets of A and B . Then we use the empty-continuity of A to show that the supremum of the sub-input set of A results in the supremum of the sub-output set of A . We further show that this supremum of the sub-output set of A results in the supremum of the sub-input set of B . With the empty-continuity of B we can then conclude that this supremum results in the respective supremum of the sub-output set of B . This proves that for any sub-relation of $A \Xi B$ the input suprema are accepted and result in the respective output suprema, i.e. $A \Xi B$ is empty-continuous.

Proof. As B is input-independent it holds that $in_B = in_B^\diamond \times in_B^*$. With $C^\Xi(out_A^*) \subseteq in_B^*$ this implies that $in_B^\diamond \times C^\Xi(out_A^*) \subseteq in_B$. From this follows that it holds $\forall_{X_A \in in_A, X_B^\diamond \in in_B^\diamond}$:

$$\forall_{X_A A(Y_A^* \cup Y_A^\diamond) \exists (X_B^\diamond \cup C^\Xi(Y_A^*)) B Y_B}$$

This lets us conclude with the definition of serial composition that $R_{A \Xi B}$ is defined $\forall_{X_A \in in_A, X_B^\diamond \in in_B^\diamond}$, i.e. it holds that $in_{A \Xi B} = in_A \times in_B^\diamond$.

If also A is input-independent it holds additionally that $in_A = \bigotimes_{p \in P_A} in_A[p]$ and as B is input-independent that $in_B^\diamond = \bigotimes_{p \in P_B^\diamond} in_B[p]$, from which follows that $A \Xi B$ is input-independent.

Lastly, consider that A and B are in addition both empty-continuous with respect to a relation α . In the following we make use of the fact that by definition α must be a compositional trace relation, such that Corollary 4.2 holds, i.e. suprema are preserved on merging and splitting of interfaces. For any sub-relation $R'_{A \Xi B} \subseteq R_{A \Xi B}$ let $R'_A \subseteq R_A$ and $R'_B \subseteq R_B$ be the respective sub-relations of A and B according to the definition of serial composition, such that:

$$(X_A \cup X_B^\diamond, Y_A^\diamond \cup Y_B) \in R'_{A \Xi B} \Leftrightarrow (X_A, Y_A^* \cup Y_A^\diamond) \in R'_A \wedge (X_B^\diamond \cup C^\Xi(Y_A^*), Y_B) \in R'_B \quad (4.1)$$

Furthermore let $in'_{A \Xi B}$, $out'_{A \Xi B}$, in'_A , out'_A , in'_B and out'_B according to Definition 20.

We need to prove that $(\sup_\alpha(in'_{A \Xi B}), \sup_\alpha(out'_{A \Xi B})) \in R_{A \Xi B}$. For that matter let us consider $\sup_\alpha(in'_{A \Xi B})$ as input of $A \Xi B$. From Equation 4.1 it follows that

$in'_A = in'_{A \Xi B}[P_A]$ and therefore also $sup_{\alpha}(in'_A) = sup_{\alpha}(in'_{A \Xi B}[P_A])$. With the empty-continuity of A we obtain that $sup_{\alpha}(in'_A)$ is accepted by A , as well as that $sup_{\alpha}(in'_A)A sup_{\alpha}(out'_A)$. From Equation 4.1 it can be further deduced that $in'_B[P_B^{\circ}] = in'_{A \Xi B}[P_B^{\circ}]$ and $in'_B[P_B^*] = C^{\Xi}(out'_A[Q_A^*])$. With Corollary 4.2 it then follows that $sup_{\alpha}(in'_B) = sup_{\alpha}(in'_{A \Xi B}[P_B^{\circ}]) \cup C^{\Xi}(sup_{\alpha}(out'_A)[Q_A^*])$.

With the empty-continuity of B we obtain that $sup_{\alpha}(in'_B)$ is accepted by B , as well as that $sup_{\alpha}(in'_B)B sup_{\alpha}(out'_B)$. From Equation 4.1 we get $out'_{A \Xi B}[Q_A^{\circ}] = out'_A[Q_A^{\circ}]$, as well as $out'_{A \Xi B}[Q_B] = out'_B$ and with Corollary 4.2 it follows that $sup_{\alpha}(out'_{A \Xi B}) = (sup_{\alpha}(out'_A)[Q_A^{\circ}] \cup sup_{\alpha}(out'_B))$.

As the respective suprema are accepted by both A and B , as $sup_{\alpha}(in'_{A \Xi B})$ is accepted by $A \Xi B$ due to the previously proven input acceptance preservation and as for this input $A \Xi B$ has the output $sup_{\alpha}(out'_{A \Xi B})$, i.e. $(sup_{\alpha}(in'_{A \Xi B}), sup_{\alpha}(out'_{A \Xi B})) \in R_{A \Xi B}$, we can finally conclude that $A \Xi B$ is empty-continuous. $\square$

Lemma 4.3 (Feedback Input Acceptance Preservation). *Let A be a component and $A \Xi A$ a feedback composition according to Definition 24. Then the input acceptance of A is preserved on feedback composition $A \Xi A$, i.e. it holds that $in_{A \Xi A} = in_A^{\circ}$, if A accepts all outputs of A , i.e. $C^{\Xi}(out_A^*) \subseteq in_A^*$, and if A is both input-independent and empty-continuous with respect to a relation α . Moreover, the feedback composition $A \Xi A$ is also both input-independent and empty-continuous with respect to relation α .*

Proof Idea. Let there be an external input that is, together with an internal input, accepted by A . Following the definition of feedback composition, we reason in sequences of inputs and outputs on the respective internal input and output interfaces. We make use of four properties to prove that the external input is also accepted by $A \Xi A$, i.e. that input acceptance is preserved. First, the starting point of all sequences, i.e. the combination of the external input with the empty trace is accepted by A , due to the empty trace being accepted by any component and A being input-independent. Second, any combination of the external input with an internal output of A is also accepted by A , due to A being input-independent and $C^{\Xi}(out_A^*) \subseteq in_A^*$. These first two properties ensure that all input sequences are accepted by A , i.e. no dead states can occur. Third, starting from the empty trace on the internal interface, all following inputs and outputs are α -larger than the last, as empty-continuity implies that the empty trace is the global infimum and that A is α -monotone. This ensures that both input and output sequences keep increasing, i.e. A cannot get into an oscillating state for any sequence. And fourth, convergence to fixed points of all input-output sequences is guaranteed, as the fixed point of any input-output sequence is just the supremum of all feedback iterations of that sequence and as the empty-continuity of A implies acceptance of suprema. This ensures that the respective limits exist for all sequences, which proves input acceptance preservation.

Proving input-independence is again trivial with the input acceptance preservation of A and A being input-independent.

To prove empty-continuity of $A\Xi A$ we have to show that for any sub-relation of the relation of $A\Xi A$ the supremum of the respective sub-input set results in the supremum of the respective sub-output set. According to the definition of feedback composition, any element of a sub-relation of $A\Xi A$ is a fixed point of a certain input-output sequence of A , starting with the empty trace on the internal interface. For any such sequence we can now construct sub-relations of A such that the k 'th sub-relation contains all input-output pairs that represent the k 'th feedback iteration of the sequences leading to aforementioned fixed points. For instance, the first sub-relation of A contains all pairs of the first inputs and outputs of the sequences, with the first inputs combining the empty trace with the respective external inputs in the sub-input set of $A\Xi A$. Then we consider the supremum of the sub-input set of $A\Xi A$ as input of A . By repetitively using the empty-continuity of A with respect to α for all sub-relations of A we show that the supremum of the sub-input set of $A\Xi A$ also results in a fixed point of A , which is the supremum of the sub-output set of $A\Xi A$. This proves that $A\Xi A$ is empty-continuous with respect to relation α .

Proof. To prove input acceptance preservation we have to show, according to the definition of feedback composition, that $\forall X^\diamond \in in_A^\diamond$ it holds that $Y^* \cup Y^\diamond$ exists for all sequences $(X^\diamond \cup C^\Xi(Y_{k-1}^*))A(Y_k^* \cup Y_k^\diamond)$ with $Y_{-1}^* = Y^{*,\infty}$ such that $\lim_{k \rightarrow \infty} Y_k^* \cup Y_k^\diamond = Y^* \cup Y^\diamond$. For that matter we need to prove that each $X^\diamond \cup C^\Xi(Y_k^*)$ is accepted by A and that each sequence converges for $k \rightarrow \infty$, i.e. it does not get into an oscillating state (note that “oscillating” in this context does not mean that values of a trace may be oscillating indefinitely in time, but that entire traces are oscillating).

In the following we denote with $X_k = (X^\diamond \cup X_k^*)$ and $Y_k = (Y_k^* \cup Y_k^\diamond)$ the input and output traces of component A in each feedback iteration. Moreover, we denote with $\{X_k\} = \bigcup_{0 \leq k' \leq k} X_{k'}$ and $\{Y_k\} = \bigcup_{0 \leq k' \leq k} Y_{k'}$ sets containing all respective input and output traces up to and including k .

Assuming $X^\diamond \in in_A^\diamond$ we first prove using mathematical induction that for all k all $X_k A Y_k$ exist and that it holds for all k that $X_k \alpha X_{k+1}$ and $Y_k \alpha Y_{k+1}$, i.e. A cannot get into an oscillating state. The latter implies for all k that $\sup_\alpha(\{X_k\}) = X_k$, as well as $\sup_\alpha(\{Y_k\}) = Y_k$. Based on this observation we show that also $\sup_\alpha(\{X_\infty\}) = X_\infty$ and $\sup_\alpha(\{Y_\infty\}) = Y_\infty$ exist.

Induction base: Let $X^\diamond \in in_A^\diamond$. With A accepting the empty trace by definition and with A being input-independent, i.e. $in_A = in_A^\diamond \times in_A^*$, it follows that $X_0 = (X^\diamond \cup C^\Xi(Y^{*,\infty})) \in in_A$, i.e. $X_0 A Y_0$. With $Y_0 = (Y_0^* \cup Y_0^\diamond)$ we have $X_1 = (X^\diamond \cup C^\Xi(Y_0^*))$. According to the definition of empty-continuity it holds that $Y^{*,\infty}$ is the global infimum with respect to α . From this follows that $Y^{*,\infty} \alpha Y_0^*$ and with α being both reflexive and compositional that also $X_0 \alpha X_1$. From $C^\Xi(out_A^*) \subseteq in_A^*$ and the input-independence of A we further obtain that $X_1 \in in_A$, i.e. $X_1 A Y_1$. Finally we have that according to Corollary 4.1 the continuity of A with respect to α implies monotonicity with respect to α , such that we can conclude from $X_0 \alpha X_1$ that also $Y_0 \alpha Y_1$.

Induction hypothesis: Let $X_{k-1}AY_{k-1}$ and X_kAY_k and let $X_{k-1} \propto X_k$, as well as $Y_{k-1} \propto Y_k$.

Induction step: With $Y_{k-1} = (Y_{k-1}^* \cup Y_{k-1}^\diamond)$, $X_k = (X^\diamond \cup C^\Xi(Y_{k-1}^*))$, $Y_k = (Y_k^* \cup Y_k^\diamond)$ and $X_{k+1} = (X^\diamond \cup C^\Xi(Y_k^*))$ we get from $Y_{k-1}^* \propto Y_k^*$ and $\propto$ being reflexive and compositional that $X_k \propto X_{k+1}$. From $C^\Xi(out_A^*) \subseteq in_A^*$ and the input-independence of A we further obtain that $X_{k+1} \in in_A$, i.e. $X_{k+1}AY_{k+1}$. As the continuity of A with respect to $\propto$ implies monotonicity with respect to $\propto$ we get from $X_k \propto X_{k+1}$ that also $Y_k \propto Y_{k+1}$.

From the induction we can thus conclude that if $X^\diamond \in in_A^\diamond$ it holds for all k that all X_kAY_k exist and that $X_k \propto X_{k+1}$, as well as $Y_k \propto Y_{k+1}$. From the latter we further obtain for all k that $sup_\propto(\{X_k\}) = X_k$ and that $sup_\propto(\{Y_k\}) = Y_k$. Now let $R'_A = \bigcup_{0 \leq k' \leq k} R_{A,k'} \subseteq R_A$ such that $in'_A = \{X_k\}$ and $out'_A = \{Y_k\}$ according to Definition 20. The continuity of A with respect to $\propto$ implies that $sup_\propto(in'_A)A sup_\propto(out'_A)$ must exist for all $R'_A \subseteq R_A$, even if the number of elements in R'_A is infinite and if $sup_\propto(in'_A) \notin in'_A$. This finally allows us to conclude that if $X^\diamond \in in_A^\diamond$ that also $sup_\propto(\{X_\infty\})A sup_\propto(\{Y_\infty\})$, i.e. $X_\infty AY_\infty$, exists for any sequence of X_k and Y_k , which proves input acceptance preservation.

Moreover, with $in_{A \Xi A} = in_A^\diamond$ and A being input-independent it immediately follows that $in_{A \Xi A} = \bigotimes_{p \in P_A^\diamond} in_A[p]$, i.e. $A \Xi A$ is also input-independent.

Lastly, we prove that $A \Xi A$ is empty-continuous with respect to $\propto$. In the following we make use of the fact that by definition $\propto$ must be a compositional trace relation, such that Corollary 4.2 holds, i.e. suprema are preserved on merging and splitting of interfaces. For any sub-relation $R'_{A \Xi A} \subseteq R_{A \Xi A}$ let $\forall k: R'_{A,k} \subseteq R_A$ and $R'_{A,\infty} \subseteq R_A$ be the respective sub-relations of A according to the definition of feedback composition, such that with $Y_{-1}^* = Y^{*,\infty}$, as well as $\lim_{k \rightarrow \infty} Y_k^* \cup Y_k^\diamond = Y_\infty^* \cup Y_\infty^\diamond$:

$$\begin{aligned} (X^\diamond, Y_\infty^\diamond) \in R'_{A \Xi A} \iff \forall k: (X^\diamond \cup C^\Xi(Y_{k-1}^*), Y_k^* \cup Y_k^\diamond) \in R'_{A,k} \wedge \\ (X^\diamond \cup C^\Xi(Y_\infty^*), Y_\infty^* \cup Y_\infty^\diamond) \in R'_{A,\infty} \end{aligned} \quad (4.2)$$

Furthermore let $in'_{A \Xi A}$, $out'_{A \Xi A}$, $in'_{A,k}$, $out'_{A,k}$, $in'_{A,\infty}$ and $out'_{A,\infty}$ for all k according to Definition 20. We need to prove that $(sup_\propto(in'_{A \Xi A}), sup_\propto(out'_{A \Xi A})) \in R_{A \Xi A}$. For that matter we use the same notation as above and make use of an induction-based proof again.

Induction base: For an $R'_{A \Xi A} \subseteq R_{A \Xi A}$ let $sup_\propto(in'_{A \Xi A})$ be an input of $A \Xi A$. From Equation 4.2 it follows that $in'_{A,0}[P_A^\diamond] = in'_{A \Xi A}$ and that $in'_{A,0}[P_A^*] = \{C^\Xi(Y^{*,\infty})\}$. Consequently it also holds that $sup_\propto(in'_{A,0}[P_A^\diamond]) = sup_\propto(in'_{A \Xi A})$, as well as that $sup_\propto(in'_{A,0}[P_A^*]) = C^\Xi(Y^{*,\infty})$. With the empty-continuity of A we obtain that $sup_\propto(in'_{A,0})$ is accepted by A and that $sup_\propto(in'_{A,0})A sup_\propto(out'_{A,0})$.

Induction hypothesis: Let $sup_\propto(in'_{A,k})$ be accepted by A and let it hold that $sup_\propto(in'_{A,k})A sup_\propto(out'_{A,k})$.

Induction step: From Equation 4.2 it follows that $in'_{A,k+1}[P_A^\diamond] = in'_{A \Xi A}$ and that $in'_{A,k+1}[P_A^*] = C^\Xi(out'_{A,k}[Q_A^*])$. Thus it also holds that $sup_\propto(in'_{A,k+1})[P_A^\diamond] =$

$\sup_{\alpha}(in'_{A \sqcup A})$ and $\sup_{\alpha}(in'_{A, k+1})[P_A^*] = C^{\Xi}(\sup_{\alpha}(out'_{A, k}[Q_A^*]))$. With the empty-continuity of A we obtain that $\sup_{\alpha}(in'_{A, k+1})$ is accepted by A , as well as that $\sup_{\alpha}(in'_{A, k+1})A \sup_{\alpha}(out'_{A, k+1})$.

From the induction we can conclude that if $R'_{A \sqcup A} \subseteq R_{A \sqcup A}$ it holds for all k that $\sup_{\alpha}(in'_{A, k})$ is accepted by A , as well as that $\sup_{\alpha}(in'_{A, k})A \sup_{\alpha}(out'_{A, k})$, such that $\sup_{\alpha}(in'_{A, k})[P_A^{\diamond}] = \sup_{\alpha}(in'_{A \sqcup A})$ and $\sup_{\alpha}(in'_{A, k})[P_A^*] = C^{\Xi}(\sup_{\alpha}(out'_{A, k-1}[Q_A^*])$.

Finally, it follows from Equation 4.2 that $in'_{A, \infty}[P_A^{\diamond}] = in'_{A \sqcup A}$ and $in'_{A, \infty}[P_A^*] = C^{\Xi}(out'_{A, \infty}[Q_A^*])$. Thus it also holds that $\sup_{\alpha}(in'_{A, \infty})[P_A^{\diamond}] = \sup_{\alpha}(in'_{A \sqcup A})$, as well as that $\sup_{\alpha}(in'_{A, \infty})[P_A^*] = C^{\Xi}(\sup_{\alpha}(out'_{A, \infty}[Q_A^*]))$. With the empty-continuity of A we obtain that $\sup_{\alpha}(in'_{A, \infty})$ is accepted by A and that it moreover holds that $\sup_{\alpha}(in'_{A, \infty})A \sup_{\alpha}(out'_{A, \infty})$. According to Equation 4.2 it further follows that $out'_{A \sqcup A} = out'_{A, \infty}[Q_A^{\diamond}]$ and therewith $\sup_{\alpha}(out'_{A \sqcup A}) = \sup_{\alpha}(out'_{A, \infty}[Q_A^{\diamond}])$.

This lets us conclude that if $R'_{A \sqcup A} \subseteq R_{A \sqcup A}$ it holds for all k that $\sup_{\alpha}(in'_{A, k})$ is accepted by A and that $\sup_{\alpha}(in'_{A, k})A \sup_{\alpha}(out'_{A, k})$, as well as that it holds that $\sup_{\alpha}(in'_{A, \infty})$ is accepted by A and that $\sup_{\alpha}(in'_{A, \infty})A \sup_{\alpha}(out'_{A, \infty})$. From the definition of feedback composition it consequently follows that $\sup_{\alpha}(in'_{A \sqcup A})$ is accepted by $A \sqcup A$ and that $(\sup_{\alpha}(in'_{A \sqcup A}), \sup_{\alpha}(out'_{A \sqcup A})) \in R_{A \sqcup A}$, i.e. $A \sqcup A$ is empty-continuous. $\square$

Note that proving input-independence and empty-continuity of $A \sqcup A$ is not strictly needed for proving input acceptance lifting from component to graph level as feedback composition can be always done in a single last step after all parallel and serial compositions.

The preceding three lemmas allow us to establish the automatic lifting of input acceptance from component to graph level as follows:

Theorem 4.1 (Input Acceptance Lifting). *Let G be a component graph consisting of components A with input interface $P_G = \bigcup_A P_A^{\diamond}$. If the output sets of all components are subsets of the input sets of the respective connected components, if all components are input-independent and if all components are empty-continuous with respect to the same relation α , then it holds that the input acceptance of the individual components is lifted to the graph level, i.e. $in_G = \bigotimes_A in_A^{\diamond}$, as well as that G is input-independent and empty-continuous with respect to relation α .*

Proof. Follows immediately from Lemmas 4.1 to 4.3 and the fact that graphs consist of parallel, serial and feedback compositions. $\square$

Theorem 4.2 (Input-Completeness Lifting). *Let G be a component graph consisting of input-complete and operational components. Then it holds that the graph G is also input-complete and operational.*

Proof. Follows immediately from Corollaries 4.3, 4.4 and Theorem 4.1. $\square$

Finally we introduce the concept of strict input acceptance preservation that we use in the next section.

Definition 33 (Strictly Input Acceptance Preserving Graph). *We call a graph G strictly input acceptance preserving iff all its component compositions adhere to the requirements of Lemmas 4.1 to 4.3.*

Corollary 4.5 (Input-Completeness vs. Strict Input Acceptance Preservation). *A graph consisting of input-complete and operational components is strictly input acceptance preserving.*

Corollary 4.6 (Strictly Input Acceptance Preserving Transformation). *Let G be a graph consisting of input-independent and empty-continuous components. If we replace in G all components A, B in serial compositions $A \Xi B$ with components A', B' such that $R_{A'} \subseteq R_A$, $R_{B'} \subseteq R_B$, $C^\Xi(\text{out}_{A'}^*) \subseteq \text{in}_{B'}^*$ and $R_{A' \Xi B'} = R_{A \Xi B}$ and all components A in feedback compositions $A \Xi A$ with components A' such that $R_{A'} \subseteq R_A$, $C^\Xi(\text{out}_{A'}^*) \subseteq \text{in}_{A'}^*$ and $R_{A' \Xi A'} = R_{A \Xi A}$ then we obtain a strictly input acceptance preserving graph G' for which it holds that $R_{G'} = R_G$.*

4.6.2 REPLACEABILITY

In the TETB refinement theory, refinement is defined in such way that input acceptance of a graph is preserved on a one-by-one replacement of refining components. By only requiring bounding or inclusion instead of abstraction or refinement of individual components, we lose this property. For that reason we have to separately derive conditions under which a component of a graph can be replaced by another component such that the input acceptance of the entire graph is preserved. We define replaceability as follows:

Definition 34 (Replaceability). *Let component A be part of a component graph G . We say that component A is replaceable by a component A' iff the input set of the resulting graph G' after the replacement is larger or equal to the original input set, i.e. $\text{in}_{G'} \supseteq \text{in}_G$.*

In the following we derive conditions for component replaceability under parallel, serial and feedback composition. Based on these conditions we then discuss replaceability of components in graphs.

Lemma 4.4 (Replaceability on Parallel Composition). *Let A and B be two components and $A || B$ a parallel composition according to Definition 21. If component A is replaced by a component A' with the same input and output interfaces as A and with $\text{in}_{A'} \supseteq \text{in}_A$ then it holds that $\text{in}_{A' || B} \supseteq \text{in}_{A || B}$.*

Proof. Trivial. □

Lemma 4.5 (Replaceability on Serial Composition). *Let A and B be two components and $A \Xi B$ a serial composition according to Definition 23.*

If A is replaced by a component A' with the same input and output interfaces as A , if B is input-independent, if $\text{in}_{A'} \supseteq \text{in}_A$ and if $C^\Xi(\text{out}_{A'}^) \subseteq \text{in}_B^*$ then it holds that $\text{in}_{A' \Xi B} \supseteq \text{in}_{A \Xi B}$.*

Moreover, if B is replaced by an input-independent component B' with the same input and output interfaces as B , if $in_{B'}^\diamond \supseteq in_B^\diamond$ and if $in_{B'}^* \supseteq C^\Xi(out_A^*)$ then it holds that $in_{A\Xi B'} \supseteq in_{A\Xi B}$.

Proof. In the first case in which A is replaced by A' it holds that Lemma 4.2 applies after the replacement, such that $in_{A'\Xi B} = in_{A'} \times in_B^\diamond$. As it furthermore holds that $in_{A\Xi B} \subseteq in_A \times in_B^\diamond$ it follows with $in_{A'} \supseteq in_A$ that $in_{A'\Xi B} \supseteq in_{A\Xi B}$.

In the second case in which B is replaced by B' it also holds that Lemma 4.2 applies after the replacement, such that $in_{A\Xi B'} = in_A \times in_{B'}^\diamond$. As it furthermore holds that $in_{A\Xi B} \subseteq in_A \times in_B^\diamond$ it follows with $in_{B'}^\diamond \supseteq in_B^\diamond$ that $in_{A\Xi B'} \supseteq in_{A\Xi B}$. $\square$

Lemma 4.6 (Replaceability on Feedback Composition). *Let A be a component and $A\Xi A$ a feedback composition according to Definition 24. If A is replaced by an input-independent and empty-continuous component A' with the same input and output interfaces as A , if $in_{A'}^\diamond \supseteq in_A^\diamond$ and if $C^\Xi(out_{A'}^*) \subseteq in_{A'}^*$, then it holds that $in_{A'\Xi A'} \supseteq in_{A\Xi A}$.*

Proof. It holds that Lemma 4.3 applies after the replacement, such that $in_{A'\Xi A'} = in_{A'}^\diamond$. As it furthermore holds that $in_{A\Xi A} \subseteq in_A^\diamond$ it follows with $in_{A'}^\diamond \supseteq in_A^\diamond$ that $in_{A'\Xi A'} \supseteq in_{A\Xi A}$. $\square$

These lemmas allow us to establish the replaceability of components in entire graphs as follows:

Theorem 4.3 (Replaceability in Graphs). *Let component A be part of a component graph G that consists of input-independent and empty-continuous components. Then component A can be replaced by a component A' according to Definition 34 if A' adheres to all requirements imposed by Lemmas 4.4 to 4.6 with respect to itself and the components that A' is composed with.*

Proof. Follows immediately from Lemmas 4.4 to 4.6 and the fact that graphs consist of parallel, serial and feedback compositions. $\square$

This theorem defines conditions under which component replacement is allowed if knowledge about the output sets of predecessors and the input sets of successors is given. However, this knowledge may not always be available. For that purpose we additionally introduce the concept of independent replaceability as follows:

Theorem 4.4 (Independent Replaceability). *Let component A be part of a strictly input acceptance preserving component graph G . Then A can be replaced by a component A' according to Definition 34 if A' is input-independent, empty-continuous, has the same input and output interfaces as A and if both $in_{A'} \supseteq in_A$ and $out_{A'} \subseteq out_A$.*

Proof. In this proof in_A^* , out_A^* , $in_{A'}^*$ and $out_{A'}^*$ always refer to the respective discussed compositions. As G is strictly input acceptance preserving it holds for all components B serially preceding A , i.e. $B \in pred(A)$, that $in_A^* \supseteq C^\Xi(out_B^*)$ and for all components D serially succeeding A , i.e. $D \in succ(A)$, that $C^\Xi(out_A^*) \subseteq in_D^*$. Moreover, if A is feedback-composed, i.e. $A \Xi A$, it further holds that $C^\Xi(out_A^*) \subseteq in_A^*$. With $in_{A'}^* \supseteq in_A^*$ and $out_{A'}^* \subseteq out_A^*$ it immediately follows that $\forall_{B \in pred(A)} : in_{A'}^* \supseteq C^\Xi(out_B^*)$, that $\forall_{D \in succ(A)} : C^\Xi(out_{A'}^*) \subseteq in_D^*$, as well as that for feedback $C^\Xi(out_{A'}^*) \subseteq in_{A'}^*$ and that $in_{A'}^* \supseteq in_A^*$. Because all components of the graph after replacement are additionally both input-independent and empty-continuous it follows that Theorem 4.3 applies. $\square$

4.7 TIMED DATAFLOW MODELS

Timed dataflow models are useful as they allow to express both the temporal and functional behavior of streaming applications and as their simpler variants can be analyzed efficiently. In the following we first introduce different variants of timed dataflow models and subsequently discuss their expression in our timed component model.

Definition 35 (HSDF). A Homogeneous Synchronous Dataflow (HSDF) graph is a directed graph $G^{DF} = (V, E)$ that consists of a set of actors V and a set of directed edges E connecting these actors. An actor $v_a \in V$ communicates with other actors by producing tokens on and consuming tokens from edges, which represent unbounded queues. An edge $e_{a,b} = (v_a, v_b) \in E$ initially contains $\delta_{a,b}$ tokens. An actor v_a is enabled to fire iff at least one token is available on each of its incoming edges. Furthermore, the firing duration ρ_a specifies the difference between the start and finish times of a firing of an actor v_a . An actor consumes one token from each of its incoming edges at the start of a firing and produces one token on each of its outgoing edges when a firing finishes.

Definition 36 (SDF). An SDF graph is a generalization of an HSDF graph such that rate conversions are supported, i.e. an actor v_a consumes $\gamma_{c,a}$ tokens per firing instead of one from its incoming edges $e_{c,a}$ and produces $\pi_{a,b}$ tokens per firing instead of one on its outgoing edges $e_{a,b}$.

Definition 37 (CSDF). A Cyclo-Static Dataflow (CSDF) graph is a generalization of an SDF graph such that cyclo-static actor phases are supported, i.e. an actor v_a consists of Θ instead of one phases, with each phase θ with $0 \leq \theta < \Theta$ having phase-specific firing durations ρ_a^θ , phase-specific consumption rates $\gamma_{c,a}^\theta$ for the incoming edges $e_{c,a}$ and phase-specific production rates $\pi_{a,b}^\theta$ for the outgoing edges $e_{a,b}$.

Aside from the standard dataflow models HSDF, SDF and CSDF in which token consumptions and productions occur in timestamp-order we also consider dataflow models in which tokens are consumed and produced in index-order. Note that the consideration of index-order enables auto-concurrency of dataflow actors,

as indices allow to maintain the correct order of values even if the values are not ordered in time. We define dataflow models with support for auto-concurrency as follows:

Definition 38 (Auto-concurrent Dataflow). *Dataflow models with support for auto-concurrency (HSDF^a, SDF^a and CSDF^a) are a generalization of standard dataflow models (HSDF, SDF and CSDF) such that tokens are associated with indices and firings are conducted in index- instead of timestamp-order.*

As we show in the following we can express the different types of dataflow models quite simply in our timed component model. By defining both actor and token components we can express arbitrary dataflow graphs. We begin with the token component that is the same for all kinds of dataflow graphs:

Corollary 4.7 (Token Component). *The temporal and functional behavior of δ initial tokens on a dataflow edge e can be captured by a token component κ with a single input port p , a single output port q and the relation R_κ as follows:*

$$\begin{aligned} R_\kappa = \{ (x, y) \in St(p) \times St(q) \mid & x = x^\infty \Rightarrow y = y^\infty \wedge x \neq x^\infty \Rightarrow \\ & \forall_{i < \delta}: \tau_y(i) = 0, \vartheta_y(i) = \vartheta_i^{init} \wedge \\ & \forall_{i \geq \delta}: \tau_y(i) = \tau_x(i - \delta), \vartheta_y(i) = \vartheta_x(i - \delta) \} \end{aligned}$$

Note that the first line is needed because components are by definition required to produce empty output traces for empty input traces.

In the definition of actor components we use the index i to differ between different firings of actors. With $i^p(i)$ and $i^q(i)$ we indicate the corresponding indices in the respective input and output streams. Note that for SDF and CSDF $i^p(i)$ and $i^q(i)$ return multiple indices in the form of sets. We use the application of the indexed timestamp and value functions on such sets as a shorthand notation to indicate that the respective functions apply to all indices in the sets.

Corollary 4.8 (Standard Actor Component). *The temporal and functional behavior of a standard dataflow actor can be captured by an actor component v with an input interface P corresponding to the incoming edges of the actor, an output interface Q corresponding to the outgoing edges of the actor and the relation R_v as follows:*

$$\begin{aligned} R_v = \{ (X, Y) \in Tr(P) \times Tr(Q) \mid & \forall_{p \in P}: \forall_i: \tau_{X[p]}(i) \leq \tau_{X[p]}(i + 1) \wedge \\ & \forall_{q \in Q}: \forall_i: \tau_{Y[q]}(i^q(i)) = \max_{p \in P} (\tau_{X[p]}(i^p(i))) + \rho, \\ & \vartheta_{Y[q]}(i^q(i)) = g_{p \in P} (\vartheta_{X[p]}(i^p(i))) \} \end{aligned}$$

For an HSDF actor component each firing consumes exactly one token from all input edges and produces exactly one token on all output edges. The stream indices accessed by a firing with index i are thus equal to that firing, i.e.:

$$\forall_{i \geq 0, p \in P, q \in Q}: i^p(i) = i^q(i) = i$$

An SDF actor component accesses per firing γ_p stream indices on each input edge $p \in P$ and π_q stream indices on each output edge $q \in Q$. This results in:

$$\begin{aligned} \forall_{i \geq 0}: \forall_{p \in P}: i^p(i) &= \{i \cdot \gamma_p + i' \mid 0 \leq i' < \gamma_p\} \wedge \\ \forall_{q \in Q}: i^q(i) &= \{i \cdot \pi_q + i' \mid 0 \leq i' < \pi_q\} \end{aligned}$$

For a CSDF actor component the stream indices accessed per firing vary according to the cyclo-static phases of the actor. Thus we obtain with the phase corresponding to a firing with index i being $\theta(i) = i \% \Theta$ (with $\%$ the modulo operator):

$$\begin{aligned} \forall_{i \geq 0}: \forall_{p \in P}: i^p(i) &= \left\{ \sum_{i^*=0}^{i-1} \gamma_p^{\theta(i^*)} + i' \mid 0 \leq i' < \gamma_p^{\theta(i)} \right\} \wedge \\ \forall_{q \in Q}: i^q(i) &= \left\{ \sum_{i^*=0}^{i-1} \pi_q^{\theta(i^*)} + i' \mid 0 \leq i' < \pi_q^{\theta(i)} \right\} \end{aligned}$$

The first line in the definition of R_v ensures that the component only accepts and produces in-order traces, i.e. that timestamps of traces are monotonically increasing in their indices. This is needed as standard dataflow actors do not have a notion of indices. The second line models the temporal behavior of a dataflow actor. The firing duration ρ is intentionally defined ambiguously. It can be a constant, a range, a function, a relation or adhere to a probabilistic distribution or even a finite state machine. This allows to express a wide range of applications using dataflow models, as well as to apply efficient dataflow analysis techniques on the simpler variants with e.g. constant firing durations. Our abstraction-refinement theory just allows to seamlessly bridge this gap between expressibility and analyzability by using various abstraction layers. The third line finally models the functional behavior, assuming that g is a function or relation that determines the values for all output trace indices corresponding to a firing i dependent on the values of all input trace indices that correspond to the same firing i . Note that this effectively makes dataflow actors stateless (actors with state must be modeled explicitly by using self-edges with at least one token).

For a dataflow actor with support for auto-concurrency we obtain analogously:

Corollary 4.9 (Auto-concurrent Actor Component). *The temporal and functional behavior of a dataflow actor with support for auto-concurrency can be captured by an actor component v with an input interface P corresponding to the incoming edges of the actor, an output interface Q corresponding to the outgoing edges of the actor, the indexing functions $i^p(i)$ and $i^q(i)$ being the same as for a standard actor component and the relation R_v as follows:*

$$\begin{aligned} R_v = \{ (X, Y) \in Tr(P) \times Tr(Q) \mid \forall_{q \in Q}: \forall_i: \tau_{Y[q]}(i^q(i)) &= \max_{p \in P}(\tau_{X[p]}(i^p(i))) + \rho, \\ \vartheta_{Y[q]}(i^q(i)) &= g_{p \in P}(\vartheta_{X[p]}(i^p(i))) \} \end{aligned}$$

Finally note that the aforementioned dataflow models are just presented here as an example of applicability. In a similar fashion also more complex dataflow models like Variable-Rate Dataflow (VRDF), as well as other models of computation like KPNs or Timed Petri Nets can be expressed in our timed component model, allowing to construct combinations, as well as abstractions and refinements between different types of models.

Abstraction & Refinement

In this chapter we discuss the different variants of abstraction and refinement, namely bounding, inclusion and exclusion. Thereafter we detail several aspects of this theory, which highlight applicability for safety-critical systems and illustrate the trade-off between accuracy and complexity.

5.1 BOUNDING

In this section we introduce the notions of best-case and worst-case bounding abstraction and refinement, which are used to create best-case and worst-case models of implementations, as well as implementations from best-case and worst-case models. For that purpose we define the relation $\triangleleft$ to express lower-bounding of streams and traces. Given trace lower-bounding, we define the relations $\trianglelefteq$ and $\triangleleft$ to express best-case and worst-case lower-bounding of components. We show that composition of components preserves bounding, which, together with input acceptance preservation, enables bounding abstraction and refinement of component graphs.

5.1.1 STREAM & TRACE BOUNDING

The bounding of streams is defined as follows:

Definition 39 (Stream Bounding). *Let $x, x' \in St(p)$ be two streams on a port p . The bounding relation $\triangleleft$ for streams is defined as:*

$$x \triangleleft x' \equiv \forall i : \tau_x(i) \leq \tau_{x'}(i) \wedge \vartheta_x(i) \models_p \vartheta_{x'}(i)$$

Just like the prefix and earlier-than relations, the stream bounding relation has the same properties as the stream refinement relation $\sqsubseteq$ defined in [GTW11], which implies reusability of results. It can be seen that the set $St(p)$ forms a

lattice with respect to the bounding relation, with the null stream $x_p^0 \in St(p)$ the infimum and the empty stream $x_p^\infty \in St(p)$ the supremum.

The bounding relation for streams is lifted to a bounding relation for traces as follows:

Definition 40 (Trace Bounding). *Let $X, X' \in Tr(P)$ be two traces on an interface P . The bounding relation $\triangleleft$ for traces is defined as:*

$$X \triangleleft X' \equiv \forall_{p \in P} : X[p] \triangleleft X'[p]$$

The set of traces $Tr(P)$ consequently also forms a lattice with respect to the bounding relation, with the null trace $X_p^0 \in Tr(P)$ the infimum and the empty trace $X_p^\infty \in Tr(P)$ the supremum.

Next to that, we define another bounding relation for traces which are not necessarily on the same interface, but on interfaces that only share a subset of ports:

Definition 41 (Partial Trace Bounding). *Let $X \in Tr(P), X' \in Tr(P')$ be two traces on interfaces P and P' , respectively. The partial bounding relation $\triangleleft$ for traces is defined as:*

$$X \triangleleft X' \equiv \forall_{p \in P \cap P'} : X[p] \triangleleft X'[p]$$

Note that trace bounding is transitive, whereas partial trace bounding is not. Trace bounding and partial trace bounding are closely related, as shown with the following lemma:

Lemma 5.1 (Trace Bounding vs. Partial Trace Bounding). *Let X be a trace on an interface P and X' be a trace on an interface P' . It holds:*

$$X \triangleleft X' \Leftrightarrow P = P' \wedge X \triangleleft X'$$

Proof. Trivial. □

Crucial to our usage of (partial) trace bounding is further the merging and splitting of traces, for which we prove the following four lemmas:

Lemma 5.2 (Partial Trace Bounding Preservation on Merging). *Let X_0 be a trace on an interface P_0 , X_1 a trace on an interface P_1 , X'_0 a trace on an interface P'_0 and X'_1 a trace on an interface P'_1 , with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$. For such traces it holds iff $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ that partial trace bounding is preserved on merging of these traces to $X = X_0 \cup X_1$ and $X' = X'_0 \cup X'_1$, respectively, i.e.:*

$$X_0 \triangleleft X'_0 \wedge X_1 \triangleleft X'_1 \Rightarrow X \triangleleft X'$$

Proof. By definition it holds that X is a trace on interface $P = P_0 \cup P_1$, whereas X' is a trace on $P' = P'_0 \cup P'_1$. Thus it holds that:

$$P \cap P' = (P_0 \cup P_1) \cap (P'_0 \cup P'_1) = (P_0 \cap P'_0) \cup (P_0 \cap P'_1) \cup (P_1 \cap P'_0) \cup (P_1 \cap P'_1)$$

From $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ further follows $P \cap P' = (P_0 \cap P'_0) \cup (P_1 \cap P'_1)$. If $X_0 \triangleleft X'_0$ then it holds by definition that $\forall_{p \in P_0 \cap P'_0} : X_0[p] \triangleleft X'_0[p]$ and if $X_1 \triangleleft X'_1$ then it holds that $\forall_{p \in P_1 \cap P'_1} : X_1[p] \triangleleft X'_1[p]$. In this case it hence holds that also $\forall_{p \in P \cap P'} : X[p] \triangleleft X'[p]$, i.e. $X \triangleleft X'$.

We prove by contradiction that partial trace bounding is not preserved on merging of traces if $P_0 \cap P'_1 \neq \emptyset$ and / or $P'_0 \cap P_1 \neq \emptyset$. Consider traces X_0 on interface $P_0 = \{p_0, p_1\}$, X_1 on interface $P_1 = \{p_2\}$, X'_0 on interface $P'_0 = \{p_0\}$ and X'_1 on interface $P'_1 = \{p_1, p_2\}$. Then it holds that $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$, as well as $P_0 \cap P'_1 = \{p_1\} \neq \emptyset$ and $P'_0 \cap P_1 = \emptyset$. Furthermore let $X_0[p_0] \triangleleft X'_0[p_0]$, $X_0[p_1] \not\triangleleft X'_1[p_1]$ and $X_1[p_2] \triangleleft X'_1[p_2]$. Then it follows that $X_0 \triangleleft X'_0$, $X_1 \triangleleft X'_1$, but $X_0 \cup X_1 \not\triangleleft X'_0 \cup X'_1$. $\square$

Lemma 5.3 (Partial Trace Bounding Preservation on Splitting). *Let X be a trace on an interface P , $X_0 \subseteq X$ a subtrace of X on an interface $P_0 \subseteq P$ and $X_1 \subseteq X$ a subtrace of X on interface $P_1 = P \setminus P_0$. Likewise, let X' be a trace on an interface P' , $X'_0 \subseteq X'$ a subtrace of X' on an interface $P'_0 \subseteq P'$ and $X'_1 \subseteq X'$ a subtrace of X' on interface $P'_1 = P' \setminus P'_0$. Then it holds that partial trace bounding is preserved on splitting of trace X to X_0 and X_1 and of X' to X'_0 and X'_1 , respectively, i.e.:*

$$X \triangleleft X' \Rightarrow X_0 \triangleleft X'_0 \wedge X_1 \triangleleft X'_1$$

Proof. If $X \triangleleft X'$ it holds that $\forall_{p \in P \cap P'} : X[p] \triangleleft X'[p]$. From $P_0 \subseteq P$ and $P'_0 \subseteq P'$ further follows that $P_0 \cap P'_0 \subseteq P \cap P'$. Thus it also holds that $\forall_{p \in P_0 \cap P'_0} : X_0[p] \triangleleft X'_0[p]$, i.e. $X_0 \triangleleft X'_0$. $X_1 \triangleleft X'_1$ follows from $X \triangleleft X'$ analogously. $\square$

Lemma 5.4 (Trace Bounding Preservation on Merging). *Let X_0 be a trace on an interface P_0 , X_1 a trace on an interface P_1 , X'_0 a trace on an interface P'_0 and X'_1 a trace on an interface P'_1 , with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$. For such traces it holds that trace bounding is preserved on merging of these traces to $X = X_0 \cup X_1$ and $X' = X'_0 \cup X'_1$, respectively, i.e.:*

$$X_0 \triangleleft X'_0 \wedge X_1 \triangleleft X'_1 \Rightarrow X \triangleleft X'$$

Proof. $X_0 \triangleleft X'_0$ implies $P_0 = P'_0$ and $X_0 \triangleleft X'_0$, while $X_1 \triangleleft X'_1$ implies $P_1 = P'_1$ and $X_1 \triangleleft X'_1$. From that follows with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$ that also $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ and with Lemma 5.2 that $X \triangleleft X'$. As X is a trace on interface $P = P_0 \cup P_1$ and X' is a trace on interface $P' = P'_0 \cup P'_1$ it further follows with $P_0 = P'_0$ and $P_1 = P'_1$ that $P = P'$. $X \triangleleft X'$ and $P = P'$ finally let us conclude that $X \triangleleft X'$. $\square$

Lemma 5.5 (Trace Bounding Preservation on Splitting). *Let X be a trace on an interface P , $X_0 \subseteq X$ a subtrace of X on an interface $P_0 \subseteq P$ and $X_1 \subseteq X$ a subtrace of X on interface $P_1 = P \setminus P_0$. Likewise, let X' be a trace on an interface P' , $X'_0 \subseteq X'$*

a subtrace of X' on an interface $P'_0 \subseteq P'$ and $X'_1 \subseteq X'$ a subtrace of X' on interface $P'_1 = P' \setminus P'_0$. If $P_0 = P'_0$ and $P_1 = P'_1$ then it holds that trace bounding is preserved on splitting of trace X to X_0 and X_1 and of X' to X'_0 and X'_1 , respectively, i.e.:

$$X \triangleleft X' \Rightarrow X_0 \triangleleft X'_0 \wedge X_1 \triangleleft X'_1$$

Proof. Trivial. □

5.1.2 COMPONENT BOUNDING

Given trace bounding we define best-case bounding and worst-case bounding of components as follows:

Definition 42 (Component Bounding). *Let A, A' be two components with input interfaces P_A and $P_{A'}$ and output interfaces Q_A and $Q_{A'}$, respectively.*

Component A is a best-case lower bound on A' , i.e. $A \trianglelefteq A'$, iff it holds that $\forall X \in \text{in}_A, X' \in \text{in}_{A'}:$

$$X \triangleleft X' \Rightarrow \forall X'A'Y' \exists XAY: Y \triangleleft Y'$$

Analogously, component A is a worst-case upper bound on A' , i.e. $A \trianglerighteq A'$, iff it holds that $\forall X \in \text{in}_A, X' \in \text{in}_{A'}:$

$$X \triangleright X' \Rightarrow \forall X'A'Y' \exists XAY: Y \triangleright Y'$$

In words this means that A is a best-case lower bound (worst-case upper bound) on A' iff for every input trace X of A that is a partial lower bound (upper bound) on an input trace X' of A' there exists an output trace Y with XAY that is a partial lower bound (upper bound) on every output trace Y' with $X'A'Y'$.

5.1.3 BOUNDING LIFTING

In this section we discuss the automatic lifting of bounding from component to graph level, i.e. that two graphs bound each other if all their components bound each other. For that purpose we prove that bounding between individual components is preserved on parallel, serial and feedback compositions.

For parallel composition it holds:

Lemma 5.6 (Parallel Bounding Preservation). *With $\nabla = \trianglelefteq$ ($\nabla = \trianglerighteq$) let $A \nabla A'$ and $B \nabla B'$. Furthermore let $Q_A \cap Q_{B'} = Q_{A'} \cap Q_B = \emptyset$. Then it holds that the respective parallel compositions also bound each other, i.e. $A || B \nabla A' || B'$.*

Proof. Trivial as the input and output interfaces of the composed components are disjoint, as both input and output traces of parallel compositions $A || B$ and $A' || B'$ are merged from input and output traces of A and B and of A' and B' , respectively, and as $Q_A \cap Q_{B'} = Q_{A'} \cap Q_B = \emptyset$ guarantees that partial bounding is lifted from subtraces to merged output traces, according to Lemma 5.2. □

For serial composition we obtain analogously:

Lemma 5.7 (Serial Bounding Preservation). *With $\nabla = \trianglelefteq$ ($\nabla = \triangleright$) let $A \nabla A'$ and $B \nabla B'$. For the serial compositions $A \Xi B$ and $A' \Xi' B'$ let further $P_B^* \cap P_{B'}^\circ = P_{B'}^* \cap P_B^\circ = Q_A^\circ \cap Q_{B'} = Q_{A'}^\circ \cap Q_B = \emptyset$ and let $\forall_{\Xi(q)=\Xi'(q') \in P_B^* \cap P_{B'}^* : q = q'}$. Then it holds that the respective serial compositions also bound each other, i.e. $A \Xi B \nabla A' \Xi' B'$.*

Proof Idea. From $A \nabla A'$ follows that for any input that is accepted by $A \Xi B$ and that partially bounds an input of $A' \Xi' B'$, there must be an output of A that partially bounds all respective outputs of A' . As these outputs must be accepted by B , according to the second line in the definition of serial composition, it follows with $B \nabla B'$ that there must also be an output of B that partially bounds all respective outputs of B' . This lets us conclude that for any input of $A \Xi B$ that partially bounds an input of $A' \Xi' B'$ there must be an output of $A \Xi B$ that partially bounds all respective outputs of $A' \Xi' B'$, i.e. $A \Xi B \nabla A' \Xi' B'$.

Proof. Let $(X_A \cup X_B^\circ) \in in_{A \Xi B}$, $(X'_A \cup X_{B'}^\circ) \in in_{A' \Xi' B'}$ and with $\Delta = \trianglelefteq$ ($\Delta = \triangleright$) let $(X_A \cup X_B^\circ) \Delta (X'_A \cup X_{B'}^\circ)$. With Lemma 5.3 it holds that also $X_A \Delta X'_A$ and from $A \nabla A'$ follows:

$$\forall_{X'_A A'(Y_A^{**} \cup Y_A^{\circ'})} \exists_{X_A A(Y_A^* \cup Y_A^\circ) : (Y_A^* \cup Y_A^\circ) \Delta (Y_A^{**} \cup Y_A^{\circ'})} \quad (5.1)$$

$(X_A \cup X_B^\circ) \in in_{A \Xi B}$ implies that it holds for all Y_A^* with $X_A A(Y_A^* \cup Y_A^\circ)$ that $(X_B^\circ \cup C^\Xi(Y_A^*)) \in in_B$. Analogously $(X'_A \cup X_{B'}^\circ) \in in_{A' \Xi' B'}$ implies that it holds for all Y_A^{**} with $X'_A A'(Y_A^{**} \cup Y_A^{\circ'})$ that $(X_{B'}^{\circ'} \cup C^{\Xi'}(Y_A^{**})) \in in_{B'}$.

For $(Y_A^* \cup Y_A^\circ)$ and $(Y_A^{**} \cup Y_A^{\circ'})$ according to Equation 5.1 it follows from $(Y_A^* \cup Y_A^\circ) \Delta (Y_A^{**} \cup Y_A^{\circ'})$ with Lemma 5.3 and $\forall_{\Xi(q)=\Xi'(q') \in P_B^* \cap P_{B'}^* : q = q'}$ that $C^\Xi(Y_A^*) \Delta C^{\Xi'}(Y_A^{**})$. With Lemma 5.2 and $P_B^* \cap P_{B'}^\circ = P_{B'}^* \cap P_B^\circ = \emptyset$ it further follows that $(X_B^\circ \cup C^\Xi(Y_A^*)) \Delta (X_{B'}^{\circ'} \cup C^{\Xi'}(Y_A^{**}))$. With $B \nabla B'$ we obtain:

$$\forall_{(X_B^{\circ'} \cup C^{\Xi'}(Y_A^{**})) B' Y_{B'}^{\circ'}} \exists_{(X_B^\circ \cup C^\Xi(Y_A^*)) B Y_B} : Y_B \Delta Y_{B'}^{\circ'}$$

From this follows with Lemma 5.2 and $Q_A^\circ \cap Q_{B'} = Q_{A'}^\circ \cap Q_B = \emptyset$ that also $(Y_A^\circ \cup Y_B) \Delta (Y_A^{\circ'} \cup Y_{B'}^{\circ'})$, such that it can be finally concluded $\forall_{(X_A \cup X_B^\circ) \in in_{A \Xi B}} \forall_{(X'_A \cup X_{B'}^\circ) \in in_{A' \Xi' B'}}$:

$$\begin{aligned} & (X_A \cup X_B^\circ) \Delta (X'_A \cup X_{B'}^\circ) \\ & \Rightarrow \forall_{(X'_A \cup X_{B'}^\circ) A' \Xi' B' (Y_A^{\circ'} \cup Y_{B'}^{\circ'})} \exists_{(X_A \cup X_B^\circ) A \Xi B (Y_A^\circ \cup Y_B)} : (Y_A^\circ \cup Y_B) \Delta (Y_A^{\circ'} \cup Y_{B'}^{\circ'}) \end{aligned}$$

This is just the definition of $A \Xi B \nabla A' \Xi' B'$, q.e.d. $\square$

Note that Lemma 5.7 does neither imply $A \Xi B \nabla A \Xi' B'$ nor $A \Xi B \nabla A' \Xi' B$ in general.

For feedback composition it holds:

Lemma 5.8 (Feedback Bounding Preservation). *With $\nabla = \triangleleft$ ($\nabla = \triangleleft$) let $A \nabla A'$. For the feedback compositions $A \Xi A$ and $A' \Xi' A'$ let further $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and let $\forall_{\Xi(q) = \Xi'(q') \in P_A^* \cap P_{A'}^*} : q = q'$. Then it holds that the respective feedback compositions also bound each other, i.e. $A \Xi A \nabla A' \Xi' A'$.*

Proof Idea. Let there be an input that is accepted by $A \Xi A$ and that partially bounds an input that is accepted by $A' \Xi' A'$. Consequently, the respective combinations of these traces with empty traces on the internal interfaces also partially bound each other. From $A \nabla A'$ then follows that for these inputs there exists an output of A that partially bounds all respective outputs of A' . With the third line of the definition of feedback composition it follows that these outputs must be accepted by A and A' as inputs, respectively, again resulting in an output of A that partially bounds all outputs of A' . As the third line ensures that for any such partially bounding sequences fixed points are reached, it follows that also the respective fixed points partially bound each other. This lets us conclude that for any input of $A \Xi A$ that partially bounds an input of $A' \Xi' A'$ there exists an output of $A \Xi A$ that partially bounds all respective outputs of $A' \Xi' A'$, i.e. $A \Xi A \nabla A' \Xi' A'$.

Proof. We denote with $X_k = (X^\diamond \cup X_k^*)$ and $Y_k = (Y_k^* \cup Y_k^\diamond)$ the input and output traces of component A in each feedback iteration k after assignment of an external input trace $X^\diamond$ and with $X'_k = (X^{\diamond'} \cup X_k^{*'})$ and $Y'_k = (Y_k^{*'} \cup Y_k^{\diamond'})$ the input and output traces of A' after assignment of $X^{\diamond'}$ analogously.

Given that an external input of A is a partial lower bound (upper bound) on an external input of A' we first show with $\Delta = \triangleleft$ ($\Delta = \triangleright$) that for each feedback iteration there exist output traces of A that are partial lower bounds (upper bounds) on all output traces of A' , using mathematical induction. Based on this we prove that there exists a fixed point of A for each fixed point of A' , such that the fixed point of A is a partial lower bound (upper bound) on the fixed points of A' . We begin with the mathematical induction:

Induction base: Let $X^\diamond \in in_{A \Xi A}$, $X^{\diamond'} \in in_{A' \Xi' A'}$ and $X^\diamond \Delta X^{\diamond'}$. It follows from the definition of components that initially (before $X^\diamond$ and $X^{\diamond'}$ are assigned) the traces on all interfaces are empty traces, such that the input traces of A on assignment of $X^\diamond$ and A' on assignment of $X^{\diamond'}$ are $X_0 = (X^\diamond \cup C^\Xi(Y^{*,\infty}))$ and $X'_0 = (X^{\diamond'} \cup C^{\Xi'}(Y^{*,\infty}'))$, respectively. With the definition of feedback composition it follows from $X^\diamond \in in_{A \Xi A}$ that $X_0 \in in_A$ and from $X^{\diamond'} \in in_{A' \Xi' A'}$ that $X'_0 \in in_{A'}$. From $Y^{*,\infty} \Delta Y^{*,\infty}$ follows with $\forall_{\Xi(q) = \Xi'(q') \in P_A^* \cap P_{A'}^*} : q = q'$ that also $C^\Xi(Y^{*,\infty}) \Delta C^{\Xi'}(Y^{*,\infty}')$ and from $X^\diamond \Delta X^{\diamond'}$ it follows with $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and Lemma 5.2 that $X_0 \Delta X'_0$. From $A \nabla A'$ we can then conclude that there exists an $X_0 A Y_0$ for all $X'_0 A' Y'_0$ such that $Y_0 \Delta Y'_0$.

Induction hypothesis: Let $X_k \in in_A$, $X'_k \in in_{A'}$, let $X_k \Delta X'_k$ and let an $X_k A Y_k$ exist for all $X'_k A' Y'_k$ such that $Y_k \Delta Y'_k$.

Induction step: With the definition of feedback composition it follows from $X^\diamond \in in_{A \Xi A}$ that $X_{k+1} = (X^\diamond \cup C^\Xi(Y_k^*)) \in in_A$ and from $X^{\diamond'} \in in_{A' \Xi' A'}$ that $X'_{k+1} = (X^{\diamond'} \cup C^{\Xi'}(Y_k^{*'})) \in in_{A'}$. From $Y_k \Delta Y'_k$ it follows with $\forall_{\Xi(q) = \Xi'(q') \in P_A^* \cap P_{A'}^*} : q = q'$

$q = q'$ that also $C^\Xi(Y_k) \Delta C^{\Xi'}(Y'_k)$ and from $X^\diamond \Delta X^{\diamond'}$ follows with $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and Lemma 5.2 that $X_{k+1} \Delta X'_{k+1}$. $A \nabla A'$ lets us conclude that there exists an $X_{k+1}AY_{k+1}$ for all $X'_{k+1}A'Y'_{k+1}$ such that $Y_{k+1} \Delta Y'_{k+1}$.

Thus it holds for $X^\diamond \in in_{A \Xi A}$, $X^{\diamond'} \in in_{A' \Xi' A'}$ and $X^\diamond \Delta X^{\diamond'}$ that:

$$\forall_{k \in \mathbb{N}}: \forall_{X'_k A' Y'_k} \exists_{X_k A Y_k}: X_k \Delta X'_k \wedge Y_k \Delta Y'_k \quad (5.2)$$

According to the definition of feedback composition $X^\diamond \in in_{A \Xi A}$ implies that for any sequence $(X^\diamond \cup C^\Xi(Y_k^*))A(Y_{k+1}^* \cup Y_{k+1}^\diamond)$ with $Y_0^* = Y^{*,\infty}$ there exists a $Y_\infty = \lim_{k \rightarrow \infty} Y_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^\diamond \cup C^\Xi(Y_\infty^*))A(Y_\infty^* \cup Y_\infty^\diamond)$. Analogously $X^{\diamond'} \in in_{A' \Xi' A'}$ implies that for any sequence $(X^{\diamond'} \cup C^{\Xi'}(Y_k^{*'}))A'(Y_{k+1}^{*' \prime} \cup Y_{k+1}^{\diamond' \prime})$ with $Y_0^{*' \prime} = Y^{*',\infty}$ there exists a $Y'_\infty = \lim_{k \rightarrow \infty} Y'_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^{\diamond'} \cup C^{\Xi'}(Y_\infty^{*' \prime}))A'(Y_\infty^{*' \prime} \cup Y_\infty^{\diamond' \prime})$.

Now consider a sequence $(X^\diamond \cup C^\Xi(Y_k^*))A(Y_{k+1}^* \cup Y_{k+1}^\diamond)$ that for any sequence $(X^{\diamond'} \cup C^{\Xi'}(Y_k^{*' \prime}))A'(Y_{k+1}^{*' \prime} \cup Y_{k+1}^{\diamond' \prime})$ satisfies $X_k \Delta X'_k$ and $Y_k \Delta Y'_k$ for all $k \in \mathbb{N}$ (existence of such a sequence is guaranteed by Equation 5.2). Furthermore, let Y_∞ and Y'_∞ be the respective fixed points of such sequences. Then it holds that $Y_\infty \Delta Y'_\infty$ and with Lemma 5.3 also $Y_\infty^\diamond \Delta Y_\infty^{\diamond'}$. With $Y^\diamond = Y_\infty^\diamond$ and $Y^{\diamond'} = Y_\infty^{\diamond'}$ this lets us finally conclude $\forall_{X^\diamond \in in_{A \Xi A}, X^{\diamond'} \in in_{A' \Xi' A'}}: X^\diamond \Delta X^{\diamond'}$.

$$X^\diamond \Delta X^{\diamond'} \Rightarrow \forall_{(X^{\diamond'})A' \Xi' A'} \exists_{(X^\diamond)A \Xi A} (Y^\diamond): Y^\diamond \Delta Y^{\diamond'}$$

This is just the definition of $A \Xi A \nabla A' \Xi' A'$, q.e.d. $\square$

Note that in contrast to TETB refinement we do not need any further requirements on the individual components to prove serial and feedback bounding. First, input-completeness of individual components is not needed as, unlike refinement in TETB, bounding does not imply input acceptance preservation. Second, monotonicity is not needed as we make use of a different definition of component bounding, which does not only ensure output bounding for the same inputs like in TETB, but also bounding outputs for different, but bounding inputs. And third, continuity is also not needed for feedback bounding as we make use of a different definition of fixed points than TETB, such that not only existence, but also reachability of fixed points is ensured.

Lemmas 5.6 to 5.8 prove the automatic lifting of bounding from component to graph level:

Theorem 5.1 (Bounding Lifting). *With $\nabla = \triangleleft$ ($\nabla = \triangleright$) let G be a graph composed of components A and let G' be a graph composed of components A' . If it holds for all components of G that they are best-case lower bounds (worst-case upper bounds) on the respective components of G' , i.e. $A \nabla A'$, and if all compositions satisfy the additional interface requirements from Lemmas 5.6 to 5.8 then it follows that also the graph G is a best-case lower bound (worst-case upper bound) on graph G' , i.e. $G \nabla G'$.*

Proof. Follows immediately from Lemmas 5.6 to 5.8 and the fact that graphs consist of parallel, serial and feedback compositions. $\square$

5.1.4 BOUNDING TRANSITIVITY & REFLEXIVITY

The presented definition of component bounding, Definition 42 generalizes the TETB component refinement, as it does not imply input acceptance preservation. While this generalization relaxes the requirement of input acceptance preservation on component level, it also comes at the cost that general transitivity does not hold for component bounding, i.e. with $\nabla = \trianglelefteq$ ($\nabla = \trianglerighteq$) it does not hold in general that $A \nabla A' \nabla A'' \Rightarrow A \nabla A''$. Nevertheless, we can define and prove a restricted form of transitivity that is sufficient for our needs as follows:

Theorem 5.2 (Restricted Bounding Transitivity). *With $\nabla = \trianglelefteq$ ($\nabla = \trianglerighteq$) let $A \nabla A' \nabla A''$, let $Q_A = Q_{A'} = Q_{A''}$ and with $\Delta = \triangleleft$ ($\Delta = \triangleright$) let $\forall X \in in_A, X'' \in in_{A''}$:*

$$X \Delta X'' \Rightarrow \exists X' \in in_{A'} : X \Delta X' \Delta X''$$

Then it holds that $A \nabla A''$. This means that transitivity of component bounding applies given that the output interfaces of A , A' and A'' are the same and that for all partially bounding input traces in in_A and $in_{A''}$ there exists a trace in $in_{A'}$ such that partial bounding transitivity holds for these traces. In particular, the latter condition is always satisfied if either $in_A \subseteq in_{A'} \subseteq in_{A''}$ or $in_A \supseteq in_{A'} \supseteq in_{A''}$.

Proof. Let $\Delta = \triangleleft$ ($\Delta = \triangleright$). For any $X \in in_A, X'' \in in_{A''}$ with $X \Delta X''$ let there exist an $X' \in in_{A'}$ such that $X \Delta X' \Delta X''$. From $A \nabla A'$ follows that $\forall X' A' Y' \exists X A Y : Y \Delta Y'$ and from $Q_A = Q_{A'}$ with Lemma 5.1 that also $Y \Delta Y'$. Likewise it follows from $A' \nabla A''$ that $\forall X'' A'' Y'' \exists X' A' Y' : Y' \Delta Y''$ and from $Q_{A'} = Q_{A''}$ that also $Y' \Delta Y''$. From combining the two statements and taking into account that trace bounding is transitive (as opposed to partial trace bounding) it can be concluded that $Y \Delta Y''$ and with Lemma 5.1 that also $Y \Delta Y''$. This implies $A \nabla A''$.

Correctness of the last sentence of the theorem follows immediately from the reflexivity of partial trace bounding. $\square$

The restricted bounding transitivity allows us to create multiple layers of bounding components while still allowing to relate the temporal behavior of the highest and lowest levels to each other.

Next to transitivity also reflexivity of component bounding does not always hold, i.e. with $\nabla = \trianglelefteq$ ($\nabla = \trianglerighteq$) it does not hold in general that $A \nabla A$. Instead, reflexivity of bounding is tightly coupled to monotonicity, as shown by the following theorem:

Theorem 5.3 (Bounding Reflexivity vs. Monotonicity). *With $\nabla = \trianglelefteq$ ($\nabla = \trianglerighteq$) it holds that $A \nabla A$ is equivalent to A being best-case (worst-case) $\triangleleft$ -monotone.*

Proof. Follows immediately from Definitions 19 and 42. $\square$

The most important implication of the lack of general reflexivity is that Lemma 5.7 does neither imply $A' \Xi' B \nabla A \Xi B$ nor $A \Xi' B' \nabla A \Xi B$. Instead it must additionally hold that A is best-case (worst-case) $\triangleleft$ -monotone if only B is replaced by B' or that B is best-case (worst-case) $\triangleleft$ -monotone if only A is replaced by A' .

After defining bounding of streams, traces and components, as well as proving the automatic lifting of bounding from component to graph level we can now also define bounding abstraction and refinement. Subsequently we discuss that bounding abstraction and refinement can be directly concluded from input acceptance preservation and bounding, as already indicated in Figure 3.1.

Definition 43 (Bounding Abstraction). *A graph G is a best-case (worst-case) abstraction of a graph G' iff G accepts at least all input traces that G' also accepts, i.e. $in_G \supseteq in_{G'}$, and iff it holds for all accepted input traces of G' that for these input traces there exists an output trace of G that is a lower bound (upper bound) on all output traces of G' for the same input traces, i.e. with $\Delta = \triangleleft$ ($\Delta = \triangleright$) it holds:*

$$\forall_{X \in in_{G'}}: \forall_{XG'Y'} \exists_{XGY}: Y \Delta Y'$$

Definition 44 (Bounding Refinement). *A graph G' is a best-case (worst-case) refinement of a graph G iff G' accepts at least all input traces that G also accepts, i.e. $in_G \subseteq in_{G'}$, and iff it holds for all accepted input traces of G that for these input traces there exists an output trace of G that is a lower bound (upper bound) on all output traces of G' for the same input traces, i.e. with $\Delta = \triangleleft$ ($\Delta = \triangleright$) it holds:*

$$\forall_{X \in in_G}: \forall_{XG'Y'} \exists_{XGY}: Y \Delta Y'$$

Definition 43 applies to the creation of best-case and worst-case models for the analysis of an existing implementation (for which the specification lies in the implementation), whereas Definition 44 applies to the design of an implementation from a best-case and / or worst-case model (for which the specification lies in the model). Note that our definition of worst-case refinement is equal to the definition of TETB refinement, whereas our definition of best-case refinement corresponds to the The-Later-the-Better (TLTB) refinement mentioned in the future work section of [GTW11]. As we additionally consider a different notion of bounding abstraction, it follows that our abstraction-refinement theory is a generalization of the TETB refinement theory.

With above definitions of bounding abstraction and refinement we can conclude the following three important theorems:

Theorem 5.4 (Bounding Abstraction). *A graph G is a best-case (worst-case) abstraction of a graph G' if $G \trianglelefteq G'$ ($G \trianglerighteq G'$), if $P_G = P_{G'}$ and $Q_G = Q_{G'}$ and if $in_G \supseteq in_{G'}$.*

Proof. Follows immediately from Definitions 42 and 43, the fact that the $\triangleleft$ relation is reflexive, i.e. $X \triangleleft X$, as well as Lemma 5.1. $\square$

Theorem 5.5 (Bounding Refinement). *A graph G' is a best-case (worst-case) refinement of a graph G if $G \trianglelefteq G'$ ($G \trianglerighteq G'$), if $P_G = P_{G'}$ and $Q_G = Q_{G'}$ and if $in_G \subseteq in_{G'}$.*

Proof. Follows immediately from Definitions 42 and 44, the fact that the $\triangleleft$ relation is reflexive, i.e. $X \triangleleft X$, as well as Lemma 5.1. $\square$

Theorem 5.6 (Bounding Abstraction-Refinement Transitivity). *With $\Delta = \trianglelefteq$ ($\Delta = \trianglerighteq$) let it hold for three graphs G, G' and G'' that $G \Delta G' \Delta G''$, that $P_G = P_{G'} = P_{G''}$ and $Q_G = Q_{G'} = Q_{G''}$ and that $in_G \supseteq in_{G'} \supseteq in_{G''}$. Then it follows that G is a best-case (worst-case) bounding abstraction of G'' . If it holds instead that $in_G \subseteq in_{G'} \subseteq in_{G''}$ then it follows that G'' is a best-case (worst-case) bounding refinement of G .*

Proof. Follows immediately from Definitions 43 and 44, Theorems 5.2, 5.4 and 5.5, as well as the fact that the $\triangleleft$ relation is reflexive, i.e. $X \triangleleft X$. $\square$

5.2 INCLUSION & EXCLUSION

This section discusses our inclusion and exclusion abstraction and refinement theory, based on the component model introduced in the previous chapter.

We begin with traces, then define when a component includes another and thereafter prove that inclusion is automatically lifted from component to graph level. Lastly, we combine component inclusion with input acceptance preservation and therewith define inclusion and exclusion abstraction and refinement.

5.2.1 TRACE EQUALITY

Inclusion and exclusion abstraction and refinement are about components that produce the same output traces for the same input traces. For that matter we first formally define when two traces are the same, i.e. equal:

Definition 45 (Trace Equality). *Let $X, X' \in Tr(P)$ be two traces on an interface P . The equality relation $=$ for traces is defined as:*

$$X = X' \equiv \forall_{p \in P} : X[p] = X'[p]$$

Besides this rather trivial definition of trace equality we make use of another kind of equality, which allows to relate traces to each other that are not on the same interfaces, but on different ones with potentially a number of shared ports. This is needed in order to allow for component inclusion between components that do not have the same interfaces, effectively enabling an addition or removal of dependencies between components.

Definition 46 (Partial Trace Equality). *Let $X \in Tr(P), X' \in Tr(P')$ be two traces on interfaces P and P' , respectively. The partial equality relation $\doteq$ for traces is defined as:*

$$X \doteq X' \equiv \forall_{p \in P \cap P'} : X[p] \doteq X'[p]$$

Apparently, trace equality and partial trace equality are closely related, as highlighted by the following lemma:

Lemma 5.9 (Trace Equality vs. Partial Trace Equality). *Let X be a trace on an interface P and X' be a trace on an interface P' . It holds:*

$$X = X' \Leftrightarrow P = P' \wedge X \doteq X'$$

Proof. Trivial. $\square$

For our usage of (partial) trace equality it is important to define rules for the merging and splitting of traces. For that matter we prove the following four lemmas:

Lemma 5.10 (Partial Trace Equality Preservation on Merging). *Let X_0 be a trace on an interface P_0 , X_1 a trace on an interface P_1 , X'_0 a trace on an interface P'_0 and X'_1 a trace on an interface P'_1 , with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$. For such traces it holds iff $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ that partial trace equality is preserved on merging of these traces to $X = X_0 \cup X_1$ and $X' = X'_0 \cup X'_1$, respectively, i.e.:*

$$X_0 \doteq X'_0 \wedge X_1 \doteq X'_1 \Rightarrow X \doteq X'$$

Proof. By definition it holds that X is a trace on interface $P = P_0 \cup P_1$, whereas X' is a trace on $P' = P'_0 \cup P'_1$. Thus it holds that:

$$P \cap P' = (P_0 \cup P_1) \cap (P'_0 \cup P'_1) = (P_0 \cap P'_0) \cup (P_0 \cap P'_1) \cup (P_1 \cap P'_0) \cup (P_1 \cap P'_1)$$

From $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ further follows $P \cap P' = (P_0 \cap P'_0) \cup (P_1 \cap P'_1)$. If $X_0 \doteq X'_0$ then it holds by definition that $\forall_{p \in P_0 \cap P'_0} : X_0[p] = X'_0[p]$ and if $X_1 \doteq X'_1$ then it holds that $\forall_{p \in P_1 \cap P'_1} : X_1[p] = X'_1[p]$. In this case it hence holds that also $\forall_{p \in P \cap P'} : X[p] = X'[p]$, i.e. $X \doteq X'$.

We prove by contradiction that partial trace equality is not preserved on merging of traces if $P_0 \cap P'_1 \neq \emptyset$ and / or $P'_0 \cap P_1 \neq \emptyset$. Consider traces X_0 on interface $P_0 = \{p_0, p_1\}$, X_1 on interface $P_1 = \{p_2\}$, X'_0 on interface $P'_0 = \{p_0\}$ and X'_1 on interface $P'_1 = \{p_1, p_2\}$. Then it holds that $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$, as well as $P_0 \cap P'_1 = \{p_1\} \neq \emptyset$ and $P'_0 \cap P_1 = \emptyset$. Furthermore let $X_0[p_0] = X'_0[p_0]$, $X_0[p_1] \neq X'_1[p_1]$ and $X_1[p_2] = X'_1[p_2]$. Then it follows that $X_0 \doteq X'_0$, $X_1 \doteq X'_1$, but $X_0 \cup X_1 \neq X'_0 \cup X'_1$. $\square$

Lemma 5.11 (Partial Trace Equality Preservation on Splitting). *Let X be a trace on an interface P , $X_0 \subseteq X$ a subtrace of X on an interface $P_0 \subseteq P$ and $X_1 \subseteq X$ a subtrace of X on interface $P_1 = P \setminus P_0$. Likewise, let X' be a trace on an interface P' , $X'_0 \subseteq X'$ a subtrace of X' on an interface $P'_0 \subseteq P'$ and $X'_1 \subseteq X'$ a subtrace of X' on interface $P'_1 = P' \setminus P'_0$. Then it holds that partial trace equality is preserved on splitting of trace X to X_0 and X_1 and of X' to X'_0 and X'_1 , respectively, i.e.:*

$$X \doteq X' \Rightarrow X_0 \doteq X'_0 \wedge X_1 \doteq X'_1$$

Proof. If $X \doteq X'$ it holds that $\forall_{p \in P \cap P'} : X[p] = X'[p]$. From $P_0 \subseteq P$ and $P'_0 \subseteq P'$ further follows that $P_0 \cap P'_0 \subseteq P \cap P'$. Thus it also holds that $\forall_{p \in P_0 \cap P'_0} : X_0[p] = X'_0[p]$, i.e. $X_0 \doteq X'_0$. $X_1 \doteq X'_1$ follows from $X \doteq X'$ analogously. $\square$

Lemma 5.12 (Trace Equality Preservation on Merging). *Let X_0 be a trace on an interface P_0 , X_1 a trace on an interface P_1 , X'_0 a trace on an interface P'_0 and X'_1 a trace on an interface P'_1 , with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$. For such traces it holds that trace equality is preserved on merging of these traces to $X = X_0 \cup X_1$ and $X' = X'_0 \cup X'_1$, respectively, i.e.:*

$$X_0 = X'_0 \wedge X_1 = X'_1 \Rightarrow X = X'$$

Proof. $X_0 = X'_0$ implies $P_0 = P'_0$ and $X_0 \doteq X'_0$, while $X_1 = X'_1$ implies $P_1 = P'_1$ and $X_1 \doteq X'_1$. From that follows with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$ that also $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ and with Lemma 5.10 that $X \doteq X'$. As X is a trace on interface $P = P_0 \cup P_1$ and X' is a trace on interface $P' = P'_0 \cup P'_1$ it further follows with $P_0 = P'_0$ and $P_1 = P'_1$ that $P = P'$. $X \doteq X'$ and $P = P'$ finally let us conclude that $X = X'$. $\square$

Lemma 5.13 (Trace Equality Preservation on Splitting). *Let X be a trace on an interface P , $X_0 \subseteq X$ a subtrace of X on an interface $P_0 \subseteq P$ and $X_1 \subseteq X$ a subtrace of X on interface $P_1 = P \setminus P_0$. Likewise, let X' be a trace on an interface P' , $X'_0 \subseteq X'$ a subtrace of X' on an interface $P'_0 \subseteq P'$ and $X'_1 \subseteq X'$ a subtrace of X' on interface $P'_1 = P' \setminus P'_0$. If $P_0 = P'_0$ and $P_1 = P'_1$ then it holds that trace equality is preserved on splitting of trace X to X_0 and X_1 and of X' to X'_0 and X'_1 , respectively, i.e.:*

$$X = X' \Rightarrow X_0 = X'_0 \wedge X_1 = X'_1$$

Proof. Trivial. $\square$

5.2.2 COMPONENT INCLUSION

Based on partial trace equality we define component inclusion, i.e. the condition under which one components includes another, such that it can produce at least the (partially) equal output traces for the (partially) equal input traces as follows:

Definition 47 (Component Inclusion). *Let A, A' be two components with input interfaces P_A and $P_{A'}$ and output interfaces Q_A and $Q_{A'}$, respectively.*

Component A includes A' , i.e. $A \supseteq A'$, iff it holds $\forall_{X \in \text{in}_A, X' \in \text{in}_{A'}} \text{ that:}$

$$X \doteq X' \Rightarrow \forall_{X'A'Y'} \exists_{XAY}: Y \doteq Y'$$

In words this means that A includes A' iff for every input trace X of A that is partially equal to an input trace X' of A' it holds that for all output traces Y' with $X'A'Y'$ there exist partially equal output traces Y with XAY .

The usage of partial trace equality instead of trace equality allows the addition or removal of ports between different layers of component inclusion. To this end it is taken care that if for instance an input port of an including component A is removed compared to a component A' , the component A can still match all outputs of component A' . Effectively, dependencies are thus replaced by non-determinism, which can result in a reduction of complexity at the cost of a loss in accuracy.

5.2.3 INCLUSION LIFTING

In this section we prove the automatic lifting of inclusion from component to graph level, i.e. that two graphs include each other if all their components include each other. As a graph consists of parallel, serial and / or feedback compositions we first prove the preservation of inclusion on these compositions.

We begin with parallel inclusion:

Lemma 5.14 (Parallel Inclusion Preservation). *Let $A \supseteq A'$ and $B \supseteq B'$. Furthermore let $Q_A \cap Q_{B'} = Q_{A'} \cap Q_B = \emptyset$. Then it holds that the respective parallel compositions also include each other, i.e. $A||B \supseteq A'||B'$.*

Proof. Trivial as the input and output interfaces of the composed components are disjoint, as both input and output traces of parallel compositions $A||B$ and $A'||B'$ are merged from input and output traces of A and B and of A' and B' , respectively, and as $Q_A \cap Q_{B'} = Q_{A'} \cap Q_B = \emptyset$ guarantees that partial equality is lifted from subtraces to merged output traces, according to Lemma 5.10. $\square$

For serial inclusion we obtain analogously:

Lemma 5.15 (Serial Inclusion Preservation). *Let $A \supseteq A'$ and $B \supseteq B'$. For the serial compositions $A \Xi B$ and $A' \Xi' B'$ let further $P_B^* \cap P_{B'}^\diamond = P_{B'}^* \cap P_B^\diamond = Q_A^\diamond \cap Q_{B'} = Q_{A'}^\diamond \cap Q_B = \emptyset$ and let $\forall_{\Xi(q)=\Xi'(q') \in P_B^* \cap P_{B'}^*} : q = q'$. Then it holds that the respective serial compositions also include each other, i.e. $A \Xi B \supseteq A' \Xi' B'$.*

Proof Idea. From $A \supseteq A'$ follows that for any input that is accepted by $A \Xi B$ and that is partially equal to an input of $A' \Xi' B'$ there must for each output of A' exist a partially equal output of A . As these outputs of A must be accepted by B , according to the second line in the definition of serial composition, it follows with $B \supseteq B'$ that there must also be for any respective output of B' a partially equal output of B . This lets us conclude that for any input of $A \Xi B$ that is partially equal to an input of $A' \Xi' B'$ there must for any respective output of $A' \Xi' B'$ exist a partially equal output of $A \Xi B$, i.e. $A \Xi B \supseteq A' \Xi' B'$.

Proof. Let $(X_A \cup X_B^\diamond) \in in_{A \Xi B}$, $(X'_A \cup X_{B'}^\diamond) \in in_{A' \Xi' B'}$ and $(X_A \cup X_B^\diamond) \doteq (X'_A \cup X_{B'}^\diamond)$. With Lemma 5.11 it holds that also $X_A \doteq X'_A$ and from $A \supseteq A'$ follows:

$$\forall_{X'_A A'(Y_A^{**} \cup Y_A^{\diamond'})} \exists_{X_A A(Y_A^* \cup Y_A^\diamond)} : (Y_A^* \cup Y_A^\diamond) \doteq (Y_A^{**} \cup Y_A^{\diamond'}) \quad (5.3)$$

$(X_A \cup X_B^\diamond) \in in_{A \Xi B}$ implies that it holds for all Y_A^* with $X_A A(Y_A^* \cup Y_A^\diamond)$ that $(X_B^\diamond \cup C^\Xi(Y_A^*)) \in in_B$. Analogously $(X_A' \cup X_B^{\diamond'}) \in in_{A' \Xi' B'}$ implies that it holds for all $Y_A^{*'}$ with $X_A' A'(Y_A^{*'} \cup Y_A^{\diamond'})$ that $(X_B^{\diamond'} \cup C^{\Xi'}(Y_A^{*'})) \in in_{B'}$.

For $(Y_A^* \cup Y_A^\diamond)$ and $(Y_A^{*'} \cup Y_A^{\diamond'})$ according to Equation 5.3 it follows from $(Y_A^* \cup Y_A^\diamond) \doteq (Y_A^{*'} \cup Y_A^{\diamond'})$ with Lemma 5.11 and $\forall_{\Xi(q)=\Xi'(q') \in P_B^* \cap P_{B'}^*} : q = q'$ that $C^\Xi(Y_A^*) \doteq C^{\Xi'}(Y_A^{*'})$. With Lemma 5.10 and $P_B^* \cap P_{B'}^\diamond = P_{B'}^* \cap P_B^\diamond = \emptyset$ it further follows that $(X_B^\diamond \cup C^\Xi(Y_A^*)) \doteq (X_B^{\diamond'} \cup C^{\Xi'}(Y_A^{*'}))$. With $B \nabla B'$ we obtain:

$$\forall_{(X_B^{\diamond'} \cup C^{\Xi'}(Y_A^{*'}))B'Y_B'} \exists_{(X_B^\diamond \cup C^\Xi(Y_A^*))BY_B} : Y_B \doteq Y_B'$$

From this follows with Lemma 5.10 and $Q_{A'}^\diamond \cap Q_{B'} = Q_{A'}^\diamond \cap Q_B = \emptyset$ that also $(Y_A^\diamond \cup Y_B) \doteq (Y_A^{\diamond'} \cup Y_B')$, such that it can be finally concluded $\forall_{(X_A \cup X_B^\diamond) \in in_{A \Xi B}} \text{ and } \forall_{(X_A' \cup X_B^{\diamond'}) \in in_{A' \Xi' B'}} :$

$$(X_A \cup X_B^\diamond) \doteq (X_A' \cup X_B^{\diamond'}) \Rightarrow \forall_{(X_A' \cup X_B^{\diamond'})A'\Xi'(Y_A^{\diamond'} \cup Y_B')} \exists_{(X_A \cup X_B^\diamond)A \Xi B(Y_A^\diamond \cup Y_B)} : (Y_A^\diamond \cup Y_B) \doteq (Y_A^{\diamond'} \cup Y_B')$$

This is just the definition of $A \Xi B \supseteq A' \Xi' B'$, q.e.d. $\square$

Lastly, we prove preservation of inclusion on feedback composition:

Lemma 5.16 (Feedback Inclusion Preservation). *Let $A \supseteq A'$. For the feedback compositions $A \Xi A$ and $A' \Xi' A'$ let further $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and let $\forall_{\Xi(q)=\Xi'(q') \in P_{A'}^* \cap P_A^*} : q = q'$. Then it holds that the respective feedback compositions also include each other, i.e. $A \Xi A \supseteq A' \Xi' A'$.*

Proof Idea. Let there be an input that is accepted by $A \Xi A$ and that is partially equal to an input that is accepted by $A' \Xi' A'$. The respective combinations of these traces with empty traces on the internal interfaces are consequently also partially equal. From $A \supseteq A'$ then follows that for any respective output of A' there exists a partially equal output of A . With the third line of the definition of feedback composition it follows that these outputs must be accepted by A and A' as inputs, respectively, again resulting in that for any output of A' there exists a partially equal output of A . As the third line ensures that for any such partially equal sequences fixed points are reached, it follows that also the respective fixed points partially equal each other. This lets us conclude for any input of $A \Xi A$ that partially equals an input of $A' \Xi' A'$ that for any respective output of $A' \Xi' A'$ there exists a partially equal output of $A \Xi A$, i.e. $A \Xi A \supseteq A' \Xi' A'$.

Proof. We denote with $X_k = (X^\diamond \cup X_k^*)$ and $Y_k = (Y_k^* \cup Y_k^\diamond)$ the input and output traces of component A in each feedback iteration k after assignment of an external input trace $X^\diamond$ and with $X_k' = (X^{\diamond'} \cup X_k^{*'})$ and $Y_k' = (Y_k^{*'} \cup Y_k^{\diamond'})$ the input and output traces of A' after assignment of $X^{\diamond'}$ analogously.

Given that an external input of A is partially equal to an external input of A' we first show for each feedback iteration that for all output traces of A' there

exist partially equal output traces of A , using mathematical induction. Based on this we prove that for each fixed point of A' there exists a fixed point of A , such that the fixed points of A are partially equal to the fixed points of A' . We begin with the mathematical induction:

Induction base: Let $X^\diamond \in in_{A \sqsubseteq A}$, $X^{\diamond'} \in in_{A' \sqsubseteq A'}$ and $X^\diamond \doteq X^{\diamond'}$. It follows from the definition of components that initially (before $X^\diamond$ and $X^{\diamond'}$ are assigned) the traces on all interfaces are empty traces, such that the input traces of A on assignment of $X^\diamond$ and A' on assignment of $X^{\diamond'}$ are $X_0 = (X^\diamond \cup C^\Xi(Y^{*,\infty}))$ and $X'_0 = (X^{\diamond'} \cup C^{\Xi'}(Y^{*,\infty}))$, respectively. With the definition of feedback composition it follows from $X^\diamond \in in_{A \sqsubseteq A}$ that $X_0 \in in_A$ and from $X^{\diamond'} \in in_{A' \sqsubseteq A'}$ that $X'_0 \in in_{A'}$. From $Y^{*,\infty} \doteq Y^{*,\infty}$ follows with $\forall \Xi(q) = \Xi'(q') \in P_A^* \cap P_{A'}^* : q = q'$ that also $C^\Xi(Y^{*,\infty}) \doteq C^{\Xi'}(Y^{*,\infty})$ and from $X^\diamond \doteq X^{\diamond'}$ it follows with $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and Lemma 5.10 that $X_0 \doteq X'_0$. From $A \supseteq A'$ we can then conclude that there exist $X_0 A Y_0$ for all $X'_0 A' Y'_0$ such that $Y_0 \doteq Y'_0$.

Induction hypothesis: Let $X_k \in in_A$, $X'_k \in in_{A'}$, let $X_k \doteq X'_k$ and let $X_k A Y_k$ exist for all $X'_k A' Y'_k$ such that $Y_k \doteq Y'_k$.

Induction step: With the definition of feedback composition it follows from $X^\diamond \in in_{A \sqsubseteq A}$ that $X_{k+1} = (X^\diamond \cup C^\Xi(Y_k^*)) \in in_A$ and from $X^{\diamond'} \in in_{A' \sqsubseteq A'}$ that $X'_{k+1} = (X^{\diamond'} \cup C^{\Xi'}(Y_k^{*'})) \in in_{A'}$. From $Y_k \doteq Y'_k$ it follows with $\forall \Xi(q) = \Xi'(q') \in P_A^* \cap P_{A'}^* : q = q'$ that also $C^\Xi(Y_k) \doteq C^{\Xi'}(Y_k')$ and from $X^\diamond \doteq X^{\diamond'}$ follows with $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and Lemma 5.10 that $X_{k+1} \doteq X'_{k+1}$. $A \supseteq A'$ lets us conclude that there exist $X_{k+1} A Y_{k+1}$ for all $X'_{k+1} A' Y'_{k+1}$ such that $Y_{k+1} \doteq Y'_{k+1}$.

Thus it holds for $X^\diamond \in in_{A \sqsubseteq A}$, $X^{\diamond'} \in in_{A' \sqsubseteq A'}$ and $X^\diamond \doteq X^{\diamond'}$ that:

$$\forall k \in \mathbb{N} : \forall X'_k A' Y'_k \exists X_k A Y_k : X_k \doteq X'_k \wedge Y_k \doteq Y'_k \quad (5.4)$$

According to the definition of feedback composition $X^\diamond \in in_{A \sqsubseteq A}$ implies that for any sequence $(X^\diamond \cup C^\Xi(Y_k^*)) A (Y_{k+1}^* \cup Y_{k+1}^\diamond)$ with $Y_0^* = Y^{*,\infty}$ there exists a $Y_\infty = \lim_{k \rightarrow \infty} Y_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^\diamond \cup C^\Xi(Y_\infty^*)) A (Y_\infty^* \cup Y_\infty^\diamond)$. Analogously $X^{\diamond'} \in in_{A' \sqsubseteq A'}$ implies that for any sequence $(X^{\diamond'} \cup C^{\Xi'}(Y_k^{*'})) A' (Y_{k+1}^{*'} \cup Y_{k+1}^{\diamond'})$ with $Y_0^{*'} = Y^{*,\infty}$ there exists a $Y'_\infty = \lim_{k \rightarrow \infty} Y'_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^{\diamond'} \cup C^{\Xi'}(Y_\infty^{*'})) A' (Y_\infty^{*'} \cup Y_\infty^{\diamond'})$.

Now consider sequences $(X^\diamond \cup C^\Xi(Y_k^*)) A (Y_{k+1}^* \cup Y_{k+1}^\diamond)$ that for any sequence $(X^{\diamond'} \cup C^{\Xi'}(Y_k^{*'})) A' (Y_{k+1}^{*'} \cup Y_{k+1}^{\diamond'})$ satisfy $X_k \doteq X'_k$ and $Y_k \doteq Y'_k$ for all $k \in \mathbb{N}$ (existence of such sequences is guaranteed by Equation 5.4). Furthermore, let Y_∞ and Y'_∞ be the respective fixed points of such sequences. Then it holds that $Y_\infty \doteq Y'_\infty$ and with Lemma 5.11 also $Y_\infty^\diamond \doteq Y_\infty^{\diamond'}$. With $Y^\diamond = Y_\infty^\diamond$ and $Y^{\diamond'} = Y_\infty^{\diamond'}$ this lets us finally conclude $\forall X^\diamond \in in_{A \sqsubseteq A}, X^{\diamond'} \in in_{A' \sqsubseteq A'} :$

$$X^\diamond \doteq X^{\diamond'} \Rightarrow \forall (X^{\diamond'}) A' \Xi' A' (Y^{\diamond'}) \exists (X^\diamond) A \Xi A (Y^\diamond) : Y^\diamond \doteq Y^{\diamond'}$$

This is just the definition of $A \sqsubseteq A' \sqsubseteq A'$, q.e.d. $\square$

Based on the preservation of inclusion on parallel, serial and feedback composition we can prove the automatic lifting of inclusion from component to graph level as follows:

Theorem 5.7 (Component Graph Inclusion). *Let G be a graph composed of components A and let G' be a graph composed of components A' . If it holds for all components of G that they are including all respective components of G' , i.e. $A \supseteq A'$, and if all compositions satisfy the additional interface requirements from Lemmas 5.14 to 5.16 then it follows that also the graph G includes graph G' , i.e. $G \supseteq G'$.*

Proof. Follows immediately from Lemmas 5.14 to 5.16 and the fact that graphs consist of parallel, serial and feedback compositions. $\square$

5.2.4 INCLUSION TRANSITIVITY $\doteq$ REFLEXIVITY

Unlike other approaches, our definition of component inclusion does not have any restriction on input acceptance preservation. This comes at the cost that general transitivity does not hold for component inclusion, i.e. it does not hold in general that $A \supseteq A' \supseteq A'' \Rightarrow A \supseteq A''$. Nevertheless, we can define and prove a restricted form of transitivity that is sufficient for our needs as follows:

Theorem 5.8 (Restricted Inclusion Transitivity). *Let $A \supseteq A' \supseteq A''$, let $Q_A = Q_{A'} = Q_{A''}$ and let $\forall X \in in_A, X'' \in in_{A''}$:*

$$X \doteq X'' \Rightarrow \exists X' \in in_{A'} : X \doteq X' \doteq X''$$

Then it holds that $A \supseteq A''$. This means that transitivity of component inclusion applies given that the output interfaces of A, A' and A'' are the same and that for all partially equal input traces in in_A and $in_{A''}$ there exists a trace in $in_{A'}$ such that partial equality transitivity holds for these traces. In particular, the latter condition is always satisfied if either $in_A \subseteq in_{A'} \subseteq in_{A''}$ or $in_A \supseteq in_{A'} \supseteq in_{A''}$.

Proof. For any $X \in in_A, X'' \in in_{A''}$ with $X \doteq X''$ let there exist an $X' \in in_{A'}$ such that $X \doteq X' \doteq X''$. From $A \supseteq A'$ follows that $\forall X', A', Y' : \exists X, A, Y : Y \doteq Y'$ and from $Q_A = Q_{A'}$ with Lemma 5.9 that also $Y = Y'$. Likewise it follows from $A' \supseteq A''$ that $\forall X'', A'', Y'' : \exists X', A', Y' : Y' \doteq Y''$ and from $Q_{A'} = Q_{A''}$ that also $Y' = Y''$. From combining the two statements and taking into account that trace equality is transitive (as opposed to partial trace equality) it can be concluded that $Y = Y''$ and with Lemma 5.9 that also $Y \doteq Y''$. This implies $A \supseteq A''$.

Correctness of the last sentence of the theorem follows immediately from the reflexivity of partial trace equality. $\square$

Component inclusion is reflexive, as captured by the following theorem:

Theorem 5.9 (Inclusion Reflexivity). *It holds for any component A that $A \subseteq A$.*

Proof. Trivial. $\square$

Inclusion reflexivity implies that it follows unconditionally from $A \supseteq A'$ that $A \Xi B \supseteq A' \Xi B$, as well as it follows unconditionally from $B \supseteq B'$ that $A \Xi B \supseteq A \Xi B'$.

After defining equality of traces, inclusion of components, as well as proving the automatic lifting of inclusion from component to graph level we can now also define inclusion and exclusion abstraction and refinement. Thereafter we discuss that inclusion and exclusion abstraction and refinement can be directly concluded from input acceptance preservation and inclusion.

Definition 48 (Inclusion & Exclusion Abstraction). *A graph G is an inclusion (exclusion) abstraction of a graph G' iff G accepts at least all input traces that G' also accepts, i.e. $in_G \supseteq in_{G'}$, and iff it holds for all accepted input traces of G' that for each output trace of G' (G) there exists an equal output trace of G (G'), i.e.:*

$$\forall_{X \in in_{G'}}: \forall_{XGY'} \exists_{XGY}: Y = Y' \ (\forall_{XGY} \exists_{XGY'}: Y = Y')$$

Definition 49 (Inclusion & Exclusion Refinement). *A graph G' is an inclusion (exclusion) refinement of a graph G iff G' accepts at least all input traces that G also accepts, i.e. $in_G \subseteq in_{G'}$, and iff it holds for all accepted input traces of G that for each output trace of G' (G) there exists an equal output trace of G (G'), i.e.:*

$$\forall_{X \in in_G}: \forall_{XGY'} \exists_{XGY}: Y = Y' \ (\forall_{XGY} \exists_{XGY'}: Y = Y')$$

Analogous to bounding abstraction and refinement, it holds that Definition 48 applies to the creation of models for the analysis of an existing implementation (for which the specification lies in the implementation), while Definition 49 applies to the design process from a model to an implementation (for which the specification is given by the models).

With above definitions of inclusion and exclusion abstraction and refinement we can conclude the following three important theorems:

Theorem 5.10 (Inclusion & Exclusion Abstraction). *A graph G is an inclusion (exclusion) abstraction of a graph G' if it holds that $G \supseteq G'$ ($G \subseteq G'$), if $P_G = P_{G'}$ and $Q_G = Q_{G'}$ and if $in_G \supseteq in_{G'}$.*

Proof. Follows immediately from Definitions 47 and 48, as well as Lemma 5.9. $\square$

Theorem 5.11 (Inclusion & Exclusion Refinement). *A graph G' is an inclusion (exclusion) refinement of a graph G if it holds that $G \supseteq G'$ ($G \subseteq G'$), if $P_G = P_{G'}$ and $Q_G = Q_{G'}$ and if $in_G \subseteq in_{G'}$.*

Proof. Follows immediately from Definitions 47 and 49, as well as Lemma 5.9. $\square$

Theorem 5.12 (Inclusion & Exclusion Abstraction & Refinement Transitivity). *Let it hold for three graphs G , G' and G'' that $G \supseteq G' \supseteq G''$ ($G \subseteq G' \subseteq G''$), that $P_G = P_{G'} = P_{G''}$ and $Q_G = Q_{G'} = Q_{G''}$ and that $in_G \supseteq in_{G'} \supseteq in_{G''}$. Then it holds that G is an inclusion (exclusion) abstraction of G'' . If it holds instead that $in_G \subseteq in_{G'} \subseteq in_{G''}$ then it follows that G'' is an inclusion (exclusion) refinement of G .*

Proof. Follows immediately from Definitions 48 and 49, as well as Theorems 5.8, 5.10 and 5.11. $\square$

chapter 6

Case Studies

In this chapter we evaluate the applicability and utility of our timed component model and abstraction-refinement theory, with an emphasis on differences with the TETB refinement theory. As the obvious advantage of supporting best-case models is already discussed in Chapter 3, our focus is in the following on other implications of separating bounding and input acceptance preservation.

6.1 REORDERING

Consider the example depicted in Figure 6.1. It is often desirable to make an abstraction of a graph containing any-order components (i.e. components producing streams whose timestamp-order does not necessarily match index-order) like A' and B' to a graph consisting of in-order components (i.e. components producing streams whose timestamps are monotonically increasing in their indices) like A and B because in-order graphs can be analyzed more efficiently. However, with the original TETB theory [GTW11] this is not possible, due to the following two shortcomings: First, the definition of streams in TETB does not support the expression of reordering. And second, the requirement of input-completeness for preservation of refinement on both serial and feedback compositions is too strict to allow abstractions of any-order components to in-order ones, even if reordering were expressible.

Both these shortcomings are amended in [HB16], which introduces indices to allow the expression of reordering and which does not require input-completeness for preservation of refinement on serial and feedback compositions, but only that abstract components accept all outputs of the connected refined components. For our example this means that the abstract components A and B must accept the respective outputs of A' , i.e. formally for the serial composition $in_B \supseteq out_{A'}^\diamond$ and for the feedback composition $in_A^* \supseteq out_{A'}^*$. But as it holds that in-order is a subset of any-order, i.e. $io \subseteq ao$, these requirements are not satisfied.

Figure 6.1: In-order abstraction of an any-order implementation (*io*: in-order, *ao*: any-order).

Therefore one has to apply a trick to narrow the output sets of the components, like the one depicted on the left-hand side of Figure 6.1: At first, input-complete so-called identity components *ID* are inserted on the outgoing edges, which simply forward input streams to output streams, resulting in a graph equivalent to the lowest level. Then these identity components are abstracted to $\sqsubseteq$ -monotone, input-complete reorder components *R* which accept any-order streams and delay them such that they become in-order. Thereafter, these reorder components are serially composed with the components *A'* and *B'*, resulting in the graph on the second-to-highest level. This graph fulfills the requirements $in_B \supseteq out_A^\diamond$, and $in_A^* \supseteq out_{A'}^*$, which enables the final abstraction to the graph with components *A* and *B*.

In our theory none of this is needed. Given worst-case bounding of the individual components, i.e. $A' \triangleleft A$ and $B' \triangleleft B$, one can directly conclude that also the entire graphs bound each other, without any further requirements on the interfaces or monotonicity of components and without any additional tricks involving identity and reorder components. And together with the input acceptance preservation on graph level one can finally conclude abstraction.

6.2 INPUT SET WIDENING ON ABSTRACTION

While the in-order / any-order case from the previous section already illustrates the potential of our theory, one could still argue that the trick with the identity and reorder components is merely an inconvenience. As we illustrate with the following example, however, there are cases in which TETB simply fails, while our theory remains applicable.

Figure 6.2: Abstraction of periodic components A' and B' to input-complete components A and B (p : strictly periodic streams with period P , a : any streams).

Consider the components A' and B' at the bottom of Figure 6.2. The components only accept input streams that are strictly periodic with a period P . This is a realistic use-case for components representing tasks being executed on actual processors, for which the determined response times are only valid assuming a certain period. The goal is to abstract these component to dataflow components A and B , which are by definition input-complete.

Both TETB approaches [GTW11, HB16] fail for this example as the components violate the fundamental requirement of input acceptance preservation on component level, i.e. it would have to hold that $in_{A'} \supseteq in_A$ and $in_{B'} \supseteq in_B$, but due to the fact that periodic streams with a period P are a subset of any streams, i.e. $p \subseteq a$, this is clearly not the case. Now one could try to apply a similar trick as discussed in the previous section, using a “*periodize*” component instead of a reorder component which is input-complete and delays all streams to ones with a period of P . In this case, such components could be used not to narrow output sets, but to widen the respective input sets. However, for input streams with a period larger than P , the only valid delayed stream would be the empty stream. Consequently, the usage of such components would prevent any useful analysis, making TETB effectively inapplicable.

In our theory, in contrast, one only has to prove that A and B worst-case upper-bound A' and B' without considering input sets, as depicted on the right-hand side of Figure 6.2. To prove abstraction, one has to additionally show that the input sets of the respective graphs are widening towards abstraction, which is trivial here as both A and B are input-complete.

6.3 VALUE BOUNDING

Another novelty in our abstraction-refinement theory is the introduction of the value ordering relation $\models_p$ on a per-port basis. This allows for the introduction of value abstractions, similar to the abstractions in the abstract interpretation theory [CC77], as well as value refinement.

Consider the left graph depicted in Figure 6.3. The two components A'' and B'' describe implementations of two parts of a program that both operate on

Figure 6.3: Example for value bounding.

Input value	-5	-4	-3	-2	-1	0	1	2	3	4	5
$L_{A''}/\mu\text{s}$	3.2	3.4	3.6	3.8	3.5	1.1	2.4	2.1	2.4	2.2	2.3
$L_{B''}/\mu\text{s}$	6.0	6.3	6.2	5.9	6.3	8.0	4.2	4.5	4.0	4.4	4.5

Table 6.1: Exemplary end-to-end latencies of components A'' and B'' in Figure 6.3 for different inputs.

streams of signed integer values. Our goal is to determine a conservative upper bound on the end-to-end latency of the graph G'' containing these components, i.e. to determine how long it maximally takes that an integer input value of component A'' results in an integer output value of component B'' . For that matter, assume that each of the components has been analyzed in separation, such that, depending on the branches taken in the respective sub-programs, different component latencies have been determined for all integers. Some of these values are depicted in Table 6.1.

To determine the end-to-end latency of the whole graph, one could now test all combinations of input-output values of both A'' and B'' , which would result in the highest accuracy. But already for this rather simple example of only two components, the number of combinations to test would be pretty large. In a graph that contained more components or even feedback compositions, the number of states that would have to be taken into account would inevitably explode.

Alternatively, one could simply take the maximum end-to-end latencies of both A'' and B'' to create a purely temporal abstraction of the graph G'' , using components A and B as depicted in the right graph in Figure 6.3. Suppose that it would have been determined that the maximum end-to-end latency of A'' for any integer input were $4\mu\text{s}$ and of B'' $8\mu\text{s}$. Then the relations of A and B could be defined as:

$$\begin{aligned}
 R_A &= \{(x, y) \in St(p_A) \times St(q_A) \mid \forall_i: \tau_y(i) = \tau_x(i) + 4.0\mu\text{s} \wedge \vartheta_y(i) = \vartheta^\infty\} \\
 R_B &= \{(x, y) \in St(p_B) \times St(q_B) \mid \forall_i: \tau_y(i) = \tau_x(i) + 8.0\mu\text{s} \wedge \vartheta_y(i) = \vartheta^\infty\}
 \end{aligned}$$

It can be easily seen that A and B are valid worst-case bounds of A'' and B'' , i.e. $A \trianglerighteq A''$ and $B \trianglerighteq B''$, and as A and B are input-complete also $in_G \supseteq in_{G''}$. This means that the graph G is a valid worst-case abstraction of G'' and the determined

end-to-end latency of $12\mu\text{s}$ for this graph would be a valid upper bound on the end-to-end latency of G'' . However, albeit easy to determine, this end-to-end latency appears rather conservative and one could ask whether there is not something in between maximum accuracy at highest complexity and minimum accuracy at lowest complexity.

Our theory just fills this gap. Consider that it has been determined, based on integer-specific latencies such as the ones in Table 6.1, that component A'' does not take more than $4\mu\text{s}$ to produce output values if its input values are negative, no more than $1.1\mu\text{s}$ on an input of 0 and no more than $2.5\mu\text{s}$ on positive inputs. Likewise, consider that B'' does not take more than $6.5\mu\text{s}$ on negative inputs, no more than $8\mu\text{s}$ on an input of 0 and no more than $4.5\mu\text{s}$ on positive inputs. Lastly, assume that A'' always produces positive outputs on negative inputs, an output of 0 for an input of 0 and negative outputs on positive inputs.

Given these observations, we can construct an abstraction of G'' that is both temporal and functional as follows. First, we define an extended value domain for all ports except the input port p_A , that does not only include all signed integer values, i.e. all values in $\mathbb{Z}$, and the infima and suprema ϑ^0 and ϑ^∞ , but also the elements $+$ and $-$, i.e. $O = \mathbb{Z} \cup \{\vartheta^0, \vartheta^\infty, +, -\}$. For this value domain we can then define a value relation $\models$, such that $\forall \vartheta \in O: \vartheta \models \vartheta \wedge \vartheta^0 \models \vartheta \models \vartheta^\infty$, as well as $\forall \vartheta \in \mathbb{Z}^-: \vartheta \models -$ and $\forall \vartheta \in \mathbb{Z}^+: \vartheta \models +$. Based on this, we can define the components A' and B' , that are depicted in the middle of Figure 6.3, via the following relations:

$$\begin{aligned}
 R_{A'} &= \{(x, y) \in St(p_A) \times St(q_A) \mid x = x^\infty \Rightarrow y = y^\infty \wedge \\
 &\quad x \neq x^\infty \Rightarrow \forall i: \vartheta_x(i) < 0 \Rightarrow \tau_y(i) = \tau_x(i) + 4.0\mu\text{s} \wedge \vartheta_y(i) = + \wedge \\
 &\quad \vartheta_x(i) = 0 \Rightarrow \tau_y(i) = \tau_x(i) + 1.1\mu\text{s} \wedge \vartheta_y(i) = 0 \wedge \\
 &\quad \vartheta_x(i) > 0 \Rightarrow \tau_y(i) = \tau_x(i) + 2.5\mu\text{s} \wedge \vartheta_y(i) = -\} \\
 R_{B'} &= \{(x, y) \in St(p_A) \times St(q_A) \mid x = x^\infty \Rightarrow y = y^\infty \wedge \\
 &\quad x \neq x^\infty \Rightarrow \forall i: \vartheta_x(i) = - \Rightarrow \tau_y(i) = \tau_x(i) + 6.5\mu\text{s} \wedge \vartheta_y(i) = \vartheta^\infty \wedge \\
 &\quad \vartheta_x(i) = 0 \Rightarrow \tau_y(i) = \tau_x(i) + 8.0\mu\text{s} \wedge \vartheta_y(i) = \vartheta^\infty \wedge \\
 &\quad \vartheta_x(i) = + \Rightarrow \tau_y(i) = \tau_x(i) + 4.5\mu\text{s} \wedge \vartheta_y(i) = \vartheta^\infty\}
 \end{aligned}$$

Using our previously defined relation $\models$ it can be seen that A' bounds A'' , but is not as conservative as A , i.e. $A'' \triangleleft A' \triangleleft A$, as well as analogously $B'' \triangleleft B' \triangleleft B$. Moreover, both G' and G'' accept any signed integer inputs, while G is input-complete, i.e. $in_G \supseteq in_{G'} \supseteq in_{G''}$. Consequently, G' is a valid worst-case abstraction of G'' and G is a valid worst-case abstraction of G' .

For the graph G' we can quite easily determine a maximum end-to-end latency by only considering three cases, namely that A' has a negative input (latency of $4 + 4.5 = 8.5\mu\text{s}$), that it has an input of 0 (latency of $1.1 + 8.0 = 10.1\mu\text{s}$) and that it has a positive input (latency of $2.5 + 6.5 = 9.0\mu\text{s}$). This gives us a maximum end-to-end latency of $10.1\mu\text{s}$ which is a valid upper bound on the end-to-end latency of G'' , as G' is a valid worst-case abstraction of G'' , but is less conservative than the end-to-end latency of $12\mu\text{s}$ determined with G .

6.4 REMOVING UNCERTAINTY & NON-DETERMINISM

We call a component non-deterministic if it has input traces for which it cannot produce only one, but multiple output traces. Non-determinism is usually introduced due to several uncertainties that are prevalent in reality. For example, it is practically impossible to determine the actual execution time of a task that is executed on a real processor. Several uncertainties like non-deterministic decisions, caching, memory port sharing, bus contention, processor sharing and even processor clock jitter make it extremely challenging to find the actual, accurate execution time of a task. Another example is the uncertainty with respect to the access order of shared variables between different tasks on different processors. And finally, there can be even multiple instances of the same task running on multiple resources, for which the exact moments at which resources become available cannot be determined, resulting in a possible reordering of data.

All such uncertainties can be addressed by inclusion abstraction, as inclusion abstraction allows to replace any kind of uncertainty by non-determinism. For instance, if the exact execution time is not known, the respective inclusion-abstract component can produce non-deterministic outputs within a range between best-case and worst-case execution times. Likewise, an unknown order of memory accesses, as well as an unknown availability of resources can be abstracted away by considering all cases non-deterministically.

However, the fundamental problem with non-deterministic models is that they usually cannot be analyzed efficiently. For that matter it is regularly required to remove non-determinism from a model. In general, there are two ways to remove non-determinism: Composition and bounding abstraction.

In some particular cases, it is possible to remove non-determinism by only composing components. For example, consider two serially connected components of which the first produces non-deterministic outputs for deterministic inputs and the second produces deterministic outputs for non-deterministic inputs. Obviously, the serial composition of the two produces deterministic outputs for deterministic inputs, despite some internal non-determinism. This means that in this case non-determinism can be removed by composition alone. However, such use-cases are rare. In the following we consequently focus on removal of non-determinism by bounding abstraction, a concept that is applicable on any use-case involving non-determinism.

For instance, assume the following scenario depicted at the bottom of Figure 6.4: A task *a* is executed on a single processor and produces outputs for a task *b*. Task *b* is executed in two instances, on two different processors. Task *a* always sends the next data item to the instance of *b* that is first available. Task *c* then gets the outputs of both instances of task *b* as inputs and always consumes the input of the instance that is first finished.

The problem with this setup is that the availability and finish times of both instances of *b* are uncertain at design time. Consequently, it is uncertain to which instance of *b* the task *a* sends data, as well as from which instance of *b* the

Figure 6.4: Example for removal of uncertainty & non-determinism (?: uncertainty, $\vee$: non-deterministic “or”, $\wedge$: deterministic “and”).

task c receives data. As discussed above, such uncertainties can be removed by inclusion abstraction. Let A' include task a , let B'_0 and B'_1 include the respective two instances of task b and let C' include task c . The uncertainty of a can be removed by introducing non-determinism in A' , such that A' can at any moment either produce an output on port q_0 or produce an output on port q_1 . Likewise, the uncertainty of c can be removed by introducing non-determinism in C' , such that C' can at any moment either accept an input from port p_0 or accept an input from port p_1 .

In the following, let us assume for simplicity that the differences between the arrival times on the inputs of both A' and C' are large enough compared to the execution times of the underlying tasks, such that iterations of these tasks cannot be delayed by preceding iterations. Moreover, let us only focus on the temporal aspects of the different relations, by ignoring any associated values. And finally, let us assume that if A produces an output for index i on p_0 , it produces an output with timestamp 0 on p_1 , and vice versa. We define an output with timestamp 0 as “no output”, which is to ensure that corresponding indices can be maintained between abstraction layers. Given this, we can express the relation of A' as follows:

$$R_{A'} = \{(X, Y) \in Tr(P_A) \times Tr(Q_A) \mid \forall_i: (k = 0 \vee k = 1) \wedge \tau_{Y[q_{1-k}]}(i) = 0 \wedge \tau_{Y[q_k]}(i) = \tau_{X[p]}(i) + \chi_a(i)\}$$

The parameter k is used to express the non-determinism of A' : On $k = 0$ for a certain index i the respective output is produced on port q_0 and no output (timestamp of 0) on port q_1 , whereas on $k = 1$ the output is produced on port q_1 and no output on q_0 . We further define the relations of B'_n (with $n \in \{0, 1\}$) as follows:

$$R_{B'_n} = \{(x, y) \in St(p_{bn}) \times St(q_{bn}) \mid \forall_i: \tau_y(i) = \max_{i' < i}(\tau_x(i'), \tau_y(i')) + \chi_b(i)\}$$

These components are deterministic as their outputs are unambiguously defined by their inputs. The formulation with *max* takes care that an iteration of a task execution cannot start before the previous ones are finished. Lastly, we define the relation of component C' :

$$R_{C'} = \{(X, Y) \in Tr(P_{C'}) \times Tr(Q_{C'}) \mid \forall_i: (k = 0 \vee k = 1) \wedge \tau_{Y[q]}(i) = \tau_{X[p_k]}(i) + \chi_c(i)\}$$

Here, the parameter k models the non-determinism of C' : On $k = 0$ for a certain index i the input on port p_0 is consumed by C' , which is reflected by the dependence of the output timestamp $\tau_{Y[q]}(i)$ on the input timestamp $\tau_{X[p_0]}(i)$, whereas on $k = 1$ the input on port p_1 is consumed.

The graph G' is apparently non-deterministic because both A' and C' are non-deterministic. We can now try to remove this non-determinism using component worst-case bounding. For component A' we can derive a deterministic worst-case bound A if we replace the choice on which port outputs are produced with a production on both ports, i.e.:

$$R_A = \{(X, Y) \in Tr(P_A) \times Tr(Q_A) \mid \forall_i: \tau_{Y[q_0]}(i) = \tau_{Y[q_1]}(i) = \tau_{X[p]}(i) + \chi_a(i)\}$$

It is easy to see that A is deterministic and a valid worst-case bound on A' , i.e. $A \triangleright A'$. The components B'_0 and B'_1 are already deterministic and it holds that $B'_0 \triangleright B'_0$, as well as $B'_1 \triangleright B'_1$. Consequently, we can leave these components unchanged. Lastly, we can bound component C' with a deterministic component C that does not use the arrival times of data on either of the two inputs, but the maximum of both, i.e.:

$$R_C = \{(X, Y) \in Tr(P_C) \times Tr(Q_C) \mid \forall_i: \tau_{Y[q]}(i) = \max(\tau_{X[p_0]}(i), \tau_{X[p_1]}(i)) + \chi_c(i)\}$$

Also this component is apparently deterministic and a valid worst-case bound on C' , i.e. $C \triangleright C'$. Because the input acceptance of the components is not changed between layers (the components are all input-complete), we can finally assess that graph G is a deterministic and valid worst-case bounding abstraction of G' .

6.5 REMOVING DEPENDENCIES

Figure 6.5: Removal of non-monotonic dependencies between task components.

Another key feature of our theory is the possibility to remove and add dependencies between different components on abstraction and refinement. With the

usage of partial inclusion and partial bounding it is enabled to add and remove connections between components, as long as inclusion or bounding are preserved despite the addition or removal. In the following we present an example on how a dependency removal can be realized in practice.

Consider the task components A' and B' depicted in Figure 6.5. This use-case resembles the execution of two tasks on the same processor, being scheduled by an SPP scheduler [But11] and task A' having a lower priority than B' . For instance, the relations of the two components could be defined as follows (with the functional parts of the relations being pruned for simplicity):

$$\begin{aligned}
 R_{A'} &= \{(X, Y) \in Tr(P_{A'}) \times Tr(Q_{A'}) \mid \forall i: \\
 &\quad \varepsilon_{A'}(i) = \max(\tau_{X[p_A]}(i), f_{A'}(i-1)) \wedge \\
 &\quad f_{A'}(i) = \tau_{Y[q_A]}(i) = \varepsilon_{A'}(i) + \chi_A + \mathcal{I}(\varepsilon_{A'}(i), f_{A'}(i), X[p_\varepsilon], X[p_f])\} \\
 R_{B'} &= \{(X, Y) \in Tr(P_{B'}) \times Tr(Q_{B'}) \mid \forall i: \\
 &\quad \tau_{X[p_B]}(i) = i \cdot P + J_B(i) + c \text{ with } 0 \leq J_B(i) \leq \hat{J}_B, c \geq 0 \wedge \\
 &\quad \varepsilon_{B'}(i) = \tau_{Y[q_\varepsilon]}(i) = \max(\tau_{X[p_B]}(i), f_{B'}(i-1)) \wedge \\
 &\quad f_{B'}(i) = \tau_{Y[q_f]}(i) = \tau_{Y[q_B]}(i) = \varepsilon_{B'}(i) + \chi_B\}
 \end{aligned}$$

The second line of $R_{B'}$ ensures that only periodic input streams with a period of P and a maximum jitter of $\hat{J}_B$ are accepted. The enabling times $\varepsilon_{A'}(i)$ and $\varepsilon_{B'}(i)$ indicate the moments in time at which A' and B' begin their executions, whereas the finish times $f_{A'}(i)$ and $f_{B'}(i)$ indicate the moments at which the respective executions finish. The different maximum formulations are used to prevent that the tasks begin executions before their previous iterations are finished. As the time between enablings and finishes of executions of A' are determined by the interference of B' (or, in other words, the remaining service characterized by $Y[q_\varepsilon]$ and $Y[q_f]$), it is required to include this interference in the finish times of A' . For that matter, the interference function $\mathcal{I}$ is included as an additive term. The function takes the enabling and finish times of A' , $\varepsilon_{A'}(i)$ and $f_{A'}(i)$, into account, alongside the interference characterization of B' and returns the total interference time between enabling and finish of an iteration i of A' .

Considering such dependencies in an analysis is quite challenging, especially because such dependencies can cause scheduling anomalies, which are basically non-monotonic effects. For instance, later enabling times of task B' can result in earlier finishes of A' as in that case A' may obtain more remaining service. To enable an efficient analysis it is therefore desirable to remove such dependencies without a significant loss in accuracy.

With our theory it is fairly simple to remove the resource dependencies between A' and B' using bounding abstraction. Consider the graph G depicted in Figure 6.5. Compared to the graph G' the resource dependencies including connections of ports q_ε and p_ε , as well as q_f and p_f are removed. Given this removal,

let component B have the following relation:

$$R_B = \{(X, Y) \in Tr(P_B) \times Tr(Q_B) \mid \forall_i: \\ f_B(i) = \tau_{Y[q_B]}(i) = \max(\tau_{X[p_B]}(i), f_B(i-1)) + \chi_B\}$$

It can be seen that B with this relation is a valid worst-case bound on component B' , i.e. $B \triangleright B'$. But with respect to A , the situation is more difficult. The problem is that A without resource dependencies has to be a valid worst-case bound on A' for all cases, i.e. for all potential interference characterizations of B' , independent of what the actual interference of B' is. In principle also the case of B' not offering any remaining service to A' can occur, if e.g. $\tau_{X[p_\varepsilon]}(0) = 0$ and $\forall_i: \tau_{X[p_\varepsilon]}(i+1) = \tau_{X[p_f]}(i)$. In this case the interference function $\mathcal{I}$ becomes infinite for all i , such that A' produces an empty stream on port $q_{A'}$ for all inputs. This implies that A can only be a valid worst-case bound on A' if it produces the empty stream for all inputs, i.e.:

$$R_A = \{(X, Y) \in Tr(P_A) \times Tr(Q_A) \mid Y[q_A] = y^\infty\}$$

Given this relation it of course holds that $A \triangleright A'$, but a component A that only produces empty streams would be rather useless.

To address this problem we can apply the following trick: First we replace component A' with a component A'_{red} that does not accept all streams on ports p_ε and p_f , but only a subset which includes all streams that can be produced by B' . Assuming that $\hat{J}_B + \chi_B \leq P$ it can be seen that it must hold for $Y[q_\varepsilon]$ and $Y[q_f]$:

$$\forall_i: i \cdot P + c \leq \tau_{Y[q_\varepsilon]}(i) \leq i \cdot P + \hat{J}_B + c \wedge \\ i \cdot P + \chi_B + c \leq \tau_{Y[q_f]}(i) \leq i \cdot P + \chi_B + \hat{J}_B + c$$

These intervals allow us to redefine A' as A'_{red} as follows:

$$R_{A'_{red}} = \{(X, Y) \in Tr(P_{A'}) \times Tr(Q_{A'}) \mid \forall_i: \\ i \cdot P + c' \leq \tau_{X[p_\varepsilon]}(i) \leq i \cdot P + \hat{J}_B + c' \wedge \\ i \cdot P + \chi_B + c' \leq \tau_{X[p_f]}(i) \leq i \cdot P + \chi_B + \hat{J}_B + c' \text{ with } c' \geq 0 \wedge \\ \varepsilon_{A'}(i) = \max(\tau_{X[p_A]}(i), f_{A'}(i-1)) \wedge \\ f_{A'}(i) = \tau_{Y[q_A]}(i) = \varepsilon_{A'}(i) + \chi_A + \mathcal{I}(\varepsilon_{A'}(i), f_{A'}(i), X[p_\varepsilon], X[p_f])\}$$

As the component A'_{red} has the same behavior as A' and does not restrict the outputs of B' in any way it can be seen that the graph G'_{red} is equivalent to the graph G' .

The advantage of using A'_{red} is that we can now bound it by a component A without resource dependencies that does not necessarily produce the empty

stream. Consider the following relation of A :

$$R_A = \{(X, Y) \in Tr(P_A) \times Tr(Q_A) \mid \forall_i : \\ \varepsilon_A(i) = \max(\tau_{X[p_A]}(i), f_a(i-1)) \wedge \\ f_A(i) = \tau_{Y[q_A]}(i) = \varepsilon_A(i) + \chi_A + \left\lceil \frac{f_a(i) - \varepsilon_a(i) + \hat{J}_B}{P} \right\rceil \cdot \chi_B\}$$

It can be seen that a component A with above relation worst-case bounds A'_{red} , i.e. $A \triangleright A'_{red}$ (for more information on the used interference bound please refer to Chapter 10), which means together with $B \triangleright B'$ and the preserved input acceptance that the graph G is a valid worst-case abstraction of G' , despite the removed resource dependencies. At the same time, A can produce output streams other than the empty stream and it holds that both A and B are monotone, which enables an efficient and useful analysis.

Note that graph G only worst-case bounds the temporal behavior of G' , i.e. G' has a higher accuracy than G . This is due to the fact that we have removed the explicit resource dependency and replaced it with an interval determined by P and $\hat{J}_B$. Finding such characterizations of resource dependencies that are as accurate as possible and at the same time do not break monotonicity is subject of the second part of this thesis.

6.6 PRACTICAL EXAMPLE

In the following we apply our theory on a more practical example. We consider a simple graph of two components of which one is to be replaced by a faster one. According to TETB [GTW11, HB16] this replacement would be invalid as the faster component has a more restricted input acceptance than the original one. According to our theory this replacement would be only valid if the graph after replacement were a worst-case refinement of the graph before. To prove refinement we have to show bounding between the components individually and input acceptance preservation between the whole graphs. Proving the latter is not trivial for this example. Nevertheless, we show that input acceptance preservation can be proven by means of both best-case and worst-case models.

Consider the graph G depicted on the left side of Figure 6.6. This graph depicts a part of a control loop which consists of two components, an estimator task that is laid out to operate on strictly periodic in-order input streams with a frequency equal to 100kHz and a controller task that is laid out in such way that it is input-complete with respect to in-order streams. It is determined that the estimator task takes between 1 μ s and 3 μ s to execute, whereas the controller task executes between 2 μ s and 4 μ s. Both tasks are executed on separate processors. Ignoring the functional behavior of the tasks, their temporal behavior can be expressed using the following relations (with $\tau_y(-1) = 0$):

Figure 6.6: Replacement of a slow input-complete controller implementation ctI by a faster controller ctI' with restricted input acceptance.

$$\begin{aligned}
 R_{est} &= \{(x, y) \in St(p_{est}) \times St(q_{est}) \mid c \geq 0 \wedge \forall_{i < |x|}: \tau_x(i) = i \cdot 10\mu s + c \wedge \\
 &\quad \forall_i: 1\mu s \leq \chi(i) \leq 3\mu s \wedge \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + \chi(i)\} \\
 R_{ctI} &= \{(x, y) \in St(p_{ctI}) \times St(q_{ctI}) \mid \\
 &\quad \forall_i: 2\mu s \leq \chi(i) \leq 4\mu s \wedge \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + \chi(i)\}
 \end{aligned}$$

To reduce the end-to-end latency of the two tasks and thereby improve control behavior we attempt to replace the controller task component ctI with a faster component ctI' that executes only between $2\mu s$ and $3\mu s$, but that requires its input stream to be periodic with a frequency of $100kHz$ and a maximum jitter of $3\mu s$. The temporal behavior of ctI' is captured by the following relation:

$$\begin{aligned}
 R_{ctI'} &= \{(x, y) \in St(p_{ctI}) \times St(q_{ctI}) \mid c' \geq 0 \wedge \\
 &\quad \forall_{i < |x|}: i \cdot 10\mu s + c' \leq \tau_x(i) \leq i \cdot 10\mu s + c' + 3\mu s \wedge \\
 &\quad \forall_i: 2\mu s \leq \chi(i) \leq 3\mu s \wedge \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + \chi(i)\}
 \end{aligned}$$

We have to show that the graph G' in the middle of Figure 6.6 is a valid worst-case refinement of G , as only then it is guaranteed that G' accepts all input streams with a frequency of $100kHz$ and that for these inputs G' is never slower than G . According to Theorem 5.5 we must thus show that $G' \triangleleft G$ and $in_{G'} \supseteq in_G$.

For $G' \triangleleft G$ it must hold that $est \triangleleft est$, which is trivial for this example, and that $ctI' \triangleleft ctI$, which holds due to the following reasoning: Consider two input streams $x' \in in_{ctI'}$ and $x \in in_{ctI}$ with $x' \triangleleft x$. Using the relations of the respective components it follows for any corresponding output streams y' of ctI' that $\forall_i: \tau_{y'}(i) \leq \max(\tau_{x'}(i), \tau_{y'}(i-1)) + 3\mu s$, whereas ctI has an output stream y with $\forall_i: \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + 4\mu s$. From $x' \triangleleft x$ it follows that $\tau_{y'}(0) \leq \tau_y(0)$, from this that also $\tau_{y'}(1) \leq \tau_y(1)$, and so on. Thus we can conclude that $y' \triangleleft y$, which satisfies Definition 42, and it follows $ctI' \triangleleft ctI$ and thus $G' \triangleleft G$.

It remains to be proven that $in_{G'} \supseteq in_G$. If it held that $in_{ctI'} \supseteq in_{ctI}$ this proof would be trivial. But ctI is input-complete with respect to in-order streams and ctI' is not (which prevents a usage of TETB for this example). Consequently we need to determine both in_G and $in_{G'}$. For graph G it holds that $in_G = in_{est}$, as ctI is input-complete for in-order streams and as no reordering takes place in

est. For graph G' it can be seen that also $in_{G'} = in_{est}$ if $out_{est} \subseteq in_{ctl'}$. From the relation $R_{ctl'}$ we determine:

$$in_{ctl'} = \{x \in St(p_{ctl}) \mid c' \geq 0 \wedge \forall_{i < |x|}: i \cdot 10\mu s + c' \leq \tau_x(i) \leq i \cdot 10\mu s + c' + 3\mu s\}$$

Deducing out_{est} is not straightforward. However, a superset of out_{est} can be determined rather easily by constructing two analysis models that bound the temporal behavior of G' from above and below. For these models we make use of deterministic HSDF actors, as depicted on the right side of Figure 6.6. Considering the semantics of HSDF actors [SB09a], it follows with Corollaries 4.7 and 4.8 for the relation $R_{est^{wc}}$ (here the self-edge token component is already feedback-composed with the HSDF actor component):

$$R_{est^{wc}} = \{(x, y) \in St(p_{est}) \times St(q_{est}) \mid \forall_i: \tau_y(i) = \max(\tau_x(i), \tau_y(i-1)) + 3\mu s\}$$

The other relations $R_{est^{bc}}$, $R_{ctl'^{wc}}$ and $R_{ctl'^{bc}}$ are all of similar forms, only the so-called firing durations are not $3\mu s$, but $1\mu s$, $3\mu s$ and $2\mu s$, respectively. By construction it holds that $est^{bc} \trianglelefteq est \trianglelefteq est^{wc}$, $ctl'^{bc} \trianglelefteq ctl' \trianglelefteq ctl'^{wc}$, and thus also $G'^{BC} \trianglelefteq G' \trianglelefteq G'^{WC}$. With the input-completeness of the analysis models it further follows that G'^{BC} is a valid best-case and G'^{WC} is a valid worst-case abstraction of G . These dataflow abstractions allow to efficiently compute schedules that bound the temporal behavior of *est* [8, HWGB13]. For any output stream y of *est* we can conclude from the best-case model that $\forall_{i < |y|}: i \cdot 10\mu s + c + 1\mu s \leq \tau_y(i)$ and from the worst-case model that $\forall_{i < |y|}: \tau_y(i) \leq i \cdot 10\mu s + c + 3\mu s$. From that we obtain:

$$out_{est} \subseteq \{y \in St(q_{est}) \mid c \geq 0 \wedge \forall_{i < |y|}: i \cdot 10\mu s + c + 1\mu s \leq \tau_y(i) \leq i \cdot 10\mu s + c + 3\mu s\}$$

By setting c' in $in_{ctl'}$ to $c + 1\mu s$ it can be seen that all streams in out_{est} are also included in $in_{ctl'}$, i.e. $out_{est} \subseteq in_{ctl'}$, and thus also $in_{G'} = in_{est} \supseteq in_G$. This concludes the proof that G' is a valid worst-case refinement of G .

chapter 7

Conclusion

In the first part of this thesis we have presented a generic timed component model and an abstraction-refinement theory for asynchronous discrete-event systems. The theory supports temporal and functional exclusion, inclusion and bounding abstraction and refinement, which enables the usage of efficiently analyzable models for the analysis and design of non-deterministic, non-monotone real-time systems. We have discussed that properties like best-case and worst-case bounding are automatically lifted from component to graph level, enabling a component-based reasoning.

Unlike the TETB theory, our theory separates the preservation of component input acceptance from exclusion, inclusion or bounding. This separation even-handedly enables abstraction, which preserves input acceptance from implementations to models, and refinement, which preserves input acceptance from models to implementations. Consequently, our theory is an abstraction-refinement theory that equally supports model-based design and model-based analysis, whereas related works are mainly refinement theories, with a limited applicability for analysis purposes. Furthermore, we have proven that the separation of input acceptance preservation and exclusion, inclusion or bounding enables the automatic lifting of the latter from component to graph level without a restriction to input-complete components, resulting in a significantly larger applicability compared to TETB. Finally, we have introduced both worst-case and best-case bounding relations, such that our theory does not only allow for the usage of worst-case models, but also of best-case models, which are both needed to analyze applications in which jitter plays a major role.

In a case study, we have discussed the extended applicability of our theory with respect to TETB on several examples. These included a simplified handling of reordering, an abstraction of input-restricted implementations to input-complete analysis models, value bounding, the removals of non-determinism, uncertainty and dependencies by means of abstraction, as well as a practical example in

which input-complete components were replaced by faster, but input-restricted components.

In Appendix A an extension of our theory is presented. The extension raises the scope from discrete-event systems to mixed discrete-event and continuous time systems, which enables abstraction and refinement in the context of CPS's. Showing the applicability and relevance of this extension is subject to future work.

PART II

Evaluation of Concurrent Real-Time Systems

chapter 8

Introduction

We have already discussed in Chapter 1 that the safety-criticality of real-time systems such as SDRs imposes the need to give both temporal and functional guarantees at design time. As particularly challenging we have asserted the subclass of concurrent real-time systems with shared resources and SPP scheduling, due to the cyclic data and cyclic resource dependencies that can occur in such applications. Giving guarantees on such applications is further complicated by the SPP scheduling that can cause scheduling anomalies leading to temporal non-monotonicity (i.e. later events can cause earlier events), as well as by uncertainties with respect to the accurate timestamps of events.

We assume FIFO buffer communication between all real-time system tasks and differ between two types of buffers: For one we consider buffers with blocking writes, which means that not only a reading task has to wait for the production of values if the buffer is empty, but that also a writing task has to wait for the consumption of values if the buffer is full. In consequence, the usage of such buffers introduces additional cyclic data dependencies, such that even inherently acyclic real-time systems, i.e. applications without feedback loops, become cyclic. Moreover, we also consider FIFO buffers with non-blocking writes in which values are overwritten by a writing task if they are not timely consumed by the respective reading task. For such buffers the applications do not become cyclic. However, we need to take special care that sufficient capacities are supplied for such buffers, as an overflow does not only affect temporal behavior, but can even result in functional errors.

To resolve these complications we have proposed a two-step analysis procedure: First, suitable, i.e. sufficiently, but not too entropic analysis models must be devised by means of abstraction, using the methods described in the first part of this thesis. Second, these analysis models must be analyzed with suitable, i.e. effective and efficient, analysis techniques. And as the best analysis model is worth nothing without a good analysis technique, we devote the better part of the remainder of this thesis to the construction of suitable analysis techniques.

Regarding the data dependencies of real-time systems it has been shown that dataflow analysis techniques are capable of efficiently determining temporal guarantees [SB09a, SGB06, WBS07b], thanks to their inherent support for cyclic data dependencies. Note that the applicability of dataflow analysis techniques is not limited to temporal analysis, but also includes the computation of required buffer capacities [WBS08], scheduler settings [WBGB10], a suitable task-to-processor assignment [SBGC07] and forms the basis for synchronization overhead minimization techniques such as task clustering [F⁺08] and resynchronization [HBC11].

For the analysis of resource dependencies in the context of SPP scheduling a variety of classical real-time analysis techniques [TBW94, H⁺05, WTVL06, GHGGPGDM01, K⁺13] has been proposed. However, none of these techniques is capable of considering the complex resource dependencies that occur due to SPP scheduling in combination with cyclic data dependencies, especially if the resource dependencies are also cyclic, i.e. if the resource dependencies are opposed to the flow of data. The classical real-time analysis approaches can therefore neither handle applications with feedback loops, nor applications that use blocking write buffers for communication between tasks.

To resolve these issues, an approach that combines classical real-time analysis [TBW94] for resource dependencies with dataflow analysis for data dependencies has been presented in [HWGB13]. Albeit not made explicit, this approach follows our proposed two-step scheme in that different analysis models are derived from the application model and thereafter analyzed using the respectively suitable techniques. Unfortunately, however, the analysis of resource dependencies is rather coarse, such that the approach leaves plenty of room for improvement with respect to analysis accuracy. Consequently, the first two analysis techniques that we propose in Chapters 10 and 11 are directly based on [HWGB13] and aim to increase analysis accuracy significantly without losing too much in terms of complexity.

Following the terminology devised in Section 2.2, the most significant improvement in effectiveness and therewith accuracy can be attributed to the explicit consideration of cyclic data dependencies in the analysis of resource dependencies, as introduced in Chapter 10: We show that cyclic data dependencies that occur due to feedback loops, but also due to the usage of blocking write buffers, effectively limit interference between tasks executing on shared processors. Therefore we can use such dependencies to increase analysis accuracy, as well as even for the optimization of applications to meet certain throughput or latency constraints. For the latter we propose a technique called iterative buffer sizing. Note that the usage of cycles to limit interference implies that the “*classical*” assumption of monotonicity between time and space is broken, in that it no longer holds that more buffer space always results in better or equal temporal properties. On the contrary, we show that restricting buffer capacities can result in superior temporal properties at lowered space requirements, a win-win situation.

An additional accuracy improvement can be realized by replacing the rather crude period-and-jitter interference characterization by a more sophisticated

execution interval technique that is similar to dynamic offsets, as proposed in Chapter 11. We show that the technique can be further enhanced by combining execution intervals with an explicit consideration of precedence constraints, as well as by the aforementioned iterative buffer sizing.

While the support of cyclic data and resource dependencies means a significant leap forward in terms of applicability, both period-and-jitter and execution interval approaches are limited to so-called single-rate and single-phase applications. In such applications all tasks consist of one phase and can only produce or consume exactly one value per buffer and execution. Moreover, the two approaches require atomic accesses of all connected buffers in each task execution, as only then it is ensured that the determined analysis results serve as guarantees for the respective implementations.

In the third technique proposed in Chapter 12 we consequently do not only aim at improving analysis accuracy, but mainly at increasing applicability. This is achieved by introducing support for multi-phase and multi-rate applications. In this context we construct a multi-phase analysis technique for resource dependencies that allows to analyze multi-phase applications at the same level of effectiveness as the simpler single-phase ones. In addition, we discuss that the multi-phase approach can be easily extended to a multi-resource approach, i.e. an analysis approach that can be used to accurately analyze applications whose temporal behavior is not only impacted by processor sharing, but also by e.g. shared buses or shared memories.

Finally, there are situations for which analysis alone is not the be-all and end-all to the evaluation of real-time systems at design time. In many cases it is not only relevant to determine worst-case, but also average-case performance. On top of that, the construction of analysis models is not always trivial, hence we would like a means to gain confidence in our models. To address both these issues we discuss a simulation approach in Chapter 13 that is nested in the same modeling framework as used for analysis.

The remainder of this introductory chapter is structured as follows: In Section 8.1 we present the basic ideas of deriving analysis models from application models, of the analysis techniques applied on these models, as well as of our simulation approach. Section 8.2 discusses related work. In Section 8.3 we highlight important differences and extensions of the paper versions on which the content of the remainder of this thesis is based on.

8.1 BASIC IDEAS

In this section we informally describe the derivation of our analysis models from application models, the basic ideas of our different analysis techniques, as well as of our simulation approach.

8.1.1 CONCURRENT REAL-TIME SYSTEMS WITH SHARED RESOURCES

We consider real-time systems that consist of one or more so-called **task graphs**. Each task graph consists of a single **strictly periodic source**, as well as one or more **tasks**. Communication between tasks is realized with **FIFO buffers** that are assumed as ideal, such that communication time is attributed to the connected tasks. Each FIFO buffer connects exactly one producing to exactly one consuming task, while tasks can be connected to multiple FIFO buffers. The buffers can be both **blocking or non-blocking on writes**. Each buffer has a certain number of initially full and initially empty locations, with their sum defining the buffer capacity.

A FIFO buffer has the same temporal behavior as a pair of token components as described in Section 4.7. This analogy is depicted in Figure 8.1, with the number of initially full locations corresponding to the number of tokens on the forward edge and the number of initially empty locations to the number of tokens on the backward edge. For better readability, we often make use of such token edge pairs to depict FIFO buffers in task graphs in the following.

Figure 8.1: Expressing FIFO buffers and self-delay of tasks using dataflow edges with initial tokens.

Besides such **data dependencies** defined by FIFO buffers we also allow for **resource dependencies** that are explicitly not limited to single task graphs, but that can also connect tasks of different task graphs. The resource dependencies consist of streams of enabling and finish times of the interfering tasks and as such model interference between tasks that occurs due to resource sharing, e.g. processor sharing. For an **SPP scheduler**, for instance, the resource dependencies go from all higher priority tasks to all lower priority tasks on the same processor.

For the tasks we consider two different types, **single-rate, single-phase tasks** on the one hand and **multi-rate, multi-phase tasks** on the other. Single-rate, single-phase tasks consume per execution exactly one value from each of their input buffers and produce exactly one value to each of their output buffers. In contrast, multi-rate, multi-phase tasks consist of multiple phases and each phase can consume and produce multiple values (even zero).

Buffer accesses always consist of three phases, the acquisition of data or space, the reading or writing of data values to the acquired locations, as well as their release. We assume that all acquisitions occur atomically at the beginnings of task (phase) executions and all releases atomically at their ends.

We define that tasks or task phases are **externally enabled** as soon as the required numbers of full locations (which are one for single-rate tasks) are available in all input buffers. For blocking write buffers, additionally the required

number of empty locations must be available in all output buffers. We say that a task is not only externally enabled, but just **enabled** as soon as also all preceding task (phase) executions of the same task have finished. In dataflow terms, such a self-delay can be modeled by adding a self-edge with one token to the tasks, as highlighted in Figure 8.1. Note that in the following we always assume self-edges of tasks as implicit (except for clarification purposes such as in Figure 9.2), while we make self-edges of dataflow actors (or their absence) always explicit.

We assume a data-driven behavior of tasks, such that task (phase) **executions immediately begin at enablings**. After the atomic acquisition of full and empty buffer locations at the beginning of a task (phase) execution, the execution takes a certain execution time between the Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET) of the task (phase). The finish time of an execution can be additionally delayed by interference due to resource sharing, such as interference from higher priority tasks for the case of SPP scheduling. The atomic release of both empty and full buffer locations **finishes** a task (phase) execution.

Note that this application model is already an exclusion abstraction of reality as behaviors that can occur in reality, such as the failure of tasks, are excluded. Moreover, it is also an inclusion abstraction of reality as we do not consider the actual execution times of tasks, but ranges between BCETs and WCETs.

8.1.2 TEMPORAL ANALYSIS MODELS & FLOWS

Our goal is to analyze the temporal behavior of concurrent real-time systems that are modeled according to the application model described in the previous section. For instance, we want to determine throughput or latency guarantees for such applications. To give such guarantees we need to determine lower and upper bounds on the enabling and finish times of task executions.

Analyzing applications that make use of processor sharing with SPP scheduling comes with several problems, however. One issue is that SPP schedulers belong to the class of so-called **non-starvation-free schedulers** [WBS09] for which the interference of higher priority tasks on lower priority ones can only be bounded if enabling rate characterizations of the tasks are given, i.e. it is not sufficient to know the WCETs of higher priority tasks, but also bounds on their enabling times must be known. These bounds are in turn dependent on the interference of other tasks, which is the reason that all analysis approaches presented in the following are iterative.

Moreover, SPP scheduling can introduce **scheduling anomalies**, such that e.g. a later enabling of a higher priority task can result in less interference and thus earlier finishes of lower priority tasks. Such non-monotonicities prevent us from directly analyzing resource dependencies. Instead, we express application models in the timed component model introduced in Chapter 4 and based on that derive several analysis models by means of inclusion and bounding abstraction that do not contain resource dependencies anymore.

In total, we make use of **four different analysis models** in each of our analysis approaches. The different analysis models and their relation to the analyzed application model are depicted in Figure 8.2.

Figure 8.2: Application model and abstract analysis models (The parameters determined with the analysis models are denoted inside the white boxes, the dependencies between parameters are marked with gray arrows).

In a first step we construct a so-called **response time analysis model** that is essentially the same as the underlying application model, but that adds constraining intervals to all input ports of all task components. For each considered task, these input intervals are defined by lower and upper bounds on the enabling and finish times of higher priority task executions for the resource dependencies, as well as by lower and upper bounds on the external enabling times of the task itself for the data dependencies. The former intervals effectively restrict the amount of interference that can be imposed by higher priority tasks, while the latter intervals restrict the acquisition times on the connected buffers. Obviously, a response time analysis model with such input intervals is only then a valid inclusion abstraction of the application model if the intervals are chosen wide enough such that the actual streams coming from higher priority tasks and connected buffers are not restricted. In the following we call this condition the **schedule criterion**. Hence the inclusion is conditional in the validity of the schedule criterion, as indicated with (a) in Figure 8.2.

Using the input intervals we can locally (i.e. without considering other tasks and the actual dependencies between them) determine so-called maximum response times of tasks ($\hat{R}$ in Figure 8.2), which we define as differences between maximum external enabling and maximum finish times of task executions. By construction, these maximum response times are always valid for the response time analysis model and, given that the input intervals are chosen wide enough (i.e. given that the schedule criterion (a) holds), also valid bounds on the actual response times of tasks in the application model.

Furthermore, we define a **best-case analysis model** which is a best-case bounding abstraction of the response time analysis model. In this model we remove all resource dependencies, as well as all edges with initial tokens, including

the self-edges of tasks. Due to this removal, the best-case analysis model becomes an HSDF model for single-phase, single-rate applications and a CSDF model for multi-rate, multi-phase ones. The firing durations of the resulting dataflow actors are set to the respective BCETs of the corresponding tasks. As such, the best-case analysis model lower-bounds the response time analysis model if all firing durations are lower bounds on the respective task execution times. The latter condition defines the first part of the so-called **firing duration criterion** denoted as (b) in Figure 8.2.

Unlike the other models discussed so far, the best-case analysis model is deterministic, which enables a rather simple holistic analysis to determine lower bounds on the schedules of tasks, i.e. lower bounds on the (external) enabling and finish times of task executions ($\hat{e}^{ext}$, $\hat{e}$ and $\hat{f}$ in Figure 8.2). We use these lower bounds as the respective lower bounds of the input intervals in the response time analysis model.

Also based on the response time analysis model we derive a **worst-case analysis model** by means of worst-case bounding abstraction. Here we remove all resource dependencies and self-edges, but keep all other data dependencies, i.e. all edges without and also with initial tokens. Consequently, also the worst-case analysis model becomes an HSDF or CSDF model. The firing durations of the resulting actors are set to the maximum response times derived from the response time analysis model. The worst-case analysis model is only a valid worst-case bounding abstraction of the response time analysis model if the maximum response times used as firing durations compensate for the removed dependencies, i.e. are upper bounds on execution times, self-delay and higher priority interference. This condition represents the other part of the **firing duration criterion** marked as (b) in Figure 8.2.

We use the worst-case analysis model, which is also deterministic, to determine upper bounds on the schedules of tasks, i.e. upper bounds on the (external) enabling and finish times of task executions ($\hat{e}^{ext}$, $\hat{e}$ and $\hat{f}$ in Figure 8.2), via holistic analysis. Analogous to the lower bounds obtained from the best-case model, we use these upper bounds as the respective upper bounds of the input intervals in the response time analysis model.

Note that the maximum response times computed in the response time analysis model depend on the upper bounds on enabling and finish times derived from the worst-case analysis model, while the upper bounds on enabling and finish times depend on the maximum response times, as depicted in Figure 8.2. This interdependence motivates the usage of the **iterative analysis flow** presented in Figure 8.3.

All analysis approaches that we present in the following are based on this flow, which essentially works as follows: Initially, we set the firing durations in the worst-case analysis model to the WCETs of the respective tasks. Using these firing durations we compute upper bounds on schedules of tasks, that we use in the response time analysis model for the upper bounds of the input intervals, alongside the lower bounds obtained from the best-case analysis model. Based on

Figure 8.3: Primitive analysis flow.

the input intervals we compute maximum response times that we insert as firing durations in the worst-case analysis model. With these new firing durations we recompute the upper bounds on task schedules, and so on. This procedure is repeated until either a constraint violation occurs or until all maximum response times (and therewith also all other parameters) have converged. Due to the monotonicity of the different steps we can guarantee termination of the analysis flow.

We prove that on convergence the input intervals are large enough such that the schedule criterion (a) in Figure 8.2 holds, i.e. the response time analysis model becomes a valid inclusion abstraction of the application model. Likewise, we show that on convergence the maximum response times are large enough such that the firing duration criterion (b) holds as well, i.e. the worst-case analysis model is a valid worst-case bounding abstraction of the response time analysis model and thereby also of the application model. Consequently, all bounds on (external) enabling, finish and response times are not only valid in the respective analysis models, but are also valid bounds on the actual (external) enabling, finish and response times of tasks in the application model. This ensures that any analysis results derived from these parameters, such as throughput or latency guarantees, are valid for the underlying application as well.

Finally, note that while it is rather simple to derive maximum response times using input intervals only, this method usually also comes with a significant loss in analysis accuracy. To improve accuracy we introduce another method to limit interference, which is based on an explicit consideration of precedence constraints derived from data dependencies between tasks. For the analysis of precedence constraints (ζ in Figure 8.2) we introduce our last analysis model, the **precedence constraint analysis model**. This model differs from the response time analysis model only in that it does not contain any resource dependencies, such that it also becomes an HSDF or CSDF model, depending on the type of the application model. As the removal of dependencies cannot reduce the number of behaviors (i.e. the pairs of input and output traces) of any components it follows that the precedence constraint analysis model is an inclusion abstraction of the response time analysis model, as depicted in Figure 8.2.

8.1.3 PERIOD-AND-JITTER ANALYSIS

The first analysis technique we consider is the period-and-jitter analysis, whose name is coined on the type of interference characterization used in this approach. The technique is capable of analyzing concurrent real-time systems with SPP scheduling of tasks on shared processors and with all tasks being both single-rate and single-phase, such that the respective best-case, worst-case and precedence constraint analysis models become HSDF graphs.

Our period-and-jitter analysis technique is inspired by [HWGB13]. The fundamental difference between our technique and the latter is that we additionally use cyclic data dependencies to limit higher priority interference, which results in a significant improvement in effectiveness and therewith also accuracy.

For the schedule bounding part of our iterative algorithm according to Figure 8.3 we use the best-case and worst-case analysis models derived according to the previous section. Using a Linear Program (LP) applied on the best-case HSDF graph that uses BCETs as firing durations we determine **best-case start times** of actors, which by construction form periodic lower bounds on the enabling times of tasks. Likewise we can compute **worst-case start times** of actors by applying an LP on the worst-case HSDF graph, initially using WCETs as firing durations and in later iterations of the flow the maximum response times derived from response time analysis. Due to the lack of self-edges, the determined start times are not periodic upper bounds on the enabling times, but only on the external enabling times of tasks, i.e. enabling times without consideration of self-delay. Nevertheless, we can use these bounds to also determine upper bounds on total task enabling times, as well as task finish times. Both lower and upper bounds on (external) enabling and finish times are used in the response time analysis model as input intervals.

In the response time analysis model we derive upper bounds on the jitters of higher priority tasks by computing differences between the enabling times from the respective input intervals. We use such **maximum jitters** of higher priority tasks in conjunction with their periods to determine upper bounds on the number of higher priority task executions per timespan. With such **period-and-jitter interference characterizations**, the WCET of the task under observation and the WCETs of all higher priority tasks we could compute maximum response times that represent upper bounds on the execution times of the observed task, as well as all higher priority interference that could occur during its executions.

However, according to the requirements of the worst-case analysis model that does not contain self-edges, our maximum response times also need to include any possible self-delay. Remember that each execution of a task t_a is either enabled externally by buffer releases issued by connected tasks or internally enabled by preceding executions of task t_a . To cover all cases, we consequently compute so-called **q -level maximum busy periods** over $q + 1$ executions, of which only the first is considered externally enabled at its maximum external enabling time, whereas all others are internally enabling each other, i.e. are in consecutive execution.

According to [TBW94], a maximum busy period over $q + 1$ executions can be obtained by taking $q + 1$ times the WCET of task t_a and adding to it the period-and-jitter interference characterizations of all higher priority tasks, multiplied by the respective higher priority WCETs. The timespans of the period-and-jitter interference characterizations are set to the maximum busy period, which makes the determinations of maximum busy periods iterative fixed point computations.

Note that by this method of computing maximum busy periods, which is identical to the one described in [HWGB13], we implicitly assume that all higher priority tasks are simultaneously enabled with the first of the $q+1$ executions of the task under observation. This assumption leads to a significant over-approximation that we attempt to alleviate with the additional consideration of **cyclic data dependencies**.

The basic idea of limiting interference using cyclic data dependencies can be easily explained using the two tasks being depicted in Figure 8.1. Let us assume that task t_a and task t_b are executed on the same processor and that task t_b has a higher priority than task t_a . Due to the $\delta_{a,b}$ initial tokens on the edge from task t_a to task t_b it holds that task t_b can execute $\delta_{a,b}$ times before it requires another tokens produced on the finish of an execution of task t_a . Likewise, it holds that task t_a can execute $\delta_{b,a}$ times before it requires a token produced by task t_b . Taking these two constraints together it follows that during one execution of task t_a no more than $\delta_{a,b} + \delta_{b,a} - 1$ executions of task t_b can take place, which gives us another way to limit the interference of task t_b on task t_a than period-and-jitter.

In the following we do not only take simple cycles as depicted in Figure 8.1 into account, but also complex cycles that expand over multiple buffers and tasks. We determine the numbers of tokens on such complex cycles by applying the Floyd-Warshall algorithm [Flo62] on the precedence constraint analysis model. Considering such cyclic dependencies we can effectively reduce the maximum busy periods based on period-and-jitter alone, which can result in a significant accuracy improvement.

Given reduced maximum busy periods we can finally compute the **maximum response time** of a task t_a as exemplified in Figure 8.4, which conservatively bounds the execution time of task t_a , any higher priority interference that can occur during this execution, as well as any self-delay. Such a maximum response time consequently represents a periodic upper bound on the intervals between maximum external enabling and maximum finish times of task executions, as required.

As described in Section 8.1.2 we calculate bounds on schedules and maximum response times iteratively until either a temporal constraint such as a required minimum throughput or maximum end-to-end latency is violated or until all maximum response times have converged. In the latter case all analysis models become valid abstractions of the respective application model and the determined parameters become valid bounds on the actual parameters within the application.

Figure 8.4: Computation of the maximum response time $\hat{R}_a$ of a task t_a based on maximum busy periods $\hat{w}_a(q)$ that comprise $q + 1$ executions of t_a , as well as any higher priority interference that can occur during the maximum busy periods.

8.1.4 ITERATIVE BUFFER SIZING AND SYNCHRONIZATION EDGES

Without the consideration of cyclic data dependencies to limit interference there would be a monotone relationship between buffer capacities and throughput, in that larger buffer capacities would never result in a lower throughput. If the primary goal were to determine whether an application can meet a certain throughput constraint irrespective of buffer capacities, it would be thus advisable to leave buffer capacities unconstrained during analysis, i.e. assume the numbers of initially empty buffer locations such as $\delta_{b,a}$ in Figure 8.1 as infinite. After convergence of the analysis flow, we could then use the analysis results to determine sufficiently large buffer capacities such that the analysis results would not be impacted by the now constrained buffers.

The consideration of cyclic data dependencies, however, breaks this monotonicity: As smaller buffer capacities impose tighter cyclic dependencies and tighter cyclic dependencies can result in less higher priority interference, it can occur that **smaller buffer capacities result in higher throughput guarantees**.

In order to exploit this effect we replace the post-analysis buffer sizing with an **iterative buffer sizing**. For that matter we simply apply the same formulas as otherwise used after analysis in each iteration of the analysis flow, such that buffer capacities are not determined based on the convergent, final analysis results, but based on the intermediate ones. We use such buffer capacity estimates as cyclic data dependencies to limit interference between tasks.

We show that by using an iterative buffer sizing superior analysis results can be obtained. Note that by applying the iterative buffer sizing for blocking write buffers we are **not only analyzing, but actually optimizing** an application to meet a certain throughput constraint. This also implies that the determined results are only valid for an application if the same buffer capacities are also used in the implementation. For non-blocking write buffers the analyzed application remains unchanged, but better analysis results can be achieved nevertheless, as cyclic data dependencies can be a more accurate interference characterization than period-and-jitter alone.

Finally, we take the limitation of interference using cyclic data dependencies to yet another level by introducing non-functional synchronization edges that directly connect higher and lower priority tasks on the same processor. We treat these edges as edges corresponding to a FIFO buffer and can use them to optimize the schedules of tasks for a certain throughput.

If we consider such edges as non-blocking write buffers, we can use them to derive a more accurate interference characterization than achievable by only using the preexisting data dependencies. If the edges are treated as edges belonging to a blocking write buffer instead, we can effectively vary the scheduling of tasks between static (all cycles on synchronization edges contain one token, which means mutual exclusivity) and pure SPP (cycles on synchronization edges unconstrained). Again, we then need to implement such edges as well, which is luckily associated with a very low implementation cost since these dependencies are only processor-local and due to their non-functionality do not have any space requirements.

8.1.5 EXECUTION INTERVAL ANALYSIS

Even with the additional consideration of cyclic data dependencies, the analysis using period-and-jitter remains rather crude. The reason for the comparably low analysis accuracy is that the period-and-jitter interference characterization assumes that the task under observation, as well as all higher priority tasks are enabled at the same critical instant, such that any correlation between enabling and finish times of task executions is lost. To amend this, we propose our second, more effective analysis technique, which is based on so-called **execution intervals**.

Instead of determining maximum jitters from the input intervals we can as well determine periodic execution intervals that are formed by lower bounds on the enabling times and upper bounds on the finish times of higher priority task executions. We use these intervals to derive a more accurate interference characterization than period-and-jitter. The fundamental principle is rather simple: Taking the same notion of execution intervals, as well as assuming that the first of the $q + 1$ executions considered in a q -level busy period is enabled at its maximum external enabling time (we prove that this is allowed), we just determine the number of overlaps between this maximum busy period and the execution intervals of each higher priority task, multiply this number by the respective higher priority task WCET and sum up all such interferences. This method results in an iterative fixed point computation of maximum busy periods again.

In order to further increase analysis accuracy, we combine this interference characterization with **precedence constraints** obtained from the precedence constraint analysis model. As the execution interval interference characterization is a two-sided bound, whereas the period-and-jitter bound is single-sided, it is not even required to consider cycles. Instead, even unidirectional precedence constraints suffice, such that also acyclic applications without feedback loops or

limited buffer capacities can benefit from this accuracy improvement. Lastly, we also consider both our iterative buffer sizing and the introduction of synchronization edges in the context of this analysis technique, resulting in an even better analysis accuracy.

8.1.6 TASK PHASE ANALYSIS

All analysis techniques discussed so far are restricted to single-rate, single-phase applications. In order to increase applicability we discuss an **analysis technique for multi-rate, multi-phase applications**, i.e. a technique that is capable of analyzing the more entropic multi-rate, multi-phase models effectively. Besides, the requirements on the underlying applications, as imposed by the aforementioned approaches, are rather strict as they assume that the acquires on all buffers connected to an implemented task are conducted atomically at the beginnings of task executions and that the respective releases are conducted atomically at their ends, which is rather challenging to realize on most hardware platforms. As detailed later, such **challenging hardware requirements** can be also addressed with a temporal analysis for multi-rate, multi-phase applications.

We could rather easily make the aforementioned techniques applicable for multi-rate, multiphase applications as follows: From Section 8.1.2 we know that the analysis models are CSDF models for multi-rate, multi-phase applications. Since there exist methods to expand any CSDF graph to an equivalent HSDF graph [BELP96] we could simply insert another step before the main part of our analysis flow, convert the CSDF graphs to HSDF graphs and thereafter treat all phases that become separate HSDF actors in the expansions as separate tasks. If we also treated the phases in the response time analysis model as separate tasks we would obtain an analysis approach on which we could apply both our techniques based on period-and-jitter and execution intervals.

However, this rather simple method would come at the cost of a significant loss in analysis accuracy. This is due to the fact that if for instance a higher priority task t_b can interfere with all five phases of a lower priority task t_a and if we treated all phases of task t_a as separate tasks, we would compute maximum busy periods for each of the phases separately, effectively accounting for the same interference of task t_b five times.

To overcome this problem we introduce our task phase analysis technique, which allows to compute **maximum busy periods over multiple task phases**, such that the same interference is only accounted for once.

In the best-case and worst-case analysis models we do not remove all self-edges of multi-phase actors, but only the ones with one token between the last and first phases, while the others are preserved to keep the internal order between phases. This means on the one hand that the maximum firing durations we use in the worst-case analysis model do not need to incorporate self-delay for any phases except for the first. And on the other hand, this also implies that the start times computed with the LPs applied on these models compute bounds on external

enabling times for the first phases and bounds on total enabling times for the remaining ones. With slight modifications of the LPs we can nevertheless also compute bounds on external enabling times for all other phases than the first. Based on that we can obtain lower and upper bounds on (external) enabling times and finish times. As before, we use these bounds as input intervals in the response time analysis model.

As the interference characterization for the maximum response times of task phases we first choose period-and-jitter with a consideration of cyclic data dependencies for simplicity. Later we also introduce a more accurate maximum response time computation for task phases that is based on execution intervals and precedence constraints. The latter technique has the same effectiveness as the respective execution interval technique for single-rate, single-phase applications, but is applied on more entropic analysis models. According to Section 2.2 this implies that our multi-phase approaches does not only come with an increased applicability, but also with an increased analysis accuracy.

Remember that task executions are always enabled either externally or internally by preceding executions of the same task. Analogously, a task phase execution is always enabled either externally or internally by preceding, but different phases of the same task. In order to compute the maximum response time of a task phase $t_{a,x}$ we consequently compute maximum busy periods over multiple task phases in consecutive execution, as exemplified in Figure 8.5.

Figure 8.5: Computation of the maximum response time $\hat{R}_{a,0}$ of a task phase $t_{a,0}$ based on maximum busy periods $\hat{w}_{a,x}(q)$ that comprise all phase executions of task t_a between execution $i - q$ of phase $t_{a,x}$ and execution i of phase $t_{a,0}$, as well as any higher priority interference in that interval (maximum busy periods beginning with executions of phase $t_{a,2}$ do not need to be considered because $t_{a,2}$ cannot be externally enabled).

More precisely, we assume that a maximum busy period covers a number of phases in consecutive execution, of which only the first is externally enabled and of which $t_{a,x}$ is the last, as well as any interference that can occur during this consecutive execution. Moreover, we take into account that some phases cannot be externally enabled, as they are not directly connected to any FIFO buffers. These phases can be skipped as first phases of maximum busy periods.

Again, any phase execution is either externally enabled or internally enabled by its preceding phase, with the preceding phase also being either externally or internally enabled by its respective preceding phase, and so on. By considering all maximum busy periods as described above we consequently cover all cases of self-delay, such that we can use such maximum busy periods to determine **maximum response times of task phases** which bound phase execution times, higher priority interference and self-delay, as shown in Figure 8.5.

Finally, note that because we do not remove all self-edges in the best-case and worst-case analysis models, but only the ones between last and first phases, we also do not need to consider self-delay for most of the maximum response times, except for the maximum response times of the first task phases that do not have incoming self-edges.

8.1.7 MULTIPLE SHARED RESOURCES

With a rather simple modification we can extend our analysis technique for multi-rate, multi-phase applications using execution intervals to a technique applicable for **applications with multiple shared resources**. The main difference compared to processor sharing is that the sharing of e.g. a memory does not need to lead to interference for all phases of a task, but only for the phases that actually access the shared memory.

To enable the consideration of interference that only affects some, but not all phases of a task we introduce the notion of so-called **interference periods** that are used to track up to which points within a maximum busy period computation the interferences of higher priority phases are considered. Based on such interference periods we devise two different, temporally conservative methods to consider higher priority interference. In the first method we compute interference by extending interference periods over multiple phases, similar to the methods discussed before, while in the second method we reinitialize interference periods, such that no previously considered interference is considered. Taking the minimum of the two methods for each higher priority interference in each maximum busy period extension allows us to determine maximum response times of task phases for the multiple shared resource use-case in a temporally conservative and nevertheless accurate manner.

8.1.8 SIMULATION

Besides the presented analysis techniques that operate on analysis models derived from application models via abstraction, we also consider simulation as a viable alternative for real-time system evaluation. One important feature of our simulation approach is that we use as our simulation model **the same application model** from which we also abstract analysis models.

Usually, neither the numbers of allowed inputs nor the numbers of possible execution times that can occur in an application model are finite. Therefore it

generally holds that our application models are not only non-deterministic, but actually entail an infinite number of scenarios. In contrast, every simulation run that we conduct on an application model gives us a single trace of all determined timestamps and values, i.e. every simulation run for itself is deterministic. From this follows that an exhaustive simulation according to Chapter 1, i.e. a simulation that covers all cases, is usually impossible.

In consequence, **simulation can be rarely used to give guarantees**. However, albeit the importance of guarantees for real-time systems is undeniable, it would be rather inappropriate to consider simulation as useless in this context: First, for many applications not only worst-case guarantees are of interest, but also **average-case performance** that can be determined much easier using simulation. Second, there are many situations during a design process in which only a **quick verification** of parts of an application is desirable. For such cases the effort for an accurate construction of analysis models would be rather disproportionate, whereas a simulation that can be directly applied on application models may be more suitable.

Third, constructing analysis models from application models is an error-prone process. Consequently it can happen that analysis results are not conservative to all traces that can occur within an application model, even though the applied analysis techniques are correct. This makes it desirable to **increase confidence in the constructed analysis models** by comparing analysis results to simulation results. However, most simulation approaches cannot be used for that purpose as they are not based on the same application models as analysis, making the respective results incomparable.

In contrast, our approach is directly applied on the application models also considered for analysis. If we take one or more traces obtained via simulation together, we can conclude that such a conjunction of traces is itself an inclusion refinement of the simulated application model. From this perspective follows that it is not possible to use simulation for verification of analysis results, unless an exhaustive simulation can be conducted such that the conjunction of simulation traces becomes equal to the simulated application model. Nevertheless, we can use simulation for the **falsification of analysis results** as analysis results must be by definition conservative to simulation results obtained from the same application model. This means that we can use our simulation to gain confidence in analysis models, as desired.

And fourth, our simulator cannot be only applied on application models, but also on analysis models. As analysis models are less entropic than application models it is also more likely that exhaustive simulation can be applied on the former. In that case we can also use simulation to actually analyze analysis models, which can be both used productively, as well as to define a benchmark for analysis techniques as no analysis technique can be as effective as an exhaustive simulation.

The Hydra API (HAPI) simulator that we introduce to address the described use-cases is defined on top of SystemC and enables simulation of models consisting of the same components that are also available in application models, i.e.

periodic sources, FIFO buffers and tasks. HAPI allows for the accurate temporal modeling of several scheduling policies, such as SPP that we mainly consider, but also e.g. RR or TDM.

8.2 RELATED WORK

After having presented the basic ideas of our analysis and simulation techniques, we discuss related work in this section. We first evaluate works related to the temporal analysis of real-time systems and thereafter other simulation approaches.

We make use of the example depicted in Figure 8.6, a benchmark taken from [P⁺08], to illustrate the differences between the various temporal analysis approaches. The left side of the figure shows the task graph of a streaming application, with tasks t_a and t_c being executed on a shared processor using an SPP scheduler and task t_c having a higher priority π_2 than task t_a with priority π_1 . The WCETs of the tasks are written next to them. The application is driven by a strictly periodic source with the period $P_s = 10\text{ms}$ and a variable jitter $J_s = [0..50]\text{ms}$. Moreover, inter-task communication is realized via FIFO buffers whose optimum capacities are unknown before analysis. The right side of Figure 8.6 depicts the maximum end-to-end latency $\hat{L}$ determined by different analysis approaches drawn against the source jitter J_s .

Figure 8.6: Left: Task graph with variable source jitter. Right: Source jitter J_s vs. maximum end-to-end latency $\hat{L}$ for different temporal analysis approaches.

[WBS09] presents a dataflow analysis approach for applications using starvation-free schedulers like RR, for which the minimum service of a task can be determined independently of the enabling rates of other tasks. This approach shows that dataflow analysis techniques can combine cyclic data and resource dependencies to derive the minimum throughput of real-time applications executed on multiprocessor systems. In [HWGB13] a dataflow analysis framework is introduced that combines dataflow modeling with response time analysis. It derives enabling rate characterizations of tasks that can be used as interference characterizations, thereby extending the scope of dataflow analysis techniques to applications with non-starvation-free schedulers like SPP. The introduction of enabling

rate characterizations allows for an accuracy improvement for starvation-free schedulers such as RR as well.

Thanks to the usage of dataflow graphs for the modeling of data dependencies it holds that [HWGB13] natively supports cyclic data dependencies. However, interference between tasks is solely determined based on a rather crude period-and-jitter interference characterization which assumes that all interfering tasks are externally enabled simultaneously, resulting in the circled curve in Figure 8.6.

To address this lack of accuracy we introduce our first analysis technique, the period-and-jitter analysis sketched in Section 8.1.3. This technique is based on [HWGB13] and derives maximum response time equations which take the effect into account that cyclic data dependencies bound interference. As shown later in the case study, the usage of cyclic data dependencies to limit interference can result in a significant accuracy improvement for the analysis of cyclic real-time systems. However, the task graph in our example is acyclic, such that the interference-limiting effect of cyclic data dependencies cannot be directly exploited. Hence our period-and-jitter analysis technique determines the same results as [HWGB13] here, i.e. the results indicated by the circled curve.

In contrast, the combination of limiting interference using cyclic data dependencies with an iterative buffer sizing, as introduced in Section 8.1.4, leads to a significant improvement of analysis results. The iterative buffer sizing makes the previously acyclic task graph cyclic, which allows to bound interference using these cycles. The bounded interference, in turn, results in smaller maximum response times and eventually smaller end-to-end latencies, as indicated by the dotted curve in Figure 8.6. Note that analysis results obtained with an iterative buffer sizing for blocking write buffers (which we applied here) are only valid for the implementation if the same FIFO buffers with blocking writes are also used in the implementation.

Generally, buffer sizing of buffers with blocking writes has been studied extensively in the context of dataflow analysis, e.g. in [WBS07a, SGB08, MBGS10]. In [SMvB14] it is shown how dataflow models can be used to compute buffer capacities of buffers with non-blocking writes as well. However, it is assumed that best-case and worst-case response times are given, which prevents an exploitation of the relation between buffer capacities, interference and response times. To the best of our knowledge, the buffer sizing approaches presented in this thesis are the only ones that are conducted iteratively alongside analysis and as such also the only ones that can systematically exploit the interference-limiting effect of restricted buffer capacities.

Another way to address the low accuracy of the crude period-and-jitter interference characterization is the usage of offsets, which represent bounds on the external enabling times of tasks. The external enabling time of a task is closely related to the so-called release time [Tin94]: Both indicate the time at which a task iteration is put in the ready queue of its scheduler, with the difference that a release time occurs periodically, whereas the external enabling time reflects the enabling of a task iteration due to satisfaction of its data dependencies. Offsets

in the context of temporal analysis were pioneered in [Tin94]. The offsets used in this approach are static, however, limiting applicability to applications with strictly periodic schedules. This limitation is relaxed in [PGH98] by the introduction of dynamic offsets, which combine static offsets with jitters and also allow to consider applications with data-driven scheduling.

The SymTA/S approach [H⁺05], one of the approaches discussed in [P⁺08], combines standard event models in the time-interval domain with response time analysis, which in turn characterizes interference between tasks using dynamic offsets. It makes use of an iterative procedure of traffic characterization and response time calculation, enabling the analysis of applications with cyclic resource dependencies. However, SymTA/S lacks support for (arbitrary) cyclic data dependencies, which is required to accurately capture FIFO buffers with limited capacities in the analysis model. Consequently, FIFO buffer capacities must be assumed as infinite during analysis and buffer sizing can be only applied post-analysis (in [Jer05] a suitable backlog-based, post-analysis buffer sizing for buffers with non-blocking writes is presented for SymTA/S).

Also making use of dynamic offsets to characterize interference is the Modular Performance Analysis (MPA) approach [WTVL06]. MPA is based on Real-Time Calculus (RTC), allows for arbitrary event models and derives its traffic characterization in the time-interval domain. MPA cannot capture cyclic data dependencies in its analysis model, which restricts the approach to post-analysis buffer sizing techniques. In [TS09] a generalization of the MPA framework is presented, which supports modeling of cyclic data dependencies by deriving a traffic characterization in the time domain. Consequently, buffer capacity constraints can be modeled in this framework. However, both buffer sizing and a combination of cyclic data dependencies with cyclic resource dependencies are not discussed. Another approach based on RTC is presented in [L⁺13] and allows for a consideration of buffers with blocking writes in the analysis model. The approach lacks support for cyclic dependencies other than those imposed by finite buffers. Cyclic resource dependencies are not considered as well.

Dynamic offsets in general have a significant shortcoming if tasks with large jitters are involved, as it is the case in our example: The analysis has to assume for the worst-case that an iteration i of task t_a executes as late as possible (that is, with maximum jitter), whereas iteration i of task t_c executes as early as possible (with minimum jitter). This leads to the detection of interference between iteration i of task t_a and iteration i of task t_c , which obviously cannot occur in the application itself. An overestimation of interference is the consequence, which is illustrated by the crossed curve obtained with both the SymTA/S approach and MPA.

This problem can be addressed by replacing dynamic offsets with relative offsets, which are not defined in relation to the source, but directly in relation to interfering tasks, as it is proposed in [HE06]. While analysis accuracy is high for this approach, its applicability is not only limited to acyclic, but even to tree-shaped task graphs.

Another possibility to reduce the inaccuracies of dynamic offsets is the addition of explicit exclusions of interference due to precedence constraints. Such

a method is first considered in [YW98] in which preempted tasks are analyzed in chains with other preempted tasks, preventing to account for the same interference multiple times. In [K⁺13] a notion of execution intervals is used in combination with precedence constraints, similar to our execution interval approach. However, precedence relations are only evaluated between tasks in the same iteration and only acyclic applications are considered.

Limiting interference using dynamic offsets and precedence constraints is also discussed in [PGH99] and refined in [Red04]. The latter refinement is implemented as part of the Modeling and Analysis Suite for Real-Time Applications (MAST) framework [GHGGPGDM01] and is capable of an accurate characterization of interference for our example. This is illustrated by the boxed curve, which is equal to the curve determined with our execution interval approach. However, the applicability of the technique in [Red04] is also limited to acyclic task graphs (in fact, even tree-shaped task graphs). Aside from the obvious shortcoming that task graphs with feedback loops cannot be modeled, this can also have a negative impact on analysis accuracy for acyclic graphs.

This negative impact can be illustrated by assuming reversed priorities in our example, such that task t_a has a higher priority than task t_c . In this case, the technique in [Red04] would conclude that an iteration i of task t_c could potentially experience interference from all iterations of task t_a with a higher iteration index than i . In contrast, our execution interval approach with its iterative buffer sizing would introduce additional backward dependencies from task t_c to task t_a which would bound interference between these tasks, resulting in more accurate analysis results than achievable with [Red04].

To the best of our knowledge, our execution interval approach is the first to consider offsets in the context of applications with arbitrary cyclic data and resource dependencies. The resulting support for feedback loops and FIFO buffers with predefined capacities distinguishes our work from all other offset-based approaches. Our approach does not only support cyclic data dependencies natively, but uses the precedence constraints imposed by such dependencies to bound interference. On top of that, we can apply the same iterative buffer sizing for our execution interval approach that we have already introduced for our period-and-jitter technique, such that even acyclic task graphs can be made cyclic. The additionally introduced backward dependencies can then be used to bound interference, which can result in a significant improvement of analysis results for acyclic applications as well.

However, all approaches considered so far are limited to single-rate, single-phase applications whose data dependencies are expressible as HSDF graphs.

This limitation is removed in [HGWB14], considering different, but fixed rates on the inputs and outputs of tasks, which allows to analyze applications expressible with SDF graphs. The main contribution of [HGWB14] is an interference characterization which captures bursts due to rate conversions more accurately than period-and-jitter. The maximum response time computation, however, is unchanged, making our contributions on the computation of maximum response

times over multiple phases somewhat orthogonal to the ones from [HGWB14]. In contrast to all aforementioned works, our task phase analysis approach can analyze any applications expressible with CSDF models, allowing the analysis of tasks with cyclo-statically changing rates and execution times. This is also relevant for applications that are naturally expressible with HSDF or SDF graphs, as CSDF models can be used to relax the requirement of atomic synchronization on the boundaries of task executions.

The consideration of multiple shared resources is discussed in [SE11] and improved in [KCH17] by taking the scheduling patterns of tasks into account. However, both approaches assume independent tasks, i.e. tasks being activated in a time-driven instead of a data-driven fashion, and use standard event models as in SymTA/S to characterize shared resource accesses. A response time analysis for dependent tasks with multiple shared resources is presented in [CKH16]. This method makes use of an iterative procedure of bounding shared resource accesses and maximum response times of tasks. Interference on shared resources other than processors is also bounded using standard event models. In contrast, we characterize interference of both shared processors and other shared resources jointly, with a combination of execution intervals and an explicit consideration of precedence constraints. This is more accurate than using standard event models. The consideration of task phases which do not necessarily all access all shared resources further improves accuracy compared to related works.

Finally, let us come to the comparison of our HAPI simulation approach to various other discrete event simulation approaches, with a focus on application scopes.

Transaction Level Modeling (TLM) [Don04] is an approach to build simulation models of multiprocessor systems at various levels of abstraction. The SystemC kernel [OSC] is used for simulation of these models. Modeling at higher abstraction levels improves simulation speed, but results in an approximation of temporal behavior. Building the required TLM models usually takes a significant amount of time. A HAPI simulation model can be seen as a TLM model that operates between the Communicating Processes (CP) and Programmer's View (PV) abstraction levels because there is no notion of registers and instructions, but preemptions of tasks by an operating system kernel can be simulated. The HAPI simulation models are equal to the application models from which also analysis models are derived by means of abstraction. The simulated application models themselves are abstract to the modeled applications in the sense that they do not directly reflect the structure of the modeled applications. For instance, there is no notion of communication buses or networks. As a result of this high level of abstraction a relatively low number of events need to be handled by the simulation kernel and simulation is typically fast.

Platform architect [Syn] is a commercial simulation tool developed by Synopsys for the evaluation of the performance of several multiprocessor system architecture configurations in early design phases. Platform architect is based on the SystemC simulation kernel, simulates at the PV abstraction level and has

a graphical user interface. Models of communication buses and memory controllers are provided and describe what happens during transactions instead of per each clock cycle. The use of these transaction level models improves simulation speed at the cost of accuracy. Instruction set simulation models are used to model processors. A key difference between Platform Studio and the HAPI simulation approach is that HAPI requires WCETs of tasks to be determined independently of other tasks in the application. Therefore HAPI is not suitable for analyzing increased task execution times due to contention on memory ports that can occur on both read and write accesses of processors.

The Parallel Object Oriented Specification Language (POOSL) [T⁺07] is a system-level modeling language based on a small set of language primitives. POOSL enables a precise representation of an application based on mathematically defined semantics. It consists of a process part and a data part. The process part is based on a real-time probabilistic extension of the process algebra Calculus of Communicating Systems (CCS), whereas the data part relies upon the concepts of traditional sequential object-oriented programming languages like C++. POOSL descriptions are simulated using the discrete event simulator Rotalumis or executed in an interpretative way by the SHESim tool. The underlying formal model of POOSL is the Timed Probabilistic Labeled Transition System (TPLTS). A TPLTS can be transformed into a Markov chain whose equilibrium distribution can be computed analytically. However, the Markov chains obtained after transformation have typically a very large number of states, resulting in a prohibitive run-time of the algorithms to compute the equilibrium distribution. As a consequence it is typically only practical to simulate a POOSL description to obtain an estimate of the equilibrium distribution. The equilibrium distribution corresponds to long-running average performance metrics. A significant difference between HAPI and POOSL is that the application models used by HAPI have dataflow instead of TPLTS as underlying formal model. For a subclass of dataflow models the minimum throughput and maximum latency can be computed analytically with computationally efficient algorithms, which is not possible for the probabilistic TPLTS models. Another conceptual difference is that HAPI has built-in support for a number of schedulers, while POOSL abstracts from scheduling.

Y-chart Application Programmer's Interface (YAPI) [dK⁺00] is an event-driven simulator for KPNs [Kah74]. The YAPI simulator simulates functional behavior like the HAPI simulator, but does not support a notion of time. Furthermore, unlike HAPI, the YAPI simulator does not support auto-concurrent execution. The YAPI simulator also does not simulate sharing of processors, such that e.g. SPP scheduling cannot be expressed.

A number of SystemC-based simulation approaches [Yu10, PVB05] have been developed to deal with asynchronous events such as interrupts. For that matter preemption points are introduced by splitting tasks into smaller fragments. The tasks are split because they can communicate with global variables during their executions. At these preemption points the scheduler checks whether an asynchronous event has occurred. The use of preemption points introduces blocking

times in the simulator which cause an unpredictable amount of additional delay between the arrival of asynchronous events and their handling, a delay that does not occur in reality. Such an additional delay can make simulation results more pessimistic than analytical results, making approaches with preemption points unsuitable for the falsification of analysis results. Our HAPI simulation approach does not make use of preemption points.

8.3 RELATION TO PAPER VERSIONS

In this section we highlight some important differences between the content presented in the remainder of this thesis and the papers on which it is mainly based on. Chapter 10 comprises the contents of [10] and [9]. Chapter 11 is based on [8], Chapter 12 on [5] and [6] and Chapter 13 presents the contents from [7].

Compared to the paper versions, we have modified several notations, in order to establish consistency throughout the thesis. The most prominent changes are as follows:

- In all chapters, tasks are denoted as t_a , t_b and t_c instead of τ_i , τ_j and τ_k , while task executions are indexed with i instead of n and timestamps are denoted with τ instead of t .
- In all chapters, WCETs are denoted with $\hat{\chi}$ instead of C .
- The analysis flows in Chapters 10 and 11 are rearranged compared to [10], [9] and [8], such that bounds on task schedules are computed before maximum response times.
- The interference characterizations in Chapter 10 are renamed, such that η refers to the period-and-jitter characterization, ζ to the cyclic data dependency characterization and γ to the combination of the two.
- Unreduced maximum busy periods are denoted as $\hat{w}'$ and reduced ones as $\hat{w}$ in Chapter 10, instead of the other way around.
- Maximum response times based on unreduced maximum busy periods are denoted as $\hat{R}'$ and maximum response times based on reduced maximum busy periods as $\hat{R}$ in Chapter 10, instead of the other way around.
- The maximum busy period index is shifted by one in Chapters 10 and 11, such that a q -level maximum busy period of a task t_a includes $q + 1$ executions of task t_a instead of q executions.
- The interference characterizations used in Chapter 11 are denoted with ψ instead of γ to prevent confusion with the interference characterizations considered in Chapter 10.

- The second partial maximum busy periods are denoted with $\hat{w}^\diamond$ instead of $\hat{w}'$ in Chapter 11 to avoid confusion with unreduced maximum busy periods.
- Execution intervals in Chapter 11 are denoted as $\mathcal{E}$ instead of $\mathcal{I}$ to avoid confusion with the interference function $\mathcal{I}$.
- x and x' are reversed in the algorithm depicted in Figure 12.5.

Besides, we have extended the paper versions in several ways:

- For all presented approaches, the relation between the used analysis models and the respective application models is formalized by means of inclusion and bounding abstraction in Chapter 9, which allows us to formally prove conservativeness of our analysis approaches, that is, to present additional proofs that the obtained analysis results are not only valid for the analysis models, but also for the underlying, implemented real-time systems.
- So-called synchronization edges are introduced in Chapter 10, which can be used to optimize task schedules for meeting throughput constraints even more than possible with the iterative buffer sizing alone.
- An elaborate discussion on the termination of the execution-interval analysis flow is added in Appendix B.2, mirroring the discussion on termination of the period-and-jitter analysis flow.
- A modified version of our task phase resource dependency analysis is presented in Chapter 12 that replaces the period-and-jitter interference characterization with execution intervals, enabling a higher analysis accuracy.
- The latter is further extended in Chapter 12 to increase applicability from multi-rate, multi-phase applications to applications with multiple shared resources such as shared processors, buses, memories or peripherals.

Modeling Concurrent Real-Time Systems

In the following chapters we discuss various analysis techniques that can be applied on analysis models in order to give guarantees on the temporal behavior of real-time systems with shared resources. Before describing these techniques, however, we need to provide a solid theoretical foundation with respect to the relationship of application models on the one hand and analysis models on the other, such that the determined guarantees do not only hold for the analysis models themselves, but also for the underlying applications. For that purpose we first define what kind of applications we attempt to analyze in Section 9.1. Thereafter, in Section 9.2 we describe how to express such applications in the timed component model introduced in the first part of this thesis. Lastly, we make use of inclusion and bounding abstraction techniques in Section 9.3 to establish a relation between such application models and the analysis models used in the remainder of this thesis, guaranteeing the validity of analysis results for the underlying applications by construction.

9.1 APPLICATIONS & APPLICATION MODELS

The applications we consider in the following are real-time streaming applications scheduled on shared processors with SPP scheduling. This implies that we consider applications with both data-dependent and resource-dependent tasks. Within this class of applications, we further differ between single-rate and single-phase applications on the one hand, as well as the broader class of multi-rate and multi-phase applications on the other. For brevity we call single-rate and single-phase applications simply single-phase applications and likewise multi-rate and multi-phase applications multi-rate applications in the following.

We consider real-time streaming applications Λ that can be expressed in application models G^Λ consisting of one or more task graphs $G^{T,\Lambda} = (T^\Lambda, D^D, B, t_s^\Lambda)$ and a set of resource dependencies D^R . Task graphs are weakly connected graphs whose vertices $t_a^\Lambda \in T^\Lambda$ we call tasks. The edges of a task graph are data dependencies $d_{ab} \in D^D$ leading from writing tasks t_a^Λ to reading tasks t_b^Λ via FIFO buffers $\beta_{a,b} \in B$, which enables communication between tasks. We further assume that each task graph is triggered by a single source t_s^Λ that is producing values to FIFO buffers at a strictly periodic rate.

A FIFO buffer $\beta_{a,b}$ connects exactly one writing task t_a^Λ to exactly one reading task t_b^Λ . In the following we differ between two different buffer types, buffers with blocking and buffers with non-blocking writes.

Accesses to blocking write buffers consist of three phases, which differ slightly depending on whether the accessing task is a reading or writing task. A read (write) buffer access always begins with the acquisition of a full (empty) buffer location. If there is no such location in the buffer then the accessing task is stalled until a location in the buffer is written (freed) by the other connected task. After acquisition, the acquired location can be read (written) by the accessing task. A buffer access is concluded by the release of the read (written) location, making it available for acquisition to the other, writing (reading) task.

In contrast, for non-blocking write buffers the acquisition of buffer locations by the writing task is unconditional, which means that locations can be overwritten if they are not read in time by the respective reading task. This behavior facilitates the implementation of an application as buffers become unidirectional and also facilitates analysis as applications without functional feedback loops remain acyclic, whereas every blocking write buffer introduces a cycle. However, for non-blocking write buffers the temporal behavior can impact the functional behavior due to the possibility of unwanted overwriting. Hence we have to take special care of this buffer type during analysis.

Communication between tasks is single-rate, which means that each task execution accesses exactly one buffer location in all connected buffers. Tasks are executed in a data-driven manner, such that a task execution is not triggered by a global or local clock, but rather by the availability of full or empty locations in the incoming or outgoing buffers, respectively. A task is externally enabled once there is at least one full location in all incoming buffers and at least one empty location in all outgoing buffers for blocking write buffers. Once a task is externally enabled and once the previous execution of the same task has finished we say that the task is enabled.

Note that we require that no task can execute without a preceding production of its source. This implies that no task can have both initially full locations in all its input buffers and initially empty locations in all its output buffers.

Reflecting the data-driven behavior of tasks, we assume that a task execution begins immediately once the task is enabled. For now we further require that all

task executions consist of a single phase, that all acquires occur atomically at the beginnings of executions and all releases atomically at their ends. This implies that accessing the buffers stored in memory does not take processor time and that all buffers can be accessed at once. As this is usually difficult to realize on existing hardware platforms, we propose a less restrictive application definition in the next section. We call the moment at which all buffers are released at the end of a task execution the finish time of an execution.

Every task t_a^Λ is statically mapped to a single processor and each of its executions takes a certain execution time, that is, the time between enabling and finish of an execution, to execute on the mapped processor. The execution time of a task explicitly does not have to be the same in every execution, but is assumed to be constrained from below by a BCET $\check{\chi}_a$ and from above by a WCET $\hat{\chi}_a$. In the following we refer to execution time as the time needed for a task to execute on the mapped processor in isolation, i.e. without interference of other tasks. If no other tasks mapped to the same processor are enabled during the execution of a task then the total time between enabling and finish of that execution is equal to the respective execution time.

However, the time between enabling and finish is usually longer than the sole execution time, as a consequence of processor sharing. For the processor sharing we consider an SPP scheduling scheme, such that each of the tasks is statically assigned an integer priority, with no two tasks on the same processor having the same. The term preemptive is coined on the policy that whenever a lower (smaller integer) priority task is executing and a higher priority task is enabled, the lower priority task is preempted and thereby suspended until the higher priority task finishes its execution. In the following we assume a friction-free scheduler implementation, such that preemption and continuation of task executions occur without any additional delays.

We express the dependencies that occur due to processor sharing as resource dependencies $r_{ab} \in D^R$. As higher priority tasks can influence the temporal behavior of lower priority tasks and not vice versa (which would be different for a non-preemptive scheduler), the resource dependencies r_{ab} always point from a higher priority task t_a^Λ to all lower priority tasks t_b^Λ on the same processor. Consequently, the highest priority task on a processor does not have incoming resource dependencies, whereas the lowest priority task does not have outgoing resource dependencies. Finally, also note that the resource dependency set D^R is defined for an entire application and not single task graphs within the application. With that we explicitly allow for processor sharing between tasks from different task graphs.

In the following we use hp_a as a shorthand notation for a set of all higher priority tasks of a task t_a^Λ .

9.1.2 MULTI-PHASE APPLICATIONS

An issue with the application definition given in the previous section is the implicit assumption that acquisitions of locations on all connected buffers are

conducted atomically at the beginnings of task executions, as are releases at execution finishes. This is problematic because in reality the implementation of atomic acquires and releases is difficult and sometimes even impossible. Moreover, the aforementioned application model restricts the class of expressible applications to single-rate applications, that is, applications in which tasks can consume or produce at most one buffer location per execution.

To overcome these shortcomings we introduce another task model in this section, which allows to split task executions into multiple phases executed in a cyclo-static order, i.e. there is a strict order between the phases and - just like with the single-phase tasks - the first phase of a task cannot execute before the last phase of the previous task execution has finished. Moreover, each of the phases can have a different execution time, which is also constrained by phase-dependent BCETs and WCETs, and can acquire and release not only one, but any fixed integer number of locations (even zero). For the rest, the application model is the same as in the previous section, i.e. processors can be shared by multiple tasks and each task is assigned a distinct priority on that processor. In particular, we also assume a single-rate and single-phase source for simplification. If a multi-rate or multi-phase source is needed, it can be always modeled by a virtual multi-phase task directly triggered by the single-phase source.

With this multi-phase application model we can express three different types of tasks, as depicted in Figure 9.1.

Figure 9.1: Different task types expressible in the multi-phase application model.

First, if not only total execution times of tasks are given, but also the execution times between the acquires and releases within single-phase task executions, then such tasks can be split into multiple phases, as exemplified in Figure 9.1(a). The phases then represent the parts of the tasks between acquires and releases, with the assigned execution times being partial execution times of entire task executions. With such split tasks we can relax the strict requirement of atomic acquires and releases from the single-phase application model.

Second, consider the case that such detailed information on acquires and releases within task executions is not given, but it nevertheless holds that the acquisitions of a task happen at the beginnings of its executions and the corresponding releases at its ends, just not atomically. Then it is possible to express such tasks making use of three phases, as shown in Figure 9.1(b). The first phase models a jitter within all acquires take place, the second the actual execution of the task (after the last acquire and before the first release) and the third describes

a jitter within which all releases occur. While the second phase is assigned the execution time of the task, the first and third phases are assigned execution times varying between zero and the maximum time between the first and last acquire or first and last release, respectively.

And third, tasks that natively operate with multiple rates and / or natively consist of multiple phases can be also directly expressed in the multi-phase task model, as depicted in Figure 9.1(c).

Finally note that all three types can be freely combined, such that it is e.g. also possible to express multi-phase tasks with accurate instants of acquire and release times within the different task phases.

9.2 APPLICATIONS IN THE TIMED COMPONENT MODEL

Now that we have introduced both the single-phase and multi-phase application models we can also formally express these models in the timed component model presented in Chapter 4. The required building blocks are FIFO buffer components, strictly periodic source components as well as single-phase and multi-phase task components. Note that in the remainder of this thesis the focus is on the temporal analysis of applications. The functional behavior of any components is consequently omitted. If needed, the functional behavior of tasks can be easily added to the components introduced in this section.

Let us begin with the blocking write FIFO buffer components. We assume that a blocking write buffer $\beta_{a,b} \in B$ that connects a writing task t_a^Λ with a reading task t_b^Λ initially contains $\delta_{a,b}$ full and $\delta_{b,a}$ empty locations, such that the total number of locations in the buffer (i.e. its capacity) is $\delta_{a,b} + \delta_{b,a}$. Each buffer has two input ports, one port p_a modeling the instants at which the writing task releases full locations and one port p_b modeling the instants at which the reading task releases empty locations. Likewise, each buffer has two output ports, one port q_a modeling the instants from which onwards the writing task can acquire empty locations and one port q_b modeling the instants from which onwards the reading task can acquire full locations. With this notation we can express the temporal behavior of a buffer $\beta_{a,b} \in B$ using a timed component with the following relation:

$$\begin{aligned} R_{\beta_{a,b}} = \{ & (X, Y) \in Tr(P) \times Tr(Q) \mid x_a = x_b = x^\infty \Rightarrow y_a = y_b = y^\infty \wedge \\ & x_a \neq x^\infty \vee x_b \neq x^\infty \Rightarrow \forall_{i < \delta_{a,b}} : \tau_{y_b}(i) = 0 \wedge \forall_{i < \delta_{b,a}} : \tau_{y_a}(i) = 0 \wedge \\ & \forall_{i \geq \delta_{a,b}} : \tau_{y_b}(i) = \tau_{x_a}(i - \delta_{a,b}) \wedge \\ & \forall_{i \geq \delta_{b,a}} : \tau_{y_a}(i) = \tau_{x_b}(i - \delta_{b,a}) \} \end{aligned}$$

The first line ensures, in accordance with the definition of timed components, that the absence of inputs results in the absence of outputs. The second line models the presence of $\delta_{a,b}$ initially full and $\delta_{b,a}$ initially empty locations. And

the third and fourth lines ensure that acquires of subsequent locations can be only performed by the connected tasks if the respective locations were previously released. Note the absence of a hard limitation of total buffer locations within the buffer component, that is, a dependence between the streams of full and empty locations. If it is taken care in the connected task components that releases do not happen before acquires (which we ensure in the following) then such a limitation is obsolete as the total number of buffer locations cannot be exceeded anyway. Besides, remember that a buffer component $\beta_{a,b}$ is nothing more than the parallel composition of two token components as presented in Section 4.7 and depicted in Figure 8.1, one with $\delta_{a,b}$ initial tokens and the other with $\delta_{b,a}$ initial tokens.

For non-blocking write buffers we simply set $\delta_{b,a}$ to infinite, which makes writing to the buffer unconditional. This captures the temporal behavior of a non-blocking write buffer accurately. However, as our application model is purely temporal, we cannot capture the impact of overwriting on a non-blocking write buffer on the functional behavior of an application. For that reason we additionally need to check whether the buffers are chosen large enough given the temporal application behavior, such that unwanted overwriting cannot occur. We do this by applying the method discussed in Section 10.4.

In both application models we assume single-rate and single-phase source tasks t_s^Λ that are executing in isolation and strictly periodic with a source period P_s . To define t_s^Λ in the timed component model we reuse the standard HSDF actor component previously introduced in Section 4.7, a self-edge with one token, a constant execution time of P_s per task iteration and an additional restriction on the accepted input streams which prevents delays of the source. For a source that is receiving an external input stream on a port p^{ext} and is writing to $|P| = |Q|$ FIFO buffers (with the ports in P corresponding to the acquires of empty locations and the ports in Q to the releases of full locations) we find the following relation, again omitting functional behavior (with $P^* = \{p^{ext}\} \cup P$):

$$R_{t_s^\Lambda} = \{(X, Y) \in Tr(P^*) \times Tr(Q) \mid \forall_i: \forall_{p \in P^*}: \tau_{X[p]}(i) \leq \tau_{X[p]}(i+1) \wedge \\ \forall_{p \in P^*}: \tau_{X[p]}(i) \leq i \cdot P_s \wedge \forall_{q \in Q}: \tau_{Y[q]}(i) = i \cdot P_s\}$$

The first line captures that only in-order streams are accepted, while the second line ensures that the source is never delayed by any of the connected tasks nor by the external input, i.e. it must hold for all ports $p \in P^*$ and all iterations i that $\tau_{X[p]}(i) \leq i \cdot P_s$. This enables a strictly periodic execution of the source, which is captured by the right-hand side of the second line. Note that proving this built-in criterion of an undelayed source is one of the problems that we attempt to address with the analysis techniques described in the following, that is, that an application adheres to the throughput constraint imposed by the source periodicity of P_s .

Now let us come to the task components in both the single-phase and multi-phase application models. With respect to data dependencies that are represented by the connection of task components to buffer components, the task components

are similar to the standard HSDF or standard CSDF components introduced in Section 4.7, with the difference that also here self-edges with one token prevent task (phase) executions before preceding executions have finished. On top of that, we also have to consider resource dependencies due to the processor sharing with SPP scheduling. For that matter we add input ports $p \in P^e$ modeling the instants at which higher priority tasks are enabled, as well as input ports $p \in P^f$ modeling the instants at which higher priority tasks finish their executions. Likewise, we add output ports $q \in Q^e$ and $q \in Q^f$ that express the enabling and finish times of the task for consideration within the lower priority tasks. With P^* and Q^* the sets of ports expressing data dependencies we can define the input and output interfaces of a task as $P = P^* \cup P^e \cup P^f$ and $Q = Q^* \cup Q^e \cup Q^f$.

For the modeling of data dependencies in our task components we use the same indexing pattern as for the dataflow actor components in Section 4.7. Just like in the actor relation we consequently do not use the index i to refer to stream locations in data streams, but to refer to task (phase) executions. To access data stream locations we further define the same port-specific indexing functions $i^p(i)$ and $i^q(i)$ as in Section 4.7 which translate the (phase) execution index i to the locations in the streams on ports p or q , respectively, that are accessed during execution i .

The single-phase tasks access exactly one location per execution in each of their input and output streams, such that stream locations equal execution indices, i.e.:

$$\forall_{i \geq 0, p \in P^*, q \in Q^*} : i^p(i) = i^q(i) = i$$

In contrast, multi-phase tasks can access zero to multiple locations per input or output stream in each phase execution, with the number of locations being phase-dependent. More precisely, let t_a^Λ be a multi-phase task with Θ phases, let γ_p^θ be the number of acquires on a port $p \in P^*$ in phase θ and let π_q^θ be the number of releases on a port $q \in Q^*$ in phase θ . Moreover, let the phase corresponding to a phase execution with index i be $\theta(i) = i \% \Theta$ (with $\%$ the modulo operator). Then we can define the indexing functions $i^p(i)$ and $i^q(i)$ for multi-phase tasks as follows:

$$\begin{aligned} \forall_{i \geq 0} : \forall_{p \in P^*} : i^p(i) &= \left\{ \sum_{i^*=0}^{i-1} \gamma_p^{\theta(i^*)} + i' \mid 0 \leq i' < \gamma_p^{\theta(i)} \right\} \wedge \\ \forall_{q \in Q^*} : i^q(i) &= \left\{ \sum_{i^*=0}^{i-1} \pi_q^{\theta(i^*)} + i' \mid 0 \leq i' < \pi_q^{\theta(i)} \right\} \end{aligned}$$

Using these formalisms we can express both single-phase and multi-phase tasks $t_a^\Lambda \in T^\Lambda$ in our component model with relations of the following form (with the potentially iteration-dependent execution time $\chi_a(i)$ of the task in isolation,

i.e. without interference of higher priority tasks, and with $f_a(-1) = 0$):

$$\begin{aligned}
 R_{t_a^\Delta} &= \{(X, Y) \in Tr(P) \times Tr(Q) \mid \forall_i: \forall_{p \in P}: \tau_{X[p]}(i) \leq \tau_{X[p]}(i+1) \wedge \quad (9.1) \\
 \varepsilon_a^{ext}(i) &= \max_{p \in P^*}(\tau_{X[p]}(i^p(i))) \wedge \varepsilon_a(i) = \max(\varepsilon_a^{ext}(i), f_a(i-1)) \wedge \\
 f_a(i) &= \varepsilon_a(i) + \chi_a(i) + \mathcal{I}_a[\varepsilon_a(i), f_a(i)] \wedge \check{\chi}_a(i) \leq \chi_a(i) \leq \hat{\chi}_a(i) \wedge \\
 \forall_{q \in Q^e}: \tau_{Y[q]}(i) &= \varepsilon_a(i) \wedge \forall_{q \in Q^f, q' \in Q^*}: \tau_{Y[q]}(i) = \tau_{Y[q']}(i^{q'}(i)) = f_a(i)\}
 \end{aligned}$$

The first line again ensures that only in-order inputs are accepted. The second line describes the computation of the external enabling time $\varepsilon_a^{ext}(i)$, which depends on the presence of both full and empty buffer locations (that are acquired at the enabling time $\varepsilon_a(i)$ of the respective task (phase) execution). The total enabling time $\varepsilon_a(i)$ additionally depends on the finish time of the preceding iteration, modeling self-delay. The finish times $f_a(i)$ of task (phase) executions are defined in line three, as the sums of enabling times $\varepsilon_a(i)$, the task (phase) execution times in isolation $\chi_a(i)$ and the interferences of higher priority tasks $\mathcal{I}_a[\varepsilon_a(i), f_a(i)]$ that occur between the enablings and finishes of iterations i . Such interference functions can be defined as follows (with hp_a the set of all higher priority tasks of task t_a^Δ on the same processor and $\varepsilon_b(j)$ and $f_b(j)$ shorthand notations for the timestamps in the enabling and finish time streams on the respective ports in P^e and P^f that are provided by a higher priority task t_b^Δ):

$$\mathcal{I}_a[\tau_0, \tau_1] = \max(0, \mathcal{I}_a[\tau_0, \varepsilon_b(j)] + [f_b(j) - \max(\tau_0, \varepsilon_b(j))] + \mathcal{I}_a[f_b(j), \tau_1])$$

$t_b^\Delta \in hp_a \wedge \varepsilon_b(j) < \tau_1 \wedge f_b(j) > \tau_0$

We formulate the higher priority interference in a recursive fashion, by splitting the interval between enabling and finish time into three parts, with the middle one expressing the largest amount of interference within the timespan between enabling and finish. The other two partial intervals express the remaining interference, i.e. the interference between task enabling and beginning of largest interference, as well as end of largest interference and task finish time. Note that for the highest priority task on each processor, the input interfaces P^e and P^f do not contain any ports, such that the interference term $\mathcal{I}_a[\varepsilon_a(i), f_a(i)]$ is always zero. Likewise, for the lowest priority task on each processor the output interfaces Q^e and Q^f can be removed. And also note that a finish time $f_a(i)$ depends on interference, which in turn depends on the finish time $f_a(i)$, resulting in an iterative fixed point computation.

The iteration-dependent execution time $\chi_a(i)$ is allowed to vary between the BCET $\check{\chi}_a(i)$ and the WCET $\hat{\chi}_a(i)$. For single-phase tasks we consider these bounds to be constants, i.e. $\forall_i: \check{\chi}_a(i) = \check{\chi}_a \wedge \hat{\chi}_a(i) = \hat{\chi}_a$. For multi-phase tasks, the BCETs and WCETs do not need to be constant for all phases, but only constant per phase, such that we define for a multi-phase task t_a^Δ with Θ phases that $\forall_{i,x=i \% \Theta}: \check{\chi}_a(i) = \check{\chi}_{a.x} \wedge \hat{\chi}_a(i) = \hat{\chi}_{a.x}$. Note that this formulation makes the application model non-deterministic in general as the uncertainty which execution time is exactly taken in each iteration is replaced by the non-determinism of a range of

execution times. In the last line of $R_{t_a^\Delta}$ the timestamps of the resource output streams on interface Q^e are assigned to the corresponding enabling times and the timestamps of the resource output streams on interface Q^f to the respective finish times, modeling the interference of task t_a^Δ on its lower priority tasks. Finally, the timestamps of the data output streams on interface Q^* are set to the finish times as well, modeling the releases of buffer locations.

With these components and the connections between components, which can be both data dependencies $d_{ab} \in D^D$ that connect data dependency ports with buffers, as well as resource dependencies $r_{ab} \in D^R$ that connect the resource dependency ports of tasks, all basic building blocks are given to express both single-phase and multi-phase application models in our timed component model.

Note that the presented application models are defined as being already both exclusion and inclusion abstractions of reality: On the one hand, many corner cases are excluded in the application models. For instance, it is assumed that all components are in a working state, such that errors, bugs and failures are not considered. And on the other hand, the uncertainty of iteration-dependent execution times is replaced by non-deterministic execution times that can vary between BCETs and WCETs, such that the presented task components include the uncertain task components after exclusion abstraction.

9.3 CONSTRUCTING ANALYSIS MODELS BY MEANS OF ABSTRACTION

As discussed in Section 8.1 and highlighted in Figure 8.2 we need to derive for our analysis approaches four different analysis models by means of inclusion, worst-case and best-case bounding abstraction. We go through the derivations of these models one-by-one, beginning with the response time analysis model, followed by worst-case and best-case analysis models and conclude with the precedence constraint analysis model. For clarification we also use the example depicted in Figure 9.2 that highlights the differences between all models.

9.3.1 RESPONSE TIME ANALYSIS MODELS

The first analysis model we introduce is the so-called response time analysis model used for the determination of maximum response times, which are defined for single-phase tasks as differences between the maximum external enabling and maximum finish times of task executions. For multi-phase tasks we compute maximum response times as differences between either external enabling or total enabling times and finish times of task phase executions, depending on whether the incoming self-edges of the respective phases are removed or not removed in the corresponding worst-case analysis model, which we define in the next section.

The maximum response times are not determined using the actual data and resource dependencies, but are instead based on intervals that restrict the input

Figure 9.2: An application model and its analysis model abstractions.

ports of task components. For that matter we need to add such input intervals to the task components in the response time analysis model and prove thereafter that the intervals do not restrict the streams coming from other task or buffer components.

A response time analysis model G^{RT} is mostly identical to the respective application model G^A , except that all task components are replaced with restricted task components which include aforementioned input intervals, as indicated by the red dots added to the input ports of tasks in Figure 9.2. This means that for an application model G^A consisting of one or more task graphs $G^{T,\Lambda} = (T^\Lambda, D^D, B, t_s^\Lambda)$ and of a set of resource dependencies D^R we define a response time analysis model G^{RT} as a combination of one or more task graphs $G^T = (T, D^D, B, t_s)$ and the same set of resource dependencies D^R .

We further assume that the source components in the response time analysis model are identical to the source components in the corresponding application model, i.e. $t_s = t_s^\Lambda$.

For the restricted task components $t_a \in T$ corresponding to a task component $t_a^\Lambda \in T^\Lambda$ let us denote with t_p the task connected to a resource input port p of t_a , with $\check{\epsilon}_p(i)$ and $\hat{\epsilon}_p(i)$ lower and upper bounds on the enabling times of such a task t_p in iteration i and with $\check{f}_p(i)$ and $\hat{f}_p(i)$ lower and upper bounds on the respective finish times. In addition, we denote with $\check{\epsilon}_a^{ext}(i)$ and $\hat{\epsilon}_a^{ext}(i)$ lower and upper bounds on the external enabling times of task t_a . With these formulations we can define the relations for all task components $t_a \in T$ as follows:

$$\begin{aligned} R_{t_a} = \{ & (X, Y) \in Tr(P) \times Tr(Q) \mid (X, Y) \in R_{t_a^\Lambda} \wedge \\ & \forall_i: \forall_{p \in P^e}: \check{\epsilon}_p(i) \leq \tau_{X[p]}(i) \leq \hat{\epsilon}_p(i) \wedge \forall_{p \in P^f}: \check{f}_p(i) \leq \tau_{X[p]}(i) \leq \hat{f}_p(i) \wedge \\ & \check{\epsilon}_a^{ext}(i) \leq \epsilon_a^{ext}(i) \leq \hat{\epsilon}_a^{ext}(i) \} \end{aligned} \quad (9.2)$$

The first line ensures that the restricted task component t_a has the same behaviors (the same pairs of input and output traces) as the corresponding task component t_a^Λ introduced in the previous section. The second line restricts the inputs on the resource dependency ports, taking into account that the streams on the ports in P^e reflect the enabling times of higher priority tasks, whereas the streams on the ports in P^f reflect the respective finish times. The last line finally restricts the external enabling times $\epsilon_a^{ext}(i)$ from Equation 9.1.

As depicted in Figure 9.2 we want to ensure that the response time analysis model is an inclusion abstraction of the corresponding application model. According to Theorem 5.10 this holds if both $G^{RT} \supseteq G^\Lambda$ and $in_{G^{RT}} \supseteq in_{G^\Lambda}$ apply. As we do not restrict the outputs of components given the same inputs the former condition always holds. The latter condition on the input sets, however, is only satisfied if all input intervals are chosen wide enough, such that any streams coming from higher priority tasks or connected buffers are always accepted (note that this is a consequence of the demonic interpretation in our component model described in Section 3.1.6). This means that for a valid inclusion abstraction the following schedule criterion must hold for all components $t_a \in T$:

$$\begin{aligned} \forall_i: \forall_{p \in P^e}: \check{\epsilon}_p(i) \leq \tau_{X[p]}(i) \leq \hat{\epsilon}_p(i) \wedge \forall_{p \in P^f}: \check{f}_p(i) \leq \tau_{X[p]}(i) \leq \hat{f}_p(i) \wedge \quad (9.3) \\ \check{\epsilon}_a^{ext}(i) \leq \epsilon_a^{ext}(i) \leq \hat{\epsilon}_a^{ext}(i) \end{aligned}$$

As we discuss later, such suitable input intervals can be determined using the worst-case and best-case analysis models introduced in the next section. Finally note that if the schedule criterion applies, then the response time analysis model does not only include the application model, but is in fact equivalent. As such, also all task components t_a are equivalent to the respective t_a^Λ in the context of their respective task graphs. For that reason we do not differ between task

components from the application and response time analysis models in the following, but always use t_a instead (except for discussions on the validity of the schedule criterion, where we need to highlight differences between application and analysis task components).

9.3.2 WORST-CASE & BEST-CASE ANALYSIS MODELS

Based on the response time analysis model we can now also construct the worst-case and best-case analysis models $G^{WC} = (\hat{V}, \hat{E})$ and $G^{BC} = (\check{V}, \check{E})$, respectively, as bounding abstractions of the former. The most apparent difference between a response time analysis model on the one hand and the corresponding worst-case and best-case analysis models on the other is that the latter neither contain resource dependencies nor self-edges with one token, as highlighted in Figure 9.2. Note that we do not construct worst-case and best-case analysis models as bounding abstractions of the application model directly because we require the input intervals of the response time analysis model to obtain useful abstractions after the dependency removal (for an in-depth discussion on this issue please refer to the case study presented in Section 6.5).

Data dependencies, i.e. the dependencies in the sets D^D , are simply copied as-is from the response time analysis model to the edge set $\hat{E}$ of the worst-case analysis model. Remember that FIFO buffers are nothing more but parallel compositions of two token components ($\beta_{a,b} = \kappa_{a,b} || \kappa_{b,a}$), as depicted in Figure 8.1. This allows us to draw each buffer $\beta_{a,b}$ in the application model with $\delta_{a,b}$ initially full locations and $\delta_{b,a}$ initially empty locations as two token components, one from task t_a to task t_b with $\delta_{a,b}$ initial tokens and another from task t_b to task t_a with $\delta_{b,a}$ initial tokens, and use the same token components in both response time and worst-case analysis models, as illustrated in Figure 9.2. As it further holds that token components are monotone with respect to the worst-case bounding operator $\trianglelefteq$ it follows with Theorem 5.3 that the token components worst-case bound themselves, i.e. $\kappa_{a,b} \triangleright \kappa_{a,b}$.

As already discussed in Section 8.1 and illustrated in Figure 9.2 we do not consider edges with tokens in the best-case model, which is equivalent to keeping such edges, but setting the numbers of initial tokens to infinity. Due to the differing numbers of initial tokens it holds that the token components in the best-case model are different from the token components in the other models. Based on the original token components from the application model we can construct best-case token components that can have two behaviors, either simply forwarding the input to the output stream in the case of zero tokens or producing the null stream y^0 in case of more than one token in the respective application model token component. The relation of such a best-case token component that best-case bounds a token component from the response time analysis model (and therewith also the corresponding component from the application model) is as

follows:

$$R_{\check{\kappa}_{a,b}} = \{(x, y) \in St(p) \times St(q) \mid x = x^\infty \Rightarrow y = y^\infty \wedge x \neq x^\infty \Rightarrow \\ \delta_{a,b} = 0 \Rightarrow y = x \wedge \delta_{a,b} > 0 \Rightarrow y = y^0\}$$

It is easy to see that $\check{\kappa}_{a,b} \leq \kappa_{a,b}$, irrespective of whether $\delta_{a,b}$ is zero or not: If $\delta_{a,b}$ is zero, then $\check{\kappa}_{a,b}$ is equal to $\kappa_{a,b}$. And if $\delta_{a,b}$ is larger than zero, then $\check{\kappa}_{a,b}$ always produces the null stream, which lower-bounds all other streams.

The source task components t_s are also copied one-to-one from response time analysis model to both worst-case and best-case analysis models, only that we call them source actor components v_s in the latter two models. As it further holds that source actor components v_s are monotone with respect to the worst-case and best-case bounding operators $\hat{\leq}$ and $\hat{\geq}$ it follows from Theorem 5.3 that also the source actor components worst-case and best-case bound the respective source task components, i.e. $v_s \hat{\geq} t_s$ and $v_s \hat{\leq} t_s$.

Lastly, we bound the restricted task components $t_a \in T$ in the response time analysis model by worst-case and best-case actors $\hat{v}_a \in \hat{V}$ and $\check{v}_a \in \check{V}$.

For single-phase worst-case and best-case models we bound the response time analysis model actors with self-edges and iteration-dependent firing durations by actors without any self-edges and with constant firing durations. With Corollary 4.8 we obtain the following relation (with $\check{v}_a$ being either $\hat{v}_a$ or $\check{v}_a$, $\bar{\rho}_a$ being either constant maximum firing durations $\hat{\rho}_a$ or minimum firing durations $\check{\rho}_a$, $\bar{\varepsilon}_a^{ext}(i)$ either maximum or minimum external enabling times $\hat{\varepsilon}_a^{ext}(i)$ or $\check{\varepsilon}_a^{ext}(i)$ and $\hat{f}_a(i)$ either maximum or minimum finish times $\hat{f}_a(i)$ or $\check{f}_a(i)$, respectively):

$$R_{\check{v}_a} = \{(X, Y) \in Tr(P^*) \times Tr(Q^*) \mid \forall_i: \forall_{p \in P^*}: \tau_{X[p]}(i) \leq \tau_{X[p]}(i+1) \wedge \quad (9.4) \\ \forall_{q \in Q^*}: \bar{\varepsilon}_a^{ext}(i) = \max_{p \in P^*}(\tau_{X[p]}(i^p(i))) \wedge \tau_{Y[q]}(i^q(i)) = \hat{f}_a(i) = \bar{\varepsilon}_a^{ext}(i) + \bar{\rho}_a\}$$

We require that the best-case (worst-case) analysis models are valid best-case (worst-case) abstractions of the response time analysis model, as shown in Figure 9.2. According to Theorem 5.4 this holds if both $G^{BC} \hat{\leq} G^{RT}$ ($G^{WC} \hat{\geq} G^{RT}$) and $in_{G^{BC}} \supseteq in_{G^{RT}}$ ($in_{G^{WC}} \supseteq in_{G^{RT}}$) apply. The conditions on input sets are always satisfied, due to the input completeness of dataflow components (except for the source components, which are the same in all models). In contrast, the graph bounding conditions only hold if best-case (worst-case) bounding applies for all components. This means that next to the bounding of source and token components, which we already discussed before, also the actor components $\check{v}_a$ and $\hat{v}_a$ according to Equation 9.4 must best-case or worst-case bound the corresponding components t_a in the response time analysis model, i.e. $\check{v}_a \hat{\leq} t_a$ and $\hat{v}_a \hat{\geq} t_a$. By definition this is the case if bounding inputs result in bounding outputs. Given bounding inputs it must hence hold:

$$\forall_i: \check{f}_a(i) \leq f_a(i) \leq \hat{f}_a(i) \quad (9.5)$$

Substituting the finish times using the equations from within the respective relations results in the following single-phase firing duration criterion that must apply for the best-case and worst-case analysis models to be valid bounding abstractions of the response time analysis model:

$$\forall_i: \hat{\varepsilon}_a^{ext}(i) + \check{\rho}_a \leq \varepsilon_a(i) + \chi_a(i) + \mathcal{I}_a[\varepsilon_a(i), f_a(i)] \leq \hat{\varepsilon}_a^{ext}(i) + \hat{\rho}_a \quad (9.6)$$

Given that $\forall_i: \hat{\varepsilon}_a^{ext}(i) \leq \varepsilon_a(i)$ we can obtain a minimum firing duration that matches this criterion by simply setting it to the respective BCET that lower-bounds all iteration-dependent execution times, i.e. $\check{\rho}_a = \check{\chi}_a$. A valid maximum firing duration $\hat{\rho}_a$ is more difficult to determine due to its dependence on self-delay (the enabling time $\varepsilon_a(i)$ can be larger than $\hat{\varepsilon}_a^{ext}(i)$ in the case of self-delay) as well as due to interference from higher priority tasks. We make use of the response time analysis models discussed in the previous section to obtain this bound.

For the multi-phase actors in the best-case and worst-case models we do not remove all self-edges, which ensures that the internal order between the different phases of a task is maintained. In addition, in Chapter 12 we do not analyze the resulting CSDF graphs directly, but expand these graphs to equivalent HSDF graphs. It can occur that the phases of a CSDF actor $\bar{v}_a$ do not appear only once in the HSDF expansion, but are replicated r times, such that the total number of phases is not Θ , but $\Theta' = r \cdot \Theta$ instead (for more information on the HSDF expansion please refer to Section 12.2).

With that in mind, we only remove every Θ' 'th self-edge. Moreover, we do not consider all firing durations as constant, but as constant per repeated phase, which results in Θ' potentially different minimum and maximum firing durations. Taking these aspects into account we define the following relation according to Corollary 4.8 (with $\bar{v}_a$ being either $\hat{v}_a \in \hat{V}$ or $\check{v}_a \in \check{V}$, $\Theta' = r \cdot \Theta$ the number of phases in the HSDF expansion, $\bar{\rho}_a(i)$ being either per-repeated-phase-constant maximum firing durations $\hat{\rho}_{a,x}$ or minimum firing duration $\check{\rho}_{a,x}$, such that $\forall_i: \bar{\rho}_a(i + \Theta') = \bar{\rho}_a(i)$, $\bar{\varepsilon}_a^{ext}(i)$ either maximum or minimum external enabling times $\hat{\varepsilon}_a^{ext}(i)$ or $\check{\varepsilon}_a^{ext}(i)$ and $\bar{f}_a(i)$ either maximum or minimum finish times $\hat{f}_a(i)$ or $\check{f}_a(i)$, respectively):

$$R_{\bar{v}_a} = \{(X, Y) \in Tr(P^*) \times Tr(Q^*) \mid \forall_i: \forall_{p \in P^*}: \tau_{X[p]}(i) \leq \tau_{X[p]}(i+1) \wedge \quad (9.7)$$

$$\forall_{q \in Q^*}: i \% \Theta' = 0 \Rightarrow \bar{\varepsilon}_a^{ext}(i) = \max_{p \in P^*}(\tau_{X[p]}(i^p(i))) \wedge$$

$$\tau_{Y[q]}(i^q(i)) = \bar{f}_a(i) = \bar{\varepsilon}_a^{ext}(i) + \bar{\rho}_a(i) \wedge$$

$$i \% \Theta' \neq 0 \Rightarrow \bar{\varepsilon}_a(i) = \max_{p \in P^*}(\tau_{X[p]}(i^p(i)), \bar{f}_a(i-1)) \wedge$$

$$\tau_{Y[q]}(i^q(i)) = \bar{f}_a(i) = \bar{\varepsilon}_a(i) + \bar{\rho}_a(i)\}$$

Multi-phase components of this form best-case or worst-case bound a task component t_a in the response time analysis model (and therewith also t_a^Λ in the application model if the schedule criterion in Equation 9.3 is satisfied) if bounding inputs

result in bounding outputs. This is the case if Equation 9.5 applies for bounding inputs. Substituting the finish times results in the following multi-phase firing duration criterion:

$$\begin{aligned}
 \forall_i: i \% \Theta^r = 0 &\Rightarrow \\
 \check{\varepsilon}_a^{ext}(i) + \check{\rho}_a(i) &\leq \varepsilon_a(i) + \chi_a(i) + \mathcal{I}_a[\varepsilon_a(i), f_a(i)] \leq \hat{\varepsilon}_a^{ext}(i) + \hat{\rho}_a(i) \wedge \\
 i \% \Theta^r \neq 0 &\Rightarrow \\
 \check{\varepsilon}_a(i) + \check{\rho}_a(i) &\leq \varepsilon_a(i) + \chi_a(i) + \mathcal{I}_a[\varepsilon_a(i), f_a(i)] \leq \hat{\varepsilon}_a(i) + \hat{\rho}_a(i)
 \end{aligned} \tag{9.8}$$

Also here it holds that if $\forall_i: \check{\varepsilon}_a^{ext}(i) \leq \varepsilon_a(i) \wedge \check{\varepsilon}_a(i) \leq \varepsilon_a(i)$ we can rather simply obtain minimum phase-constant firing durations that match this criterion by assigning them the respective phase-constant BCETs, i.e. $\forall_{i,x=i \% \Theta}: \check{\rho}_a(i) = \check{\chi}_{a,x}$. Obtaining upper bounds that satisfy the criterion is more complex and subject to Chapter 12.

If the minimum and maximum firing durations adhere to the presented firing duration criteria it is easy to see that the respective actor components $\bar{v}_a$ best-case and worst-case bound the components t_a from the response time analysis model, i.e. $\check{v}_a \trianglelefteq t_a$ and $\hat{v}_a \trianglerighteq t_a$. With the bounding of source and token components it follows that the best-case and worst-case analysis models presented in this section are valid best-case and worst-case abstractions of the response time analysis model if the firing duration criteria hold. And if additionally the schedule criterion from Equation 9.3 holds as well, we can further conclude that the best-case and worst-case analysis models also bound the respective application model, i.e. all abstractions depicted in Figures 8.2 and 9.2 are valid on satisfaction of both criteria.

9.3.3 PRECEDENCE CONSTRAINT ANALYSIS MODELS

The last analysis model we construct is the so-called precedence constraint analysis model that allows to analyze precedence constraints between tasks. In the following we use precedence constraints to reduce the maximum response times that are based on input intervals only, which results in a significant increase of analysis accuracy.

The precedence constraint analysis model is equal to the underlying response time analysis model, except that all resource dependencies are removed. Due to this removal the precedence constraint analysis model becomes an HSDF model for the case of a single-phase application and a CSDF model for a multi-phase application, as exemplified in Figure 9.2. As such, we can denote the precedence constraint model as a dataflow graph $G^{PC} = (V, E)$ for which data dependencies, token and source components are simply copied from the response time analysis model. The actor components $v_a \in V$ can be defined as follows:

$$\begin{aligned}
 R_{v_a} &= \{(X^*, Y^*) \in Tr(P^*) \times Tr(Q^*) \mid \\
 &\quad \exists_{X^\diamond \in P^e \cup P^f, Y^\diamond \in Q^e \cup Q^f}: (X^* \cup X^\diamond, Y^* \cup Y^\diamond) \in R_{t_a}\}
 \end{aligned} \tag{9.9}$$

Obviously it holds for all such actor components that they include the corresponding restricted task components from the response time analysis model, i.e. $v_a \supseteq t_a$. With the self-inclusion of all other components we can thus conclude that the precedence constraint analysis model is a valid inclusion abstraction of the respective response time analysis model, and that unconditionally.

Period-and-Jitter Analysis

After having introduced the application models that define the context, as well as having constructed the required analysis models by means of abstraction we are now ready to present our first analysis technique: The period-and-jitter analysis for single-phase applications.

The remainder of this chapter is structured as follows: First we introduce two different analysis flows making use of period-and-jitter analysis in Section 10.1, one with a post-analysis buffer sizing and the other with an iterative buffer sizing. In Section 10.2 we discuss the determination of bounds on task schedules using the best-case and worst-case analysis models. Section 10.3 describes the computation of maximum task response times (the differences between maximum external enabling and maximum finish times), including execution times, self-delay and higher-priority interference based on task periods and jitters. Besides period-and-jitter, we also consider an alternative way to bound interference, based on cyclic data dependencies between tasks. We show that combining the two methods results in a higher analysis accuracy. In Section 10.4 we elaborate the determination of sufficiently large buffer capacities based on analysis results. Moreover, we discuss the iterative buffer sizing in more detail, which can be used to optimize applications by modifying buffer sizes for meeting throughput constraints. Section 10.5 presents a further extension of our iterative buffer sizing analysis flow that involves an optimization of task schedules via the insertion of so-called synchronization edges. In Section 10.6 we prove that our analysis flows derive bounds on task schedules and maximum response times that are conservative to the actual schedules and response times occurring in the respective application model. Section 10.7 finally evaluates our period-and-jitter based approaches in a case study.

10.1 ANALYSIS FLOWS

In this section we present two different analysis flows, first one that assumes buffer capacities either as given or to be determined after analysis and thereafter another which incorporates an iterative buffer sizing. In the latter, buffer sizes are determined alongside analysis, which can result in smaller buffer capacities, higher throughput guarantees, as well as an overall more accurate analysis.

10.1.1 ANALYSIS WITH POST-ANALYSIS BUFFER SIZING

Figure 10.1 depicts the analysis flow of our temporal analysis approach with a post-analysis buffer sizing. We use this analysis flow for the verification of throughput and latency constraints. Moreover, if buffer capacities are not defined, we can also use this flow for the calculation of sufficiently large buffer capacities.

Figure 10.1: Overview of the analysis flow with a post-analysis buffer sizing.

Inputs to this analysis flow are the single-phase analysis models that are derived from a single-phase application according to the methods described in Chapter 9. In the HSDF graphs representing the best-case, worst-case and precedence constraint analysis models, the actors and edges connecting the actors are fixed, whereas the minimum and maximum firing durations in the best-case and worst-case analysis models are initially set to the respective lower and upper bounds of task execution times, i.e. the BCETs and WCETs. In later flow iterations, the maximum firing durations are set to the maximum response times computed with the response time analysis model in step 2. Moreover, if the analysis flow is to be used for determination of buffer capacities, the respective capacities (more precisely, the numbers of empty buffer locations) are set to infinity during analysis. In the response time analysis model the tasks, buffers and resource dependencies (which are derived from the task-to-processor mappings and task priorities) are fixed, whereas the input intervals, i.e. the lower and upper bounds on (external) enabling and finish times are variably assigned in each iteration of the analysis flow.

Given the best-case and worst-case HSDF graphs, two periodic schedules are computed in step 1, the first a lower bound on the best-case schedule and the second an upper bound on the worst-case schedule. Using these schedules as input intervals in the response time analysis model, maximum enabling jitters of tasks are derived in step 2. And based on these maximum jitters, as well as cyclic data dependencies between tasks obtained from the precedence constraint model, maximum response times of tasks are determined that are valid upper bounds with respect to the response time analysis model. In step 3, the best-case and worst-case schedules are checked against the temporal constraints and it is verified whether all maximum response times have converged, i.e. did not change since the previous iteration of the flow. If a constraint is violated then the algorithm stops. Otherwise, depending on whether the maximum response times have converged, the algorithm either continues with step 4, or repeats the steps 1 to 3 until either convergence or constraint violation is observed.

Lastly, if buffer capacities are not given beforehand, but are assumed as infinite during analysis, we can determine sufficiently large buffer capacities after convergence in step 4, based on the schedules computed in the iterative part of the analysis flow.

We discuss in Appendix B.1 that termination (i.e. convergence or constraint violation) of the analysis flow can be guaranteed if the iteratively determined maximum response times are in some way bounded from above. One method to achieve this is to define upper bounds on end-to-end latencies for all task graphs of an analyzed application that can be used as temporal constraints in step 3.

10.1.2 ANALYSIS WITH ITERATIVE BUFFER SIZING

Figure 10.2 depicts the analysis flow of our temporal analysis approach with an iterative buffer sizing. The main difference to the previous analysis flow is that buffer sizing of unknown buffer capacities is not conducted post-analysis, but repeated within the iterative loop of the flow, such that estimates on buffer capacities are computed in each flow iteration.

Figure 10.2: Overview of the analysis flow with an iterative buffer sizing.

The impact of the iterative buffer sizing on the different analysis flow steps is as follows. First, we do not use iteratively computed estimates on buffer capacities in the worst-case model analyzed in step 1, which is required to ensure flow termination. In step 2, which previously occurred after the iterative part of the analysis flow, the estimates on buffer capacities are computed based on the schedules determined in the previous step. We have to take care that the iterative sizing of blocking write buffers does not break flow convergence. We ensure this by preventing decreasing buffer capacities from one flow iteration to the next. In step 3, the estimates are used in both response time and precedence constraint analysis models. This allows for a consideration of the resulting cyclic data dependencies to limit interference and therewith for the computation of tighter maximum response times.

In Appendix B.1 we prove that the iterative buffer sizing analysis flow also terminates if instead of maximum latencies upper bounds on all iteratively determined buffer capacities are specified and subsequently verified in step 4 of the flow.

10.2 DETERMINING BOUNDS ON TASK SCHEDULES

A method to derive periodic bounds on the schedules, i.e. the external enabling times, total enabling times and finish times of tasks is presented in this section. To compute such bounds we use best-case and worst-case analysis models as defined in Section 9.3.2. As we are considering single-phase applications in this chapter, it holds that both best-case and worst-case models are HSDF graphs. We use the computed bounds on schedules as input intervals in the response time analysis model. Based on these intervals we determine maximum enabling jitters of tasks, as well as buffer capacities that allow for these schedules.

According to Section 9.3.2 we bound the tasks from the response time analysis model with dataflow actors in both best-case and worst-case analysis models. For the best-case model we consider that cyclic data dependencies with positive, but finite numbers of initial tokens can delay periodic start times of actors, while it can occur that the corresponding tasks do not always experience the same delays. Consequently, we assume all positive and finite numbers of initial tokens to be infinite in the best-case model, which is equivalent to removing all edges containing tokens. In the worst-case model, we consider all buffer capacities that are fixed before analysis as cyclic data dependencies, as shown in Figure 9.2. Buffer capacities that are determined with our analysis flows are set to infinity, irrespective of whether the flow with a post-analysis or with an iterative buffer sizing is applied, to ensure both conservativeness and termination of the flows. The firing durations of actors in the best-case dataflow model are set to the BCETs of the corresponding tasks, while the firing durations of actors in the worst-case model are initially set to the respective WCETs and in later flow iterations to the maximum response times that are obtained from the response time analysis model according to Section 10.3. Using the best-case and worst-case dataflow

models we compute two periodic schedules consisting of the respective best-case and worst-case start times of actors.

In [HWGB13] it has been shown that the following LP can be used to derive a periodic schedule of actor start times for the best-case model (with $\check{V}$ the set of actors in the best-case analysis model and $\check{E}$ the respective set of edges, which includes all data dependencies from the response time analysis model that do not contain initial tokens):

$$\begin{aligned} \text{Minimize: } & \sum_{\check{v}_a \in \check{V}} \check{s}_a \\ \text{Subject to: } & \check{s}_s = 0, \forall_{e_{a,b} \in \check{E}}: \check{s}_b - \check{s}_a \geq \check{\rho}_a \end{aligned} \quad (10.1)$$

By setting the start time of the source actor v_s to $\check{s}_s = 0$ all start times are computed relative to the start time of the source. As the enabling times of tasks in the response time analysis model are also relative to the first execution of the strictly periodic source t_s , we can use the minimum start time $\check{s}_a$ of an actor $\check{v}_a$ to derive a lower bounds $\check{\epsilon}_a^{ext}(i)$ on the external enabling times $\epsilon_a^{ext}(i)$ of the task t_a in the response time analysis model as follows:

$$\forall_{i \geq 0}: \check{\epsilon}_a^{ext}(i) = \check{s}_a + i \cdot P_a \leq \epsilon_a^{ext}(i)$$

As task executions cannot be enabled earlier than at their minimum external enabling times, we can use the same bounds as lower bounds on the total enabling times $\epsilon_a(i)$ of task t_a as well, such that:

$$\forall_{i \geq 0}: \check{\epsilon}_a(i) = \check{s}_a + i \cdot P_a \leq \epsilon_a(i)$$

Analogously, a periodic schedule consisting of the start times of actors in the worst-case model can be computed by solving the following LP (with $\hat{V}$ and $\hat{E}$ the sets of actors and edges in the worst-case analysis model):

$$\begin{aligned} \text{Minimize: } & \sum_{\hat{v}_a \in \hat{V}} \hat{s}_a \\ \text{Subject to: } & \hat{s}_s = 0, \forall_{e_{a,b} \in \hat{E}}: \hat{s}_b - \hat{s}_a \geq \hat{\rho}_a - \delta_{a,b} \cdot P_a \end{aligned} \quad (10.2)$$

We use the maximum start time $\hat{s}_a$ of an actor $\hat{v}_a$ to determine upper bounds $\hat{\epsilon}_a^{ext}(i)$ on the external enabling times $\epsilon_a^{ext}(i)$ of a task t_a in the response time analysis model:

$$\forall_{i \geq 0}: \epsilon_a^{ext}(i) \leq \hat{\epsilon}_a^{ext}(i) = \hat{s}_a + i \cdot P_a$$

Using the determined periodic bounds on external enabling times and the respective constant best-case and worst-case firing durations we can also determine periodic bounds on the finish times of a task t_a as follows:

$$\forall_{i \geq 0}: \check{f}_a(i) = \check{s}_a + \check{\rho}_a + i \cdot P_a \leq f_a(i) \leq \hat{f}_a(i) = \hat{s}_a + \hat{\rho}_a + i \cdot P_a$$

What remains is the need to determine a periodic upper bound on the total enabling times of task executions. For that matter we consider that every task execution in iteration i is either enabled externally at its external enabling time or internally at the finish time of its preceding iteration $i-1$. We define $f_a(-1) = -\infty$ and $\hat{f}_a(-1) = \hat{s}_a + \hat{p}_a - P_a$, thus it also holds that $f_a(-1) \leq \hat{f}_a(-1)$. A periodic upper bound $\hat{e}_a(i)$ on the enabling times $e_a(i)$ of a restricted task t_a in the response time analysis model can then be determined as follows:

$$\begin{aligned} \forall_{i \geq 0}: e_a(i) &\leq \hat{e}_a(i) = \max(\hat{e}_a^{ext}(i), \hat{f}_a(i-1)) \\ &= \max(\hat{s}_a + i \cdot P_a, \hat{s}_a + \hat{p}_a + (i-1) \cdot P_a) \\ &= \hat{s}_a + \max(0, \hat{p}_a - P_a) + i \cdot P_a \end{aligned}$$

With that we have derived all required bounds on (external) enabling and finish times of tasks that we use as input intervals of tasks in the response time analysis model according to Section 9.3.1. Using these input intervals we determine maximum response times, as presented in the next section.

10.3 COMPUTING MAXIMUM RESPONSE TIMES

In this section we first introduce a maximum response time calculation that makes use of period-and-jitter to determine higher priority interference. Thereafter we describe another way to limit interference, using cyclic data dependencies between tasks. Finally we discuss the combination of this technique with period-and-jitter to compute tighter maximum response times.

10.3.1 CHARACTERIZING INTERFERENCE USING PERIOD-AND-JITTER

Following the requirement imposed by the firing duration criterion in Equation 9.6 we define the maximum response time of a task t_a in the response time analysis model as an upper bound on the differences between maximum external enabling and maximum finish times of that task. Consequently, a maximum response time needs to subsume an upper bound on the execution time of the task, i.e. its WCET, any self-delay, as well as an upper bound on its higher priority interference. As the considered SPP scheduler belongs to the class of non-starvation-free schedulers [WBS09] we need to determine enabling rate characterizations of all higher priority tasks in order to bound interference.

In the analysis approaches presented in this chapter we use the periods P_b of higher priority tasks t_b in conjunction with upper bounds on their enabling jitters $\hat{J}_b$ to determine such interference characterizations. From the input intervals that are added to the tasks in the response time analysis model we can obtain lower and upper bounds on the enabling times of all higher priority tasks. According to our iterative analysis flows these bounds are set to the respective periodic bounds determined in the previous section. This allows us to compute the maximum

enabling jitters $\hat{J}_b$ of higher priority tasks t_b in the response time analysis model as differences between periodic minimum and maximum enabling times, such that:

$$\hat{J}_b = \max_{i \geq 0} (\hat{\epsilon}_b(i) - \check{\epsilon}_b(i)) = \hat{s}_b + \max(0, \hat{\rho}_b - P_b) - \check{s}_b \quad (10.3)$$

Using such maximum jitters we can determine the interference characterization $\eta_b(\Delta\tau)$ of a task t_b , which provides an upper bound on the number of enablings that task t_b can have during any time interval $\Delta\tau$, as follows [RRE03]:

$$\eta_b(\Delta\tau) = \left\lceil \frac{\hat{J}_b + \Delta\tau}{P_b} \right\rceil \quad (10.4)$$

With this interference characterization we can compute the q -level maximum busy period $\hat{w}'_a(q)$ of a task t_a , an upper bound on the time between the first enabling and last finish of $q+1$ consecutive executions of a task t_a , by the following iterative equation [TBW94]:

$$\hat{w}'_a(q) = (q+1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \eta_b(\hat{w}'_a(q)) \cdot \hat{\chi}_b \quad (10.5)$$

$\hat{\chi}_a$ denotes the WCET of one execution of a task t_a and the set hp_a contains all tasks t_b that are mapped to the same processor as task t_a and have a higher priority. Provided that external enablings of a task t_a are periodic with a period P_a , its maximum response time $\hat{R}'_a$ can be derived as follows (for an illustration please refer to Figure 8.4):

$$\hat{R}'_a = \max_{q \geq 0} (\hat{w}'_a(q) - q \cdot P_a) \quad (10.6)$$

Besides $q = 0$ only values of q for which $\hat{w}'_a(q-1) \geq q \cdot P_a$ holds need to be considered [TBW94].

Note that except for the WCETs of the task t_a and its higher priority tasks all other parameters to compute maximum response times can be obtained from the input intervals of t_a in the response time analysis model. Consequently we do not need to consider neither data nor resource dependencies explicitly and can compute maximum response times of all tasks locally, i.e. without requiring a holistic analysis. Moreover, the maximum response times determined in this section determine interference based on the input intervals of the respective tasks. As long as these intervals are not wide enough to accommodate all streams that can occur in the underlying application model, it holds that the maximum response times are only valid for the response time and worst-case analysis models, but not yet for the application model. Only once the schedule criterion in Equation 9.3 is satisfied, which occurs on convergence of our analysis flows, the determined maximum response times become valid for the application model as well.

Later we prove that maximum response times determined as described in this section can be used as firing durations in the worst-case analysis model because

they adhere to the firing duration criterion in Equation 9.6 and as such form conservative upper bounds on task execution times, self-delay and higher priority interference that can occur in the response time analysis model. Before we provide that proof, however, let us first discuss an alternative way to limit interference than with period-and-jitter.

10.3.2 LIMITING INTERFERENCE USING CYCLIC DATA DEPENDENCIES

In this section we discuss another method to characterize higher priority interference that is based on cyclic data dependencies. For that matter we derive a function $\zeta_{b \rightarrow a}(q)$ that returns the maximum number of enablings a task t_b can have during $q + 1$ consecutive executions of a task t_a , according to cyclic data dependencies between the two tasks. We do not base this derivation on the response time analysis model, but use the corresponding HSDF graph from the precedence constraint analysis model instead. This is allowed as both models have the same structure, except for the lack of resource dependencies in the latter. Hence we also do not refer to tasks in the remainder of this section, but to the HSDF actors corresponding to tasks.

We use precedence constraints on firings of actors in the derivation of $\zeta_{b \rightarrow a}(q)$, which are defined as follows:

Definition 50 (Precedence Constraint). *We define $v_b(j)$ as firing j of an actor v_b . If a firing $v_b(j)$ of an actor v_b cannot start before a firing $v_a(i)$ of an actor v_a has finished, then we say that $v_a(i)$ precedes $v_b(j)$ and denote this relation as $v_b(j) > v_a(i)$. The firing number i is defined in a sequential manner such that a firing $j > i$ of an actor v_b cannot start before its firing i has started. $v_a(0)$ denotes the first firing of an actor v_a .*

According to the definition of HSDF, an actor can only fire if there is at least one token on each of its incoming edges. Moreover, an actor produces one token on each of its outgoing edges on every completed firing. If an edge $e_{a,b}$ contains $\delta_{a,b}$ initial tokens then actor v_b can fire $\delta_{a,b}$ times before it must get enabled by a completed firing of actor v_a for a subsequent firing. This implies that firing i of actor v_b cannot start before firing $i - \delta_{a,b}$ of actor v_a has finished, which is captured by the following precedence constraint:

$$\forall i \geq \delta_{a,b} : v_b(i) > v_a(i - \delta_{a,b}) \quad (10.7)$$

Definition 51 (Path). *A directed path in a precedence constraint analysis model G^{PC} is defined as a sequence of edges:*

$$p_{a,b} = \langle e_0, e_1, e_2, \dots, e_{|p_{a,b}|-1} \rangle$$

We require that $e_k = (v_k, v_{k+1}) \in E$ and call $|p_{a,b}|$ the number of edges on a path $p_{a,b}$. We speak of a path from an actor v_a to an actor v_b if $e_0 = (v_a, v_1)$ and $e_{|p_{a,b}|-1} = (v_{|p_{a,b}|-1}, v_b)$. The set $\mathcal{P}_{a,b}$ contains all paths from an actor v_a to an actor v_b .

Definition 52 (Path Length). The number of initial tokens on a path p is defined as the sum of initial tokens on its edges:

$$\delta(p_{a,b}) = \sum_{k=0}^{|p_{a,b}|-1} \delta(e_k)$$

The minimum number of initial tokens on a path from an actor v_a to an actor v_b is defined as follows:

$$\delta(\mathcal{P}_{a,b}) = \begin{cases} \min_{p_{a,b} \in \mathcal{P}_{a,b}} \delta(p_{a,b}) & \text{if } \mathcal{P}_{a,b} \neq \emptyset \\ \infty & \text{otherwise} \end{cases} \quad (10.8)$$

Figure 10.3: Paths in a cyclic HSDF graph.

The relation between edges and paths is illustrated by the HSDF graph depicted in Figure 10.3, which contains one path from actor v_a to actor v_b and two paths from actor v_b to actor v_a . The minimum number of initial tokens on paths from actor v_a to actor v_b is $\delta(\mathcal{P}_{a,b}) = 0$, due to $\delta_{a,d} + \delta_{d,b} = 0$, whereas the minimum number of initial tokens on paths from actor v_b to actor v_a equals to $\delta(\mathcal{P}_{b,a}) = \min(\delta_{b,d} + \delta_{d,a}, \delta_{b,a}) = 2$. Moreover, consider the dotted edge $e_{a,c}$ from actor v_a to actor v_c . As $\delta_{a,c} = \infty$, there are always sufficient tokens on this edge for actor v_c to fire, independent of firings of actor v_a . Due to this independence of firings, an edge with an infinite number of initial tokens can be considered equivalent to the absence of that edge. Consequently, a non-existent path between two actors can also be considered equivalent to a path containing a single edge with an infinite number of initial tokens. This equivalence is reflected by the definition of $\delta(\mathcal{P}_{a,b}) = \infty$ for $\mathcal{P}_{a,b} = \emptyset$.

The Floyd-Warshall algorithm [Flo62] can be used to compute minimum distances between all pairs of vertices of a weighted directed graph. We apply this algorithm on the HSDF graph forming the precedence constraint analysis model, considering actors as vertices, using the same edges between vertices as between actors and setting the required edge weights to numbers of initial tokens. The resulting minimum distances between all pairs of vertices then just equal to $\delta(\mathcal{P}_{a,b})$ for all pairs of actors in the HSDF graph. Note that in our case edge weights cannot be negative, as an HSDF graph cannot contain edges with negative numbers of initial tokens. Hence there also cannot be negative cycles in the graph processed by the Floyd-Warshall Algorithm, guaranteeing that the correct $\delta(\mathcal{P}_{a,b})$ can be always obtained.

So far we have only defined precedence constraints for single edges. Proving the following lemma allows us to use precedence constraints for paths as well.

Lemma 10.1 (Path Precedence). *For two actors $v_a, v_b \in V$ it holds that the following precedence constraint is the tightest precedence constraint imposed by any path from actor v_a to actor v_b :*

$$\forall_{i \geq \delta(\mathcal{P}_{a,b})}: v_b(i) > v_a(i - \delta(\mathcal{P}_{a,b}))$$

Proof. All edges of a path $p_{a,b} \in \mathcal{P}_{a,b}$ from an actor v_a to an actor v_b impose precedence constraints as defined in Equation 10.7. By recursively substituting the firing numbers of the actors on such a path we derive the following relation:

$$\begin{aligned} v_b(i) &= v_{|p_{a,b}|}(i) > v_{|p_{a,b}|-1}(i - \delta(e_{|p_{a,b}|-1})) \\ &> v_{|p_{a,b}|-2}(i - \delta(e_{|p_{a,b}|-1}) - \delta(e_{|p_{a,b}|-2})) \\ &> \dots \\ &> v_0(i - \sum_{k=0}^{|p_{a,b}|-1} \delta(e_k)) = v_a(i - \delta(p_{a,b})) \end{aligned} \quad (10.9)$$

Due to the definition of $\delta(\mathcal{P}_{a,b})$, there must be a path $p_{a,b} \in \mathcal{P}_{a,b}$ with $\delta(p_{a,b}) = \delta(\mathcal{P}_{a,b})$. As $\forall_{p_{a,b} \in \mathcal{P}_{a,b}}: \delta(\mathcal{P}_{a,b}) \leq \delta(p_{a,b})$, it holds that no path from an actor v_a can impose a tighter precedence constraint on the firings of actor v_b than a path $p_{a,b} \in \mathcal{P}_{a,b}$ with $\delta(p_{a,b}) = \delta(\mathcal{P}_{a,b})$. By substituting $\delta(p_{a,b})$ in Equation 10.9 with $\delta(\mathcal{P}_{a,b})$ we obtain the precedence constraint from Lemma 10.1. $\square$

Based on Definition 52 and the definition of HSDF graphs we define sets of firings of actors that can overlap in time with a single firing of an actor v_a .

Definition 53 (Outgoing Precedence Set). *The outgoing precedence set $P_{v_b \rightarrow v_a(i)}^{\text{out}}$ contains all firings of an actor v_b that can overlap in time with firing i of an actor v_a , despite the precedence constraints imposed by paths from actor v_a to actor v_b .*

From Lemma 10.1 we derive:

$$\forall_{i \geq \delta(\mathcal{P}_{a,b})}: v_b(i) > v_a(i - \delta(\mathcal{P}_{a,b})) \Leftrightarrow \forall_{i \geq 0}: v_b(i + \delta(\mathcal{P}_{a,b})) > v_a(i)$$

This implies together with the sequentiality of actor firings that only firings $v_b(j)$ with $j < i + \delta(\mathcal{P}_{a,b})$ can occur before the end of firing $v_a(i)$. With this observation we can formalize Definition 53 as follows:

$$P_{v_b \rightarrow v_a(i)}^{\text{out}} = \{j \mid j < i + \delta(\mathcal{P}_{a,b})\}$$

Analogous to the outgoing precedence set we define an precedence set for incoming paths of actor v_a :

Definition 54 (Incoming Precedence Set). *The incoming precedence set $P_{v_b \rightarrow v_a(i)}^{\text{in}}$ contains all firings of an actor v_b that can overlap in time with firing i of an actor v_a , despite the precedence constraints imposed by paths from actor v_b to actor v_a .*

By using Lemma 10.1 we obtain:

$$\forall_{i \geq \delta(\mathcal{P}_{b,a})}: v_a(i) > v_b(i - \delta(\mathcal{P}_{b,a}))$$

From this follows with the sequentiality of actor firings that only firings $v_b(j)$ with $j > i - \delta(\mathcal{P}_{b,a})$ can occur after the start of firing $v_a(i)$. This allows us to construct the formalization of $P_{v_b \rightarrow v_a(i)}^{\text{in}}$:

$$P_{v_b \rightarrow v_a(i)}^{\text{in}} = \{j \mid j > i - \delta(\mathcal{P}_{b,a})\}$$

As we are not only interested in the firings of an actor v_b that can take place during a single firing of an actor v_a , but the firings that can take place during $q + 1$ consecutive firings, we can derive precedence sets for $q + 1$ consecutive firings as unions of single firing precedence sets:

$$P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}^{\text{out}} = \bigcup_{i'=i}^{i+q} P_{v_b \rightarrow v_a(i')}^{\text{out}} = \{j \mid j < i + q + \delta(\mathcal{P}_{a,b})\}$$

$$P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}^{\text{in}} = \bigcup_{i'=i}^{i+q} P_{v_b \rightarrow v_a(i')}^{\text{in}} = \{j \mid j > i - \delta(\mathcal{P}_{b,a})\}$$

To determine all firings of an actor v_b that can occur during $q + 1$ consecutive firings of an actor v_a , we draw the intersection between outgoing and incoming precedence sets:

$$P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}} = P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}^{\text{out}} \cap P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}^{\text{in}}$$

$$= \{j \mid i - \delta(\mathcal{P}_{b,a}) < j < i + q + \delta(\mathcal{P}_{a,b})\}$$

The number of firings of actor v_b that can interfere with $q + 1$ consecutive firings of actor v_a then equals to the number of elements in $P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}$ (with $\delta_{a,b}^\circ = \delta(\mathcal{P}_{a,b}) + \delta(\mathcal{P}_{b,a})$ a shorthand notation):

$$\zeta_{b \rightarrow a}(q) = |P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}| = \delta_{a,b}^\circ + q - 1$$

In the following we use this function to determine a tighter maximum response time than the one solely based on period-and-jitter.

10.3.3 COMBINING PERIOD-AND-JITTER & CYCLIC DATA DEPENDENCIES

In order to include the effect that cyclic data dependencies bound interference in the maximum response time equations, we replace the interference characterization $\eta_b(\Delta\tau)$ of a task t_b with a more accurate characterization $\gamma_{b \rightarrow a}(\Delta\tau, q)$. This characterization denotes the maximum number of enablings a task t_b can have during a time interval $\Delta\tau$ and $q + 1$ consecutive executions of a task t_a :

$$\gamma_{b \rightarrow a}(\Delta\tau, q) = \min(\eta_b(\Delta\tau), \zeta_{b \rightarrow a}(q)) \quad (10.10)$$

The first term of the minimum function denotes the interference of a task t_b on a task t_a , given that their enablings are independent of each other. It ensures that the maximum response time of a task t_a cannot become more pessimistic than by applying Equation 10.4. The function $\zeta_{b \rightarrow a}(q)$ that we have defined in the previous section describes the maximum number of enablings a task t_b can have during $q + 1$ consecutive iterations of a task t_a due to cyclic data dependencies between the two. Consequently, $\gamma_{b \rightarrow a}(\hat{w}'_a(q), q)$ is equal to $\eta_b(\hat{w}'_a(q))$ if there is no directed cycle containing both tasks t_a and t_b , as then either $\delta(\mathcal{P}_{a,b})$, $\delta(\mathcal{P}_{b,a})$ or both are infinite. However, if the tasks t_a and t_b are both part of a single directed cycle then $\zeta_{b \rightarrow a}(q)$ is finite, which can result in a tighter upper bound on the response time of task t_a . Note that a cycle does not necessarily have to be a simple cycle, as it is illustrated by the cycle in Figure 10.3 consisting of the paths $\{e_{a,d}, e_{d,b}\}$ and $\{e_{b,a}\}$.

Using this new upper bound on the maximum number of enablings of a task t_b to reduce the maximum busy period calculated with Equation 10.5 results in the following maximum response time equations:

$$\hat{w}_a(q) = (q + 1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \gamma_{b \rightarrow a}(\hat{w}'_a(q), q) \cdot \hat{\chi}_b \quad (10.11)$$

$$\hat{R}_a = \max_{q \geq 0} (\hat{w}_a(q) - q \cdot P_a) \quad (10.12)$$

Note that the maximum q that needs to be taken into account is not impacted by the interference reduction, i.e. it still suffices to consider q for which either $q = 0$ or $\hat{w}'_a(q - 1) \geq q \cdot P_a$ applies.

We require the maximum response times calculated with Equation 10.12 to be more accurate than the maximum response times calculated with Equation 10.6, i.e.:

Lemma 10.2 (Accuracy Improvement of Reduced Maximum Response Times). *It always holds that $\hat{R}_a \leq \hat{R}'_a$.*

Proof. It can be seen that $\hat{R}_a \leq \hat{R}'_a$ if the same holds for all maximum busy periods, i.e. it holds for all q that $\hat{w}_a(q) \leq \hat{w}'_a(q)$. From $\forall x, y: \min(x, y) \leq x$ it immediately follows that $\forall \Delta\tau, q: \gamma_{b \rightarrow a}(\Delta\tau, q) \leq \eta_b(\Delta\tau)$. Therefore it also holds that:

$$\hat{w}_a(q) = q \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \gamma_{b \rightarrow a}(\hat{w}'_a(q), q) \cdot \hat{\chi}_b \leq q \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \eta_b(\hat{w}'_a(q)) \cdot \hat{\chi}_b = \hat{w}'_a(q)$$

□

10.4 BUFFER SIZING

In this section we discuss buffer sizing in the context of our period-and-jitter based analysis. For that purpose we first discuss buffer sizing based on the analysis results obtained by the analysis flow depicted in Figure 10.1. We differ between

FIFO buffers with blocking and with non-blocking writes. Subsequently we introduce the basic idea of an iterative buffer sizing according to Figure 10.2 and finally discuss modifications that need to be considered if the buffer sizing is conducted in an iterative manner.

10.4.1 COMPUTATION OF SUFFICIENTLY LARGE BUFFER CAPACITIES

Recall the relation between a FIFO buffer and the corresponding two dataflow edges depicted in Figure 8.1. The capacity of the modeled buffer is $\delta_{a,b} + \delta_{b,a}$. We assume that the number of initially full locations $\delta_{a,b}$ is given, as this number is usually determined by the functional behavior of the analyzed application. Using the best-case and worst-case schedules determined with the algorithms in Equations 10.1 and 10.2, we can determine sufficiently large numbers of empty locations $\delta_{b,a}$ for buffers with blocking and non-blocking writes.

To prevent an overflow of a buffer with non-blocking writes that can lead to a functionally incorrect behavior of the application it must be ensured that there is at least one empty location in the buffer whenever task t_a is enabled. If the buffer initially holds $\delta_{b,a}$ empty locations, then task t_a can execute $\delta_{b,a}$ times before another empty location must be freed by a finished execution of task t_b . To prevent a buffer overflow it therefore must hold that an iteration i of task t_b must have finished before iteration $i + \delta_{b,a}$ of task t_a is enabled, i.e.:

$$\forall_{i \geq 0}: f_b(i) \leq e_a(i + \delta_{b,a})$$

Using the periodic bounds on enabling and finish times presented in Section 10.2, it can be seen that:

$$\forall_{i \geq 0}: \hat{f}_b(i) \leq \check{e}_a(i + \delta_{b,a}) \Rightarrow f_b(i) \leq e_a(i + \delta_{b,a})$$

With substitution of the periodic bounds and $P_a = P_b$, which always holds for tasks of the same task graph, it follows for buffers with non-blocking writes:

$$\forall_{i \geq 0}: \hat{s}_b + i \cdot P_b + \hat{p}_b \leq \check{s}_a + (i + \delta_{b,a}) \cdot P_b \Leftrightarrow \delta_{b,a} \geq \frac{\hat{s}_b + \hat{p}_b - \check{s}_a}{P_b} \quad (10.13)$$

Hence it can be concluded that a buffer overflow cannot occur if the number of initially free locations $\delta_{b,a}$ is large enough to satisfy the constraint in Equation 10.13.

In contrast to buffers with non-blocking writes it holds for buffers with blocking writes that a buffer overflow can never occur. Due to the blocking on writes, an early enabling of task t_a is delayed until task t_b finishes its execution and thereby frees a location in the buffer. Given best-case and worst-case schedules that represent bounds on the enabling times of tasks, the buffer capacities therefore do not have to be large enough to account for any difference between best-case and worst-case. Instead, it only has to be guaranteed that both bounds

remain valid, to ensure that the calculated maximum enabling jitters remain temporally conservative.

In the computation of the best-case schedule with Equation 10.1, unknown buffer capacities are assumed to be infinite. From the monotonicity of dataflow graphs [WBS07b] it follows that reducing a number of tokens cannot lead to earlier enablings of actors. Therefore any finite number of initially empty locations $\delta_{b,a}$ cannot lead to earlier start times in the best-case model, the best-case schedule remains a valid lower bound on the enabling times for any $\delta_{b,a}$.

With respect to the bound on the worst-case schedule, it has to be ensured that the number of initially empty locations $\delta_{b,a}$ is large enough such that the start times computed with the algorithm in Equation 10.2 are not affected. Considering the number of empty locations $\delta_{b,a}$ in Equation 10.2 would result in an additional constraint on the maximum start time of actor $\hat{v}_a$:

$$\hat{s}_a - \hat{s}_b \geq \hat{\rho}_b - \delta_{b,a} \cdot P_b$$

It can be seen that if this constraint is not violated for the start times computed by Equation 10.2, then the periodic worst-case schedule remains admissible and the upper bounds on the enabling times of tasks remain valid. Resolving the constraint to $\delta_{b,a}$ results in the following constraint on the number of sufficient empty locations for buffers with blocking writes:

$$\delta_{b,a} \geq \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a}{P_b} \quad (10.14)$$

From this follows that the bounds on enabling times remain valid if the number of initially free locations $\delta_{b,a}$ is large enough to satisfy the constraint in Equation 10.14.

This lets us conclude that we can use Equation 10.13 for non-blocking write buffers and Equation 10.14 for blocking write buffers in a post-analysis buffer sizing according to Section 10.1.1.

10.4.2 BASIC IDEA OF AN ITERATIVE BUFFER SIZING

This section illustrates the non-monotone relation between buffer capacities, interference and minimum throughput with an example. Using this example, we show that performing buffer sizing iteratively during temporal analysis instead of once afterwards can result in both smaller buffer capacities and higher throughput guarantees.

Consider the task graph depicted in Figure 10.4. The tasks t_a and t_b are executed on a shared processor using an SPP scheduler, with task t_b having a higher priority π_2 than task t_a . Executions of task t_b can preempt and thereby delay executions of task t_a due to the given priority assignment, which can make the maximum response time $\hat{R}_a$ of task t_a larger than its WCET of $5\mu s$. The tasks are triggered by a periodic source t_s with a period of $P_s = 10\mu s$ and the tasks

Figure 10.4: Left: Task graph containing buffers with blocking writes and tasks scheduled by an SPP scheduler. Right: Relation between buffer capacity δ and minimum throughput $\tilde{Y}$.

communicate via FIFO buffers with blocking writes of the capacities 3 and δ , respectively.

In a dataflow model of the task graph the FIFO buffers with blocking writes correspond to cyclic data dependencies. According to the Maximum Cycle Mean (MCM) equation [SB09a] it holds that both cyclic data dependencies limit the minimum throughput $\tilde{Y}$ as follows:

$$\tilde{Y} = \min \left(\frac{3}{10\mu s + \hat{R}_b}, \frac{\delta}{\hat{R}_b + 4\mu s} \right)$$

Classical response time analysis [TBW94, HWGB13] considers maximum response times to be independent of buffer capacities. Under this assumption, the minimum throughput $\tilde{Y}$ is monotonically increasing in the buffer capacity δ , until it is not limited by the rightmost buffer, but by the leftmost buffer with the fixed capacity of 3. This behavior can be explained by the fact that larger buffer capacities generally allow for more pipeline parallelism. For the given example the minimum throughput allowed by the cyclic data dependencies, which is indicated by the dotted curve in Figure 10.4, is not large enough for any δ to meet the throughput constraint imposed by the source, indicated by the horizontal, dashed line.

In contrast, if the response time calculation in Equation 10.12 is applied, then the interference of task t_b on task t_a is bounded by the cyclic data dependency between the tasks, leading to smaller maximum response times $\hat{R}_a$ for smaller δ . This is due to the fact that the buffer between the tasks t_a and t_b blocks on both reads and writes. The buffer thus effectively bounds jitter between the tasks, and thereby also interference of task t_b with task t_a .

For instance, if the capacity of the buffer between the tasks t_a and t_b were equal to $\delta = 1$, then both tasks would have to wait for the other to finish its execution before they can start theirs. Consequently, the executions of the tasks t_a and t_b would be mutually exclusive and the interference of task t_b with task t_a zero.

If the positive effect of smaller buffer capacities on response times exceeds the negative effect of smaller buffer capacities on pipeline parallelism, then an increased minimum throughput $\tilde{Y}$ can be observed for these buffer capacities.

The resulting, in general non-monotone relation between buffer capacities and minimum throughput is illustrated by the boxed curve in Figure 10.4. As it can be concluded from the graph, the required throughput guarantee indicated by the dashed line can be met for buffer capacities δ between 1 and 4.

Note that existing temporal analysis and buffer sizing approaches do not come to this conclusion. This is due to the fact that unknown buffer capacities are not considered during temporal analysis. Instead, buffer sizing is performed only once post-analysis, as depicted in Figure 10.1. Not considering buffer capacities during temporal analysis is equivalent to assuming unknown buffer capacities as unconstrained, i.e. $\delta = \infty$. As it can be seen from the graph in Figure 10.4, the minimum throughput $\tilde{\gamma}$ is constant for any $\delta \geq 5$, a buffer capacity of $\delta = \infty$ would therefore result in a throughput guarantee below the required throughput. Existing temporal analysis approaches would hence conclude that the required throughput is infeasible for the given example. This shortcoming motivates the introduction of our iterative, integrated temporal analysis and buffer sizing flow depicted in Figure 10.2.

10.4.3 BUFFER SIZING IN THE ITERATIVE ANALYSIS FLOW

To apply buffer sizing iteratively we can just use the same equations to determine sufficiently large buffer capacities as in the post-analysis buffer sizing. In addition, for blocking write buffers we need to conduct a clamping of initially empty locations by the initially empty locations of the previous iteration of the analysis flow. Otherwise, termination of the flow would not be guaranteed, as detailed in Appendix B.1. Using Equations 10.13 and 10.14, we can consequently estimate the numbers of initially empty locations in step 2 of the analysis flow as follows (with n being the flow iteration):

$$\delta_{b,a}^n = \begin{cases} \max\left(\left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a}{P_b} \right\rceil, 0\right) & \text{buffer with non-blocking writes} \\ \max\left(\left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a}{P_b} \right\rceil, \delta_{b,a}^{n-1}\right) & \text{buffer with blocking writes} \end{cases} \quad (10.15)$$

Let us now compare the buffer capacities obtained with Equation 10.15 for the cases that buffers with blocking or non-blocking writes are used, as well as their effects on the different analysis models and the resulting interference between tasks.

As it holds for all actors that $\check{s}_a \leq \hat{s}_a$ it follows that capacities of buffers with blocking writes are always smaller or equal to capacities of buffers with non-blocking writes for the same bounds on schedules. This is due to the fact that buffers with non-blocking writes cannot compensate for differences between best-case and worst-case schedules, while buffers with blocking writes can delay tasks such that the differences between enablings become smaller. Buffers with blocking writes effectively reduce the maximum enabling jitter between tasks. Hence using buffers with blocking writes instead of buffers with non-blocking

writes does not only result in smaller buffer capacities, but also less interference and thereby a higher minimum throughput, provided that the additional synchronization overhead is neglectable.

Remember that the analysis results determined with an iterative buffer sizing of blocking write buffers are only valid for the underlying application model if the same buffer capacities are considered within the application model as well. Likewise it is not sufficient to update buffer capacities in the precedence constraint analysis model in each flow iteration, but we also need to update capacities in the response time analysis model to maintain inclusion abstraction between response time analysis and precedence constraint analysis models in each flow iteration. The best-case and worst-case analysis models both consider unknown buffer capacities as infinite or as constant upper bounds, given that the latter are specified as inputs to the analysis flow. Therefore both best-case and worst-case analysis models remain the same throughout all flow iterations, except for the dependence of the worst-case analysis model on maximum response time estimates.

Note that on convergence of the analysis flow, we need to replace the infinite or fixed buffer capacities in the worst-case analysis model by the convergent estimates, as otherwise bounding between the token components in the worst-case model and their counterparts in the response time analysis model would not hold. This is allowed as the convergent buffer capacities are derived from the start times of actors in the worst-case model, which implies that the temporal behavior of the worst-case model is not affected by the replacement.

Buffers with non-blocking writes do not change the temporal behavior of tasks, the jitter reduction observed for buffers with blocking writes consequently also cannot occur. However, the iteratively calculated estimates on buffer capacities still represent upper bounds on the enabling jitters between tasks, which can be more accurate than the absolute maximum enabling jitters obtained with Equation 10.3. A consideration of these estimates in the maximum response time calculations can therefore result in tighter bounds on interference, which can lead to the determination of both smaller buffer capacities and a higher minimum throughput.

Finally note that because the iteratively determined non-blocking write buffer capacities are merely a modeling construct it is sufficient to consider such capacities in the precedence constraint analysis model, while assuming them as infinite in all other analysis models.

10.5 USING SYNCHRONIZATION EDGES FOR THE OPTIMIZATION OF SCHEDULES

Given our iterative buffer sizing technique, we can also take the concept of bounding interference using cyclic data dependencies one step further. For that matter we add non-functional data dependencies between tasks that belong to the same task graph and share the same processor. More precisely, as exemplified in Figure 10.5, we add two edges between any two tasks in the same task graph and on

the same processor, one in the forward and one in the backward direction. The initial numbers of tokens on such edges we leave variable and use our iterative buffer sizing to determine suitable numbers.

Figure 10.5: Inserting synchronization edges $e_{a,b}$ and $e_{b,a}$ between tasks t_a and t_b that share the same processor.

Such dependencies, which we call synchronization edges, have the same temporal behavior as buffer components, but are non-functional as no data is exchanged between the tasks via the respective edges. As for the regular buffers, we can treat these pairs of edges either as buffers with non-blocking writes or as buffers with blocking writes, with the difference that for actual buffers we assume the numbers of full locations as given, whereas for synchronization edges we can apply Equation 10.15 for both forward and backward edges.

If we consider the synchronization edges as buffers with non-blocking writes, then the dependencies are merely a modeling construct that can be used as an alternative way to characterize interference between tasks. In this case the behavior of the application itself is not changed at all, such that the respective dependencies also do not need to be added to the implementation. Nevertheless, as we show in the case study, the usage of non-blocking synchronization edges can result in an increased analysis accuracy.

If we treat synchronization edges as buffers with blocking writes, then the applied iterative buffer sizing does not only increase analysis accuracy, but actively influences the schedule of tasks. For instance, if the iterative buffer sizing would determine total capacities of one for all pairs of synchronization edges, which would correspond to FIFO buffers with capacities of one, then the scheduling of these tasks would not be SPP anymore, but in fact be entirely static. In consequence, the usage of blocking write synchronization edges enables a schedule optimization which determines a schedule that ranges between static scheduling on the one hand and unrestricted SPP scheduling on the other.

Note that if blocking write synchronization edges are used then the determined analysis results are only valid if the respective edges are also added to the implementation itself. Fortunately, inserting synchronization edges to an implementation has a very low cost. For one, the edges only affect tasks that are executed on the same processor, such that the implementation does not involve any inter-processor communication or synchronization. And second, no data is exchanged using synchronization edges, such that also space requirements are usually neglectable.

Lastly, also note that since the synchronization edges do not communicate any data, we can further allow for the respective numbers of tokens to become negative. For example, remember that an edge $e_{a,b}$ with $\delta_{a,b} = 1$ initial token implies that an iteration i of a task t_b cannot start before iteration $i + 1$ of task t_a has finished. Likewise, if we consider an edge with $\delta_{a,b} = -1$ then we can use this to express that an iteration i of task t_b cannot start before iteration $i - 1$ of task t_a has finished. We only need to ensure two things: First, while single numbers of tokens can become negative, we need to ensure that the numbers of tokens on each cycle never become negative as well, as otherwise the Floyd-Warshall algorithm used to determine numbers of tokens on paths would not converge. And second, we still need to ensure that the numbers of tokens are monotonically increasing in increasing iterations of the analysis flow, such that termination as discussed in the previous section can be guaranteed. We can easily conform to both these constraints by a clamping similar to the one discussed in Section 10.4.

10.6 PROOF OF CONSERVATIVENESS

Now that we have described all steps of our analysis flows we can finally prove that the results obtained from our analysis models are not only valid within the analysis models themselves, but are also valid for the underlying application model.

First we show that the maximum response times calculated with Equation 10.12 are conservative upper bounds on the differences between maximum external enabling and finish times in the response time analysis model, such that using these maximum response times as firing durations in the worst-case model makes the latter model a bounding abstraction of the response time analysis model. This relation is captured by the following lemma:

Lemma 10.3 (Worst-Case Conservativeness). *Let the firing durations $\hat{\rho}_a$ of actor components $\hat{v}_a$ in the worst-case analysis model be set to the respective $\hat{R}_a$ and let $\hat{\varepsilon}_a^{ext}(i)$ be the strictly periodic upper bounds on the respective external enabling times used in the task components t_a in the response time analysis model. Then it holds that $G^{WC} \triangleright G^{RT}$ and $in_{G^{WC}} \supseteq in_{G^{RT}}$, i.e. the worst-case analysis model is a valid worst-case bounding abstraction of the response time analysis model.*

Proof. According to the firing duration criterion in Equation 9.6, the worst-case bounding abstraction is valid if it holds for all firing durations of actors in the worst-case analysis model that $\varepsilon_a(i) + \chi_a(i) + \mathcal{I}_a[\varepsilon_a(i), f_a(i)] \leq \hat{\varepsilon}_a^{ext}(i) + \hat{\rho}_a$. To prove this inequality let us first bring its left side to a different form. We can denote with $w_a(i)$ an actual busy period that precedes and includes an iteration i of a task t_a in the response time analysis model. As an actual busy period we define an iteration i and all its q_i preceding iterations for which it holds that the first iteration is externally enabled while all other iterations are self-delaying each other, i.e. it holds that $\varepsilon_a(i - q_i) = \varepsilon_a^{ext}(i - q_i)$ and $\forall_{i - q_i < i' \leq i: \varepsilon_a(i') > \varepsilon_a^{ext}(i')}$.

Using the definition of self-delay and Equation 9.2 we can formulate an actual busy period $w_a(i)$ as follows:

$$w_a(i) = \sum_{i'=i-q_i}^i \chi_a(i') + \mathcal{I}_a[\varepsilon_a^{ext}(i - q_i), \varepsilon_a^{ext}(i - q_i) + w_a(i)]$$

By definition, it holds that $\varepsilon_a^{ext}(i - q_i) + w_a(i)$ is equal to the left side of the firing duration criterion. With that it follows:

$$\begin{aligned} \varepsilon_a(i) + \chi_a(i) + \mathcal{I}_a[\varepsilon_a(i), f_a(i)] &= \varepsilon_a(i - q_i) + w_a(i) \quad (10.16) \\ &= \varepsilon_a^{ext}(i - q_i) + \sum_{i'=i-q_i}^i \chi_a(i') + \mathcal{I}_a[\varepsilon_a^{ext}(i - q_i), \varepsilon_a^{ext}(i - q_i) + w_a(i)] \\ &\leq \hat{\varepsilon}_a^{ext}(i - q_i) + (q_i + 1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \gamma_{b \rightarrow a}(w_a(i), q_i) \cdot \hat{\chi}_b \\ &\leq \hat{\varepsilon}_a^{ext}(i - q_i) + (q_i + 1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \gamma_{b \rightarrow a}(\hat{w}'_a(q_i), q_i) \cdot \hat{\chi}_b \\ &= \hat{s}_a + (i - q_i) \cdot P_a + \hat{w}_a(q_i) \leq \hat{\varepsilon}_a^{ext}(i) + \hat{R}_a = \hat{\varepsilon}_a^{ext}(i) + \hat{\rho}_a \end{aligned}$$

Equality between the first and the second line holds by definition of the busy period, which implies $\varepsilon_a(i - q_i) = \varepsilon_a^{ext}(i - q_i)$. Between the second and the third line we make use of the fact that in the restricted task components of the response time analysis model only external enabling times $\varepsilon_a^{ext}(i)$ smaller or equal to $\hat{\varepsilon}_a^{ext}(i)$ can occur, that $\forall_i: \chi_a(i) \leq \hat{\chi}_a$ and that the considered bound on interference is conservative to the actual interference that can occur in the response time analysis model, as discussed in the previous two sections.

Between the third and the fourth line we have to show that $\forall_i: w_a(i) \leq \hat{w}'_a(q_i)$. For that matter assume that $w_a(i) = g(w_a(i))$ and $\hat{w}'_a(q_i) = h(\hat{w}'_a(q_i))$ are the fixed points of two functions $g(x)$ and $h(x)$ that we can define as follows:

$$\begin{aligned} g(x) &= \sum_{i'=i-q_i}^i \chi_a(i') + \mathcal{I}_a[\varepsilon_a^{ext}(i - q_i), \varepsilon_a^{ext}(i - q_i) + x] \\ h(x) &= (q_i + 1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \eta_b(x) \cdot \hat{\chi}_b \end{aligned}$$

By definition it holds that $\forall_i: \chi_a(i) \leq \hat{\chi}_a$, as well as that the interference characterization using period-and-jitter is a conservative upper bound on higher priority interference, i.e.:

$$\forall_x: \mathcal{I}_a[\varepsilon_a^{ext}(i - q_i), \varepsilon_a^{ext}(i - q_i) + x] \leq \sum_{t_b \in hp_a} \eta_b(x) \cdot \hat{\chi}_b$$

From this follows that $\forall_x: g(x) \leq h(x)$. However, as exemplified by the left graph in Figure 10.6, this does not necessarily imply that the same relation also holds for the fixed points of these functions.

Figure 10.6: Fixed points of two monotonically increasing functions $g(x)$ and $h(x)$ with $\forall_x: g(x) \leq h(x)$.

In contrast, the right graph in Figure 10.6 depicts a situation in which the relation of the functions is preserved in the relation of their fixed points. It can be seen that this is generally the case if both $g(x)$ and $h(x)$ are monotonically increasing and if $g(0)$ is not negative. Both $g(x)$ and $h(x)$ as defined above are monotonically increasing in x . Moreover, $g(0) \geq 0$ follows immediately from the fact that we do not consider negative execution times, i.e. it holds $\forall_i: \chi_a(i) \geq 0$, as well as that the interference function $\mathcal{I}_a[\varepsilon_a^{ext}(i - q_i), \varepsilon_a^{ext}(i - q_i) + x]$ cannot become negative. This allows us to conclude that indeed $\forall_i: w_a(i) \leq \hat{w}'_a(q_i)$.

In the fifth line of Equation 10.16 we substitute the upper bound on the external enabling time $\hat{\varepsilon}_a^{ext}(i - q_i)$ obtained from the worst-case analysis model with $\hat{s}_a + (i - q_i) \cdot P_a$ and the right-hand expression from the fourth line with $\hat{w}_a(q_i)$. Substituting $\hat{s}_a + i \cdot P_a$ back to $\hat{\varepsilon}_a^{ext}(i)$ and using the definition of the maximum response time from which follows that $\forall_{q_i \geq 0}: \hat{w}_a(q_i) - q_i \cdot P_a \leq \hat{R}_a$ finally lets us conclude the proof that the firing duration criterion holds for all restricted tasks and actors and therewith also $G^{WC} \supseteq G^{RT}$ and $in_{G^{WC}} \supseteq in_{G^{RT}}$ if the maximum response times $\hat{R}_a$ determined in this section are used as firing durations $\hat{\rho}_a$ in the worst-case analysis model. $\square$

It remains to be shown that on convergence of our analysis flows also the response time analysis model is a valid inclusion abstraction of the underlying application model, such that all bounds computed with our flows are also valid bounds on the application. We prove this fundamental property by the following theorem:

Theorem 10.1 (Analysis Conservativeness). *On convergence of our analysis flows it holds that $G^{RT} \supseteq G^\Lambda$ and $in_{G^{RT}} \supseteq in_{G^\Lambda}$, i.e. the response time analysis model is a valid inclusion abstraction of the application model. This implies that on convergence also the worst-case, best-case and precedence constraint analysis models are valid worst-case, best-case and inclusion abstractions of the application model, respectively.*

Proof. In Section 9.3.1 we have determined that a response time analysis model is a valid inclusion abstraction of the underlying application model if the bounds on schedules used in the former model are wide enough such that all streams

that can occur in the latter model are allowed. This property is subsumed in the schedule criterion in Equation 9.3, whose satisfaction we consequently prove here.

Put into words, the schedule criterion applies if the lower and upper bounds on (external) enabling and finish times computed with our analysis flows are conservative lower and upper bounds on the actual (external) enabling and finish times that can occur in the application model.

It can be easily seen that the periodic lower bounds on (external) enabling and finish times computed with the best-case analysis model are conservative lower bounds on the respective (external) enabling and finish times that can occur in the application model. Hence we only need to prove the other half of the schedule criterion, i.e. that on convergence of our analysis flows the periodic upper bounds on (external) enabling and finish times are conservative upper bounds on the respective times in the application model.

For that matter consider the following perspective on application models as defined in Section 9.2. First, it can be seen that all streams transferred between components in the application model are defined either by the finish times of sources or tasks (this holds for all data dependencies, as well as the finish time streams of resource dependencies) or by the enabling times of tasks (this applies for the enabling time streams of resource dependencies). Second, no matter whether we consider data or resource dependencies, we can classify any dependencies in an application model as forward dependencies (that is, dependencies leading away from the source tasks) or backward dependencies (dependencies leading back to the source tasks). It can be seen that application models without neither cyclic data nor cyclic resource dependencies do not have backward dependencies. And third, consider that any application model expressed in our timed component model can be expressed as the feedback composition of a single component G_{FW}^Λ as depicted on the left side of Figure 10.7, with G_{FW}^Λ being obtained by repetitive application of parallel and serial composition. We assume that the left-hand input trace X^s is defined by the respective streams of all source components, that the right-hand output trace Y^f consists of the enabling and finish times on all forward dependencies and that the trace $Y^b = X^b$ on the feedback loop contains all enabling and finish times on backward dependencies.

Figure 10.7: Left: Application model as feedback composition of component G_{FW}^Λ that is obtained from repetitive application of parallel and serial composition. Right: Iterative analysis flow captured in the timed component model.

Now let us come to our iterative analysis flows. It can be seen that for acyclic applications, i.e. for applications without backward dependencies, our analysis

flows would not need to be iterative, but could compute bounds on schedules and response times in one iteration (given a clever rearrangement of the steps within the iterative loop such that bounds on (external) enabling and finish times, as well as maximum response times of tasks are computed in an interleaved way, following the dependencies away from the source tasks). This implies that for cyclic applications the only parameters that need to be transferred between different iterations of our flows are the enabling and finish times of tasks that have outgoing backward dependencies.

With that in mind, we can rearrange the steps within the iterative loops of our analysis flows as described above, such that they always return upper bounds $\hat{Y}^f \cup \hat{Y}^b$ on the enabling and finish times of all tasks, given the source periods X^s and upper bounds on enabling and finish times of tasks with outgoing backward dependencies $\hat{X}^b$. Let us capture such rearranged steps in a component G_{FW}^{Flow} in our timed component model. Given that we further connect the component G_{FW}^{Flow} with a feedback loop for the enabling and finish times of backward dependencies, as depicted on the right side of Figure 10.7, it holds by construction that this feedback composition produces the same analysis results as the corresponding original analysis flow, in the form of streams of enabling and finish times.

Therewith we have constructed a component that has the same input and output ports as component G_{FW}^Λ on the left side of the figure. Moreover, it follows from Lemma 10.3 that our analysis flows produce upper bounds on maximum response times given lower and upper bounds on schedules, whereas our worst-case analysis produces upper bounds on schedules given firing durations set to maximum response times [HWGB13]. This implies that given bounding inputs our component G_{FW}^{Flow} produces bounding outputs, compared to G_{FW}^Λ , i.e. it holds that $G_{FW}^{Flow} \triangleright G_{FW}^\Lambda$.

Finally, it follows immediately from Lemma 5.8 that worst-case bounding is preserved on feedback composition, such that given the same inputs (given the same source input streams) and given that the feedback composition has a fixed point for these inputs (which corresponds to convergence of the original analysis flow, whereas a lack of fixed points is the equivalent of a constraint violation) the right graph in Figure 10.7 produces enabling and finish times that are conservative upper bounds on the enabling and finish times produced by the left graph. And since external enabling times of tasks are defined by the finish times of other tasks connected via data dependencies, it follows that conservativeness must also apply for the respective upper bounds on external enabling times.

This lets us conclude that besides the lower bounds also the convergent upper bounds on (external) enabling and finish times determined with our analysis flows are temporally conservative bounds on the respective (external) enabling and finish times in the application model, i.e. the schedule criterion applies and therewith $G^{RT} \supseteq G^\Lambda$ and $in_{G^{RT}} \supseteq in_{G^\Lambda}$.

□

In this section we evaluate the benefits of our period-and-jitter analysis approaches on two examples. The first example we discuss is the simplified task graph of the packet decoding mode of a WLAN 802.11p transceiver application. We show that the additional consideration of cyclic data dependencies results in an improved analysis accuracy compared to the approach from [HWGB13] that is solely based on a period-and-jitter interference characterization. Moreover, we demonstrate that results can be further improved compared to a post-analysis buffer sizing if an iterative buffer sizing and synchronization edges are used, and that for both buffers with blocking and non-blocking writes. The second example we consider is synthetic, although it could well be part of a realistic application. With that example we discuss the effects of an iterative buffer sizing in more detail. Note that no related works discussed in Section 8.2 except for [HWGB13] can analyze the presented applications due to their lack of support for combinations of cyclic data and cyclic resource dependencies, which are prevalent in both discussed applications.

10.7.1 WLAN 802.11P TRANSCEIVER

Figure 10.8: Task graph of the packet decoder of a WLAN 802.11p transceiver.

We analyze the temporal behavior of a WLAN 802.11p transceiver [AHG07] application. WLAN 802.11p transceivers are used in safety-critical automotive applications like automated braking systems, which imposes the requirement to give guarantees on their temporal behavior. The considered transceiver has several modes and is executed on a multiprocessor system for performance reasons. Here we only analyze the part of the application that is active in the packet decoding mode.

Figure 10.8 depicts the task graph of the packet decoding mode, with the peculiarity that we use dataflow edges with tokens instead of buffers to depict data dependencies between tasks for clarification, according to Figure 8.1. The task graph is triggered by a single periodic source with a period P_{SRC} . All symbols transmitted by the source are first processed by a hardware filter with a variable execution time, followed by a Fast Fourier Transformation (FFT) whose execution time is constant. Thereafter, the input data is processed by a feedback loop, as

the settings of the channel equalizer t_{EQ} for the reception of symbol i are based on an estimate of the channel t_{CHEST} during the reception of symbol $i - 2$. This estimate of the channel is based on the received symbol $i - 2$ and the reencoded symbol $i - 2$, which is obtained by reencoding the error corrected bits of symbol $i - 2$ produced by the viterbi channel decoder t_{VIT} .

As you may have noticed, the depicted data dependencies between tasks are only the forward dependencies, modeling the production and consumption of data, whereas the backward edges modeling the production and consumption of space are missing. Consequently we assume that all buffers have infinite capacities initially and it is subject to our analysis flows to determine sufficiently large numbers of tokens for the backwards edges, i.e. to determine sufficiently large buffer capacities. We determine suitable buffer capacities both with a post-analysis buffer sizing according to the analysis flow in Figure 10.1 and with an iterative buffer sizing according to the analysis flow in Figure 10.2.

We assume all software tasks (all tasks except source and hardware filter) being mapped to three different processors, which is indicated by the different colors of these tasks. If multiple tasks are mapped to a shared processor, then they are scheduled by an SPP scheduler, with their priorities denoted as π_x next to the tasks and with larger x denoting higher priorities. For instance, the tasks t_{FFT} and t_{EQ} share a processor, with task t_{FFT} having a higher priority than task t_{EQ} .

The BCETs and WCETs of the tasks, which are denoted next to the corresponding vertices in the graph, represent theoretical bounds on the actual execution times on the Starburst platform [DWBS13]. The bounds adhere to the requirements given in Chapter 9, i.e. they are temporally conservative, independent of schedules and include communication times. On the Starburst platform these requirements can be satisfied if the considered processors are uncached, if SDRAM is not used and if communication scheduling is done in a round-robin fashion.

In the following, we first apply the original analysis flow from [HWGB13] that calculates maximum response times without consideration of cyclic data dependencies and then compare its results to the results obtained by an application of our accuracy-improved analysis flow from Figure 10.1. Hereby we assume a fixed source period of $P_{SRC} = 12\mu s$, which corresponds to a source frequency of roughly $83.3kHz$, and put special focus on detailing the computations in the different steps of our analysis flow. Afterwards, we additionally consider an iterative buffer sizing, both for buffers with blocking and non-blocking writes. For the sake of a quantified performance comparison between our different analysis approaches we do not keep the source period fixed, but vary it between $17\mu s$ and $10\mu s$.

Furthermore, we consider an optional Direct Memory Access (DMA) unit between the A/D converter at the source and the filter task that can introduce bursts of up to three simultaneously arriving symbols. This behavior can be modeled by an additional source jitter of $J_{SRC} = 2 \cdot P_{SRC}$.

Before applying the analysis flows with or without a consideration of cyclic dependencies let us resolve the inequalities from the LPs in Equations 10.1 and 10.2

such that we get compact formulations of stop criteria, maximum start times and maximum jitters.

At first, we iteratively substitute the inequalities from the worst-case LP in Equation 10.2 for each cycle in Figure 10.8 to derive the temporal constraints that must be met for an admissible upper bound on the worst-case schedule. Due to the unconstrained buffer capacities in the presented task graph we only need to consider one cyclic data dependency, which is imposed by the feedback loop to the right of the task graph. The tasks t_{FILTER} , t_{FFT} , t_{CHEST} and t_{REENC} either have the highest priorities on their shared processors or are executed on separate processors. Therefore they cannot experience any interference from other tasks and the maximum firing durations of the corresponding actors in the worst-case analysis model are equal to their WCETs. By iterative substitution of dependencies we can derive the following temporal constraint imposed by the feedback loop:

$$\begin{aligned} \hat{s}_{\text{CHEST}} &\geq \hat{s}_{\text{REENC}} + \hat{\rho}_{\text{REENC}} \geq \dots \geq \hat{s}_{\text{CHEST}} + \hat{\rho}_{\text{CHEST}} + \hat{\rho}_{\text{EQ}} + \hat{\rho}_{\text{DEMAP}} + \\ &\quad \hat{\rho}_{\text{DEINT}} + \hat{\rho}_{\text{VIT}} + \hat{\rho}_{\text{REENC}} - 2 \cdot P_{\text{SRC}} \quad (10.17) \\ \Leftrightarrow \hat{\rho}_{\text{EQ}} + \hat{\rho}_{\text{DEMAP}} + \hat{\rho}_{\text{DEINT}} + \hat{\rho}_{\text{VIT}} &\leq 18\mu\text{s} \end{aligned}$$

Using the maximum jitter equation from Section 10.2 we can further derive dependencies between maximum jitters and maximum response times. For instance, the maximum jitter of task t_{EQ} is defined as $\hat{J}_{\text{EQ}} = \hat{s}_{\text{EQ}} + \max(0, \hat{\rho}_{\text{EQ}} - P_{\text{SRC}}) - \check{s}_{\text{EQ}}$. With the best-case LP in Equation 10.1 it follows for the minimum start time of the corresponding actor $\check{v}_{\text{EQ}}$:

$$\check{s}_{\text{EQ}} \geq \check{s}_{\text{FFT}} + \check{\rho}_{\text{FFT}} \geq \check{\rho}_{\text{FILTER}} + \check{\rho}_{\text{FFT}} = 5\mu\text{s} \quad (10.18)$$

Note that the edge $e_{\text{CHEST,EQ}}$ is not considered in the best-case LP in Equation 10.1 as it contains initial tokens. From the worst-case LP in Equation 10.2 we obtain the following constraints on the start time of actor $\check{v}_{\text{EQ}}$:

$$\hat{s}_{\text{EQ}} \geq \hat{s}_{\text{FFT}} + \hat{\rho}_{\text{FFT}} \geq \hat{\rho}_{\text{FILTER}} + \hat{\rho}_{\text{FFT}} = 7\mu\text{s} \quad (10.19)$$

$$\hat{s}_{\text{EQ}} \geq \hat{s}_{\text{CHEST}} + \hat{\rho}_{\text{CHEST}} - 2 \cdot P_{\text{SRC}} \geq \hat{s}_{\text{FFT}} + \hat{\rho}_{\text{FFT}} + \hat{\rho}_{\text{CHEST}} - 2 \cdot P_{\text{SRC}} \geq -15\mu\text{s} \quad (10.20)$$

$$\hat{s}_{\text{EQ}} \geq \hat{s}_{\text{CHEST}} + \hat{\rho}_{\text{CHEST}} - 2 \cdot P_{\text{SRC}} \geq \dots \quad (10.21)$$

$$\begin{aligned} &\geq \hat{s}_{\text{EQ}} + \hat{\rho}_{\text{EQ}} + \hat{\rho}_{\text{DEMAP}} + \hat{\rho}_{\text{DEINT}} + \hat{\rho}_{\text{VIT}} + \hat{\rho}_{\text{REENC}} + \hat{\rho}_{\text{CHEST}} - 2 \cdot P_{\text{SRC}} \\ &= \hat{s}_{\text{EQ}} + \hat{\rho}_{\text{EQ}} + \hat{\rho}_{\text{DEMAP}} + \hat{\rho}_{\text{DEINT}} + \hat{\rho}_{\text{VIT}} - 18\mu\text{s} \end{aligned}$$

Equation 10.19 imposes a tighter constraint on $\hat{s}_{\text{EQ}}$ than Equation 10.20 and Equation 10.21 equals to the temporal constraint in Equation 10.17, which must be true for an admissible schedule. Therefore $\check{s}_{\text{EQ}}$ is only constrained by Equation 10.18 and $\hat{s}_{\text{EQ}}$ only by Equation 10.19. The LPs in Equation 10.1 and Equation 10.2 both minimize start times and as the Equations 10.18 and 10.19 are the only constraints that have to be considered, we can replace the inequalities in these constraints by equalities. Altogether, this results in the following maximum jitter of task t_{EQ} :

$$\hat{J}_{\text{EQ}} = \hat{s}_{\text{EQ}} + \max(0, \hat{\rho}_{\text{EQ}} - P_{\text{SRC}}) - \check{s}_{\text{EQ}} = \max(0, \hat{\rho}_{\text{EQ}} - 12\mu\text{s}) + 2\mu\text{s}$$

Analogously, we derive for the other maximum jitters:

$$\begin{aligned}\hat{J}_{\text{FILTER}} &= 0\mu\text{s}, \hat{J}_{\text{FFT}} = 2\mu\text{s}, \hat{J}_{\text{DEMAP}} = \hat{\rho}_{\text{EQ}} + \max(0, \hat{\rho}_{\text{DEMAP}} - 12\mu\text{s}), \\ \hat{J}_{\text{DEINT}} &= \hat{\rho}_{\text{EQ}} + \hat{\rho}_{\text{DEMAP}} + \max(0, \hat{\rho}_{\text{DEINT}} - 12\mu\text{s}) - 1\mu\text{s}, \\ \hat{J}_{\text{VIT}} &= \hat{\rho}_{\text{EQ}} + \hat{\rho}_{\text{DEMAP}} + \hat{\rho}_{\text{DEINT}} + \max(0, \hat{\rho}_{\text{VIT}} - 12\mu\text{s}) - 2\mu\text{s}, \\ \hat{J}_{\text{REENC}} &= \hat{J}_{\text{CHEST}} = \hat{\rho}_{\text{EQ}} + \hat{\rho}_{\text{DEMAP}} + \hat{\rho}_{\text{DEINT}} + \hat{\rho}_{\text{VIT}} - 3\mu\text{s}\end{aligned}$$

For the first iteration of the analysis flow we initialize all maximum firing durations to the respective WCETs, which results in initial maximum jitters of $2\mu\text{s}$ for all tasks except task t_{FILTER} , whose maximum jitter remains $0\mu\text{s}$, as presented in the first column of Table 10.1.

t_a	Period-and-Jitter only				Period-and-Jitter & Cyclic Data Dependencies			
	1 st iteration		2 nd iteration		1 st iteration		2 nd iteration	
	$\hat{J}_a$	$\hat{R}'_a$	$\hat{J}_a$	$\hat{R}'_a$	$\hat{J}_a$	$\hat{R}_a$	$\hat{J}_a$	$\hat{R}_a$
t_{FILTER}	0	3	0	3	0	3	0	3
t_{FFT}	2	4	2	4	2	4	2	4
t_{EQ}	2	6	2	6	2	6	2	6
t_{DEMAP}	2	5	6	9	2	5	6	5
t_{DEINT}	2	4	10	7	2	4	10	4
t_{VIT}	2	3	13	5	2	3	13	3
t_{REENC}	2	4	15	4	2	4	15	4
t_{CHEST}	2	2	15	2	2	2	15	2

Table 10.1: Temporal analysis results for Figure 10.8 (all values in μs).

With these temporal constraints and maximum jitters we can now also compute maximum response times, first using the maximum response time equations which do not consider that cyclic data dependencies limit interference. For that matter we use Equation 10.6, i.e. the response time computation using interference characterization $\eta_b(\Delta\tau)$ instead of Equation 10.12 with $\gamma_{b \rightarrow a}(\Delta\tau, q)$. For instance, we can compute the initial maximum response time of task t_{DEINT} as follows, using iterative fixed point computation:

$$\begin{aligned}\hat{w}'_{\text{DEINT}}(0) &= 1 \cdot \hat{\chi}_{\text{DEINT}} + \left\lceil \frac{\hat{J}_{\text{VIT}} + \hat{w}'_{\text{DEINT}}(0)}{P_{\text{SRC}}} \right\rceil \cdot \hat{\chi}_{\text{VIT}} + \left\lceil \frac{\hat{J}_{\text{CHEST}} + \hat{w}'_{\text{DEINT}}(0)}{P_{\text{SRC}}} \right\rceil \cdot \hat{\chi}_{\text{CHEST}} \\ &= 1 \cdot 1\mu\text{s} + \left\lceil \frac{2+3}{12} \right\rceil \cdot 1\mu\text{s} + \left\lceil \frac{2+3}{12} \right\rceil \cdot 2\mu\text{s} = 4\mu\text{s}\end{aligned}$$

As we only need to consider $\hat{w}'_{\text{DEINT}}(1)$ if $\hat{w}'_{\text{DEINT}}(0) > 1 \cdot P_{\text{SRC}}$, which is not the case here, we obtain:

$$\hat{R}'_{\text{DEINT}} = \max_{q \geq 0} (\hat{w}'_{\text{DEINT}}(q) - q \cdot P_{\text{SRC}}) = 4\mu\text{s}$$

Similarly we calculate the maximum response times for all other tasks, which can be found in the second column of Table 10.1. It can be easily verified that no maximum response time of the first iteration violates the temporal constraint imposed by Equation 10.17. Consequently, we also compute the maximum jitters of the second iteration that are depicted in the third column, based on the maximum response times of the first, and thereafter the maximum response times of the second iteration that are shown in the fourth column. However, the maximum response times in the second iteration lead to a violation of the temporal constraint in Equation 10.17:

$$\hat{\rho}_{\text{EQ}} + \hat{\rho}_{\text{DEMAP}} + \hat{\rho}_{\text{DEINT}} + \hat{\rho}_{\text{VIT}} = 6\mu\text{s} + 9\mu\text{s} + 7\mu\text{s} + 5\mu\text{s} = 27\mu\text{s} > 18\mu\text{s}$$

Therefore we conclude infeasibility with the analysis flow that does not consider the interference-limiting effect of cyclic data dependencies.

Now we apply our accuracy-improved analysis flow which takes into account that cyclic data dependencies limit interference. Using the maximum response time formulas from Equations 10.11 and 10.12 results in the maximum jitters and maximum response times presented in the fifth and sixth column of Table 10.1. These are just the same as in the first iteration of the original analysis flow, as are the maximum jitters computed in the second iteration with the accuracy-improved flow. However, the maximum response times in the second iteration are different, as we can e.g. see on the maximum response time calculation for task t_{VIT} :

$$\begin{aligned}\hat{w}'_{\text{VIT}}(0) &= 1 \cdot \hat{\chi}_{\text{VIT}} + \left\lceil \frac{\hat{J}_{\text{CHEST}} + \hat{w}'_{\text{VIT}}(0)}{P_{\text{SRC}}} \right\rceil \cdot \hat{\chi}_{\text{CHEST}} = 1\mu\text{s} + \left\lceil \frac{15 + 5}{12} \right\rceil \cdot 2\mu\text{s} = 5\mu\text{s} \\ \hat{w}_{\text{VIT}}(0) &= 1 \cdot \hat{\chi}_{\text{VIT}} + \max\left(\left\lceil \frac{\hat{J}_{\text{CHEST}} + \hat{w}'_{\text{VIT}}(1)}{P_{\text{SRC}}} \right\rceil, \delta^{\circ}_{\text{VIT,CHEST}} - 1\right) \cdot \hat{\chi}_{\text{CHEST}} \\ &= 1\mu\text{s} + \max\left(\left\lceil \frac{15 + 5}{12} \right\rceil, 1\right) \cdot 2\mu\text{s} = 3\mu\text{s}\end{aligned}$$

As $\hat{w}'_{\text{VIT}}(0) \leq 1 \cdot P_{\text{SRC}}$ it follows that $\hat{R}_{\text{VIT}} = 3\mu\text{s}$. The number of tokens on the cycle between the tasks t_{VIT} and t_{CHEST} effectively limits interference, which results in the same maximum response time of task t_{VIT} as in the first iteration. The same happens to all other maximum response times in our example as well, resulting in the maximum response times of the second iteration being equal to those of the first. Hence we can conclude a convergence of maximum response times without a violation of temporal constraints after the second iteration of our accuracy-improved analysis flow.

Based on these results we can now also determine sufficiently large FIFO buffer capacities in a post-analysis buffer sizing, according to Figure 10.1. Let us assume buffers with blocking writes for all data dependencies. Then we can determine

suitable capacities by first inserting backward edges for each forward edge in the task graph and thereafter calculating sufficient numbers of initial tokens for these edges, using Equation 10.14. For this calculation we require the maximum start times of all actors in the worst-case analysis model. For instance, the sufficient number of initial tokens on the inserted edge from actor v_{CHEST} to actor v_{FFT} can be calculated as the smallest integer that satisfies:

$$\delta_{\text{CHEST,FFT}} \geq \frac{\hat{s}_{\text{CHEST}} + \hat{\rho}_{\text{CHEST}} - \hat{s}_{\text{FFT}}}{P_{\text{SRC}}} = \frac{29 + 2 - 3}{12} \approx 2.33\mu\text{s}$$

It follows that the sufficient buffer capacity for a FIFO buffer between the tasks t_{FFT} and t_{CHEST} is three. The sufficient numbers of initial tokens for the other edges are all one, except for the edge $e_{\text{EQ,CHEST}}$, on which zero initial tokens suffice. Note that the FIFO buffer between the tasks t_{CHEST} and t_{EQ} must contain at least $0 + 2 = 2$ locations, as the number of initial tokens on the edge $e_{\text{CHEST,EQ}}$ is two, corresponding to two initially full locations.

Now that we have detailed the different steps of our analysis approach with a post-analysis buffer sizing for a source period of $10\mu\text{s}$ let us also consider an iterative buffer sizing for FIFO buffers with both blocking and non-blocking writes. Furthermore, let us assume that the source period is not fixed. Instead we try to determine the lowest period for each analysis approach such that the respective flow converges, i.e. no constraint is violated. And besides the scenario with a strictly periodic source we also consider the case that the source has an additional jitter of twice its source period. We apply our integrated temporal analysis and buffer sizing approach for the different source periods and, in case no constraint is violated, compute the smallest sufficient buffer capacities δ_1 to δ_8 for both buffers with blocking and non-blocking writes.

The results obtained from the application of our different analysis approaches are presented in Table 10.2. In total we differ between four different analysis approaches, that is, period-and-jitter (p), period-and-jitter with the consideration of cyclic dependencies to limit interference (p.c), period-and-jitter with cyclic dependencies and an iterative buffer sizing (p.c.i), as well as iterative buffer sizing with additionally inserted synchronization edges (p.c.i.s). Furthermore, we apply all approaches assuming buffers with blocking (b) or non-blocking (n) writes. For the synchronization edges we assume the same type as for the buffers, i.e. if blocking write buffers are used, then also blocking write synchronization edges are assumed, whereas non-blocking write buffers imply non-blocking synchronization edges. For each of the approaches we have determined the lowest source period for which it converges, and that for both scenarios with or without source jitter. In addition to the minimum source periods we have also computed the maximum end-to-end latency $\hat{L}$ and the total buffer capacities, i.e. the total numbers of initial tokens Δ for each approach.

The most obvious conclusion we can draw is that the best analysis results are obtained with the consideration of cyclic data dependencies, an iterative buffer sizing and synchronization edges, whereas period-and-jitter only with

$P_{\text{SRC}} / \mu\text{s}$		17	13	12	10	9	8	17	13	12	10	9	8
$J_{\text{SRC}} / \mu\text{s}$					0			34	26	24	20	18	16
p	$\hat{L}$	34	cv	cv	cv	cv	cv	129	cv	cv	cv	cv	cv
	Δ^n	17						39					
	Δ^b	13						17					
p.c	$\hat{L}$	25	25	25	cv	cv	cv	97	89	65	cv	cv	cv
	Δ^n	14	17	18				33	34	40			
	Δ^b	12	13	13				14	14	17			
p.c.i.n	$\hat{L}^n$	12	19	19	20	cv	cv	93	85	61	cv	cv	cv
	Δ^n	11	14	14	16			33	33	39			
p.c.i.s.n	$\hat{L}^n$	12	12	12	20	cv	cv	93	85	61	cv	cv	cv
	Δ^n	11	12	12	16			33	33	39			
p.c.i.b	$\hat{L}^b$	12	19	19	19	19	cv	76	68	44	43	37	cv
	Δ^b	11	12	12	13	13		13	13	13	14	15	
p.c.i.s.b	$\hat{L}^b$	12	12	12	12	12	14	76	68	44	36	30	30
	Δ^b	11	12	12	12	12	13	13	13	13	14	14	15
sim.n	$\hat{L}^n$	12	12	12	12	12	14	32	33	35	39	51	cv
	Δ^n	11	12	12	12	12	15	32	32	33	40	∞	
sim.b	$\hat{L}^b$	12	12	12	12	12	14	42	42	42	29	29	28
	Δ^b	11	12	12	12	12	13	13	13	13	14	14	15

Table 10.2: Latencies $\hat{L}$ in μs and total buffer capacities Δ in numbers of locations determined for Figure 10.8 (Legend: p = period-and-jitter, c = consideration of cyclic dependencies to limit interference, i = iterative buffer sizing, s = synchronization edges, b = blocking write buffers, n = non-blocking write buffers, sim = simulation, cv = constraint violation).

post-analysis buffer sizing performed the worst. With period-and-jitter alone all source frequencies below $17\mu\text{s}$ result in a constraint violation, while an additional consideration of cyclic data dependencies allows for source periods of down to $12\mu\text{s}$, an iterative buffer sizing of blocking write buffers even for source periods down to $9\mu\text{s}$ and the addition of synchronization edges for source periods down to $8\mu\text{s}$. With throughput being defined as symbols per period, this translates to 41%, 89% and 113% higher guaranteeable throughputs, respectively, compared to the period-and-jitter only case.

For the same source periods also lower end-to-end latencies and lower total buffer capacities can be determined with an iterative buffer sizing. For instance for $P_{\text{SRC}} = 17\mu\text{s}$ we can compute a maximum end-to-end latency of $12\mu\text{s}$ and total buffer capacities of 11 locations using the iterative buffer sizing for blocking write buffers, for period-and-jitter with cyclic data dependencies we determine a latency of $25\mu\text{s}$ at 12 locations, while for period-and-jitter alone we can only

determine a maximum end-to-end latency of $34\mu\text{s}$ at 13 locations. The consideration of cyclic data dependencies consequently gives us a 26% lower end-to-end latency at 8% lower space requirements, while the iterative buffer sizing (both with or without synchronization edges) even results in a 65% lower latency at 15% less space.

Note that the iterative buffer sizing for buffers with blocking writes does not only analyze the application, but actively influences the scheduling of tasks. This is most apparent for the scenario with source jitter. Here, the approaches without iterative buffer sizing for blocking write buffers perform quite badly, whereas the iterative buffer sizing for blocking write buffers can achieve the same low source periods as in the scenario without source jitter. The reason for this advantage is that the lower blocking write buffer capacities determined with the iterative buffer sizing act as a kind of traffic shaper that reduces the influence of the source jitter and thereby also interference. This advantage is amplified if synchronization edges are inserted, enabling even lower source periods.

Interestingly, even for non-blocking write buffers, the iterative buffer sizing gives better results than period-and-jitter with cycles and a post-analysis buffer sizing. The reason for this difference is exclusively due to a better analysis accuracy, as non-blocking write buffers cannot act as traffic shapers, i.e. they cannot influence the scheduling of tasks. The higher analysis accuracy is achieved because the iteratively determined buffer capacities serve as an alternative interference characterization to period-and-jitter that can result in tighter maximum response time estimates. And again, the same effect is even more amplified if synchronization edges are inserted. This can be for instance seen on the source period of $13\mu\text{s}$ for which the end-to-end latency of $19\mu\text{s}$ can be further reduced down to $12\mu\text{s}$, only due to the insertion of synchronization edges as an additional modeling construct to characterize interference.

Overall, we can conclude from this case study that the consideration of cyclic data dependencies for limiting interference, the iterative buffer sizing and the insertion of synchronization edges all give us a better analysis accuracy in terms of higher throughput guarantees, lower end-to-end latencies and that at lower space requirements, compared to state-of-the-art.

However, in the last four lines of Table 10.2 we have also inserted the maximum end-to-end latencies and throughput guarantees as determined via simulation according to our simulation approach presented in Chapter 13 (using the buffer capacities as determined via analysis). While these values, in contrast to the values obtained via analysis, are not to be understood as guarantees, they nevertheless show that even between our best analysis results and the respective simulation results there remains a significant gap in accuracy. Closing this gap even further is subject to the next chapter.

10.7.2 SYNTHETIC APPLICATION

In this section we discuss the task graph of a synthetic application, with a focus on the various consequences of using an iterative buffer sizing.

Figure 10.9: Task graph of a synthetic application. Case 1: Task t_d has a higher priority than task t_b , Case 2: Task t_b has a higher priority than task t_d .

Figure 10.9 depicts the task graph, again using edges with tokens to represent buffers for better readability. The tasks are enabled by a periodic source t_s with the period P_s being either $12\mu s$, $11\mu s$ or $10\mu s$. The tasks are executed on three different processors, which is indicated by the different colors of vertices, and the BCETs and WCETs of tasks are denoted next to the vertices. The tasks t_b and t_d are executed on a shared processor with an SPP scheduler. We first assume that task t_d has a higher priority than task t_b and later reverse these priorities. The FIFO buffers between tasks t_a and t_b and between tasks t_c and t_d are buffers with blocking writes and fixed capacities of two and one, respectively. The capacity of the FIFO buffer between t_b and t_c remains to be determined and is denoted with δ .

For the different source periods we apply our integrated temporal analysis and iterative buffer sizing approach, calculate the smallest sufficient buffer capacity δ and determine whether any constraints are violated. We assume that the buffer between tasks t_b and t_c either blocks or does not block on writes and compare our results to the results obtained with the analysis flow with a post-analysis buffer sizing, in which unknown buffer capacities are assumed to be infinite during temporal analysis.

For the first case, in which task t_d has a higher priority than task t_b , the results are presented in Table 10.3. It can be seen that if our iterative buffer sizing is applied, both the resulting buffer capacity δ and the maximum response time of task t_b are smaller or equal compared to the results of a post-analysis buffer sizing for all source frequencies. This is due to the fact that a consideration of the iteratively computed estimates on buffer capacities in the interference calculation leads to an accuracy improvement of temporal analysis.

For instance, the interference calculated for the period $P_s = 12\mu s$ and buffers with non-blocking writes is smaller if estimates on the buffer capacity δ are considered during temporal analysis. This can be seen by comparing the parameters of the minimum function in $\gamma_{d \rightarrow b}(\Delta\tau, q)$ from Equation 10.10. On convergence of the analysis flow, the function $\zeta_{d \rightarrow b}(q)$ from Equation 10.10, which bounds interference based on cyclic data dependencies, resolves to the following (with $\hat{R}_b = \hat{w}'_b(0)$):

$$\zeta_{d \rightarrow b}(0) = \delta + 1 - 1 = \left\lceil \frac{\hat{s}_c + \hat{R}_c - \hat{s}_b}{P_s} \right\rceil = \left\lceil \frac{21}{12} \right\rceil = 2$$

Buffer Sizing Buffer Type	Post-Analysis		Iterative	
	Non-blocking	Blocking	Non-blocking	Blocking
$P_s = 12\mu\text{s}$	$\delta = 2$ $\hat{R}_b = 15\mu\text{s}$	$\delta = 2$ $\hat{R}_b = 15\mu\text{s}$	$\delta = 2$ $\hat{R}_b = 12\mu\text{s}$	$\delta = 1$ $\hat{R}_b = 9\mu\text{s}$
$P_s = 11\mu\text{s}$	constraint violation	constraint violation	$\delta = 2$ $\hat{R}_b = 12\mu\text{s}$	$\delta = 1$ $\hat{R}_b = 9\mu\text{s}$
$P_s = 10\mu\text{s}$	constraint violation	constraint violation	constraint violation	$\delta = 2$ $\hat{R}_b = 12\mu\text{s}$

Table 10.3: Buffer sizing results for Figure 10.9.

This is smaller than the jitter-based bound on interference $\eta_d(\Delta\tau)$ from Equation 10.4:

$$\eta_d(\hat{w}'_b(0)) = \left\lceil \frac{\hat{J}_d + \hat{w}'_b(0)}{P_s} \right\rceil = \left\lceil \frac{17 + 12}{12} \right\rceil = 3$$

In case of a post-analysis buffer sizing, the buffer capacity δ is assumed to be infinite during temporal analysis. The interference is therefore only bounded by $\eta_d(\hat{R}_b)$, resulting in a decreased accuracy compared to an iterative buffer sizing.

Moreover, according to Equation 10.15, the capacity δ computed for a buffer with blocking writes is always smaller or equal to the capacity of a buffer with non-blocking writes, which also results in smaller or equal interference. For instance, for the period of $P_s = 11\mu\text{s}$ it holds on convergence if a buffer with blocking writes is used:

$$\delta = \left\lceil \frac{\hat{s}_c + \hat{R}_c - \hat{s}_b}{P_s} \right\rceil = \left\lceil \frac{17 + 2 - 8}{11} \right\rceil = 1$$

This is obviously smaller than the computed convergent capacity for the case of a buffer with non-blocking writes:

$$\delta = \left\lceil \frac{\hat{s}_c + \hat{R}_c - \hat{s}_b}{P_s} \right\rceil = \left\lceil \frac{20 + 2 - 1}{11} \right\rceil = 2$$

Remember that if buffers with blocking writes are used, then the iterative buffer sizing does not only realize an accuracy improvement in the analysis model, but an actual limitation of interference between tasks. Whenever a buffer with blocking writes has a smaller capacity than a sufficiently large buffer with non-blocking writes, the interference between the tasks connected by the buffer is effectively reduced. For instance, the buffer with blocking writes of the capacity $\delta = 1$ makes the executions of tasks t_b and t_c mutually exclusive, from which follows that task t_d can preempt task t_b only once per execution of task t_a .

These observations allow us to explain the remaining results as well: All buffer sizing approaches converge without a violation of constraints for the source

period $P_s = 12\mu\text{s}$, with the buffer capacity δ being the smallest for an iterative buffer sizing of a buffer with blocking writes. For $P_s = 11\mu\text{s}$, the methods with a post-analysis buffer sizing report a violation of constraints. For $P_s = 10\mu\text{s}$ only the combination of iterative buffer sizing and a buffer with blocking writes results in convergence without violation of constraints.

To conclude this case study, let us finally assume that the priorities of tasks t_b and t_d are reversed, such that task t_b has a higher priority than task t_d . For the source period of $P_s = 10\mu\text{s}$ a violation of temporal constraints can be observed for all four buffer sizing techniques. To be more precise, the maximum response time of the task t_d is computed as $\hat{R}_d = 21\mu\text{s}$ for the post-analysis buffer sizing techniques, $\hat{R}_d = 15\mu\text{s}$ for the iterative buffer sizing of a buffer with non-blocking writes and $\hat{R}_d = 9\mu\text{s}$ for the iterative buffer sizing of a buffer with blocking writes. Although all these maximum response times are too large for the temporal constraint imposed by the rightmost cycle, the results still indicate that the iterative buffer sizing for both buffers with blocking and non-blocking writes can lead better analysis results, no matter how the priorities of tasks are distributed. Finally, for the periods $P_s = 11\mu\text{s}$ and $P_s = 12\mu\text{s}$ only the iterative buffer sizing of a buffer with blocking writes leads to convergence without a violation of constraints, with $\delta = 1$ and $\hat{R}_d = 9\mu\text{s}$.

Execution Interval Analysis

In this chapter we discuss our second analysis technique, the execution interval analysis for single-phase applications. In terms of Section 2.2, this technique is more effective than the period-and-jitter technique from the previous chapter, resulting in a higher analysis accuracy.

The remainder of this chapter is structured as follows: Section 11.1 presents the analysis flows for this analysis technique. Section 11.2 recaps the computation of bounds on schedules using best-case and worst-case analysis models, which is identical to the method described in the previous chapter. In Section 11.3 we discuss the computation of maximum response times. We characterize higher priority interference by the respective execution intervals, as well as an explicit consideration of precedence constraints. The former replaces the period-and-jitter characterization and the latter the cyclic dependencies from the previous chapter. Section 11.4 presents equations for buffer sizing in the context of our execution interval analysis technique, Section 11.5 a proof of conservativeness and Section 11.6 finally evaluates the technique in a case study.

11.1 ANALYSIS FLOWS

In this section we present the analysis flows that we consider for the execution interval analysis approach. Essentially, the flows are the same as the ones used in our period-and-jitter analyses.

11.1.1 ANALYSIS WITH POST-ANALYSIS BUFFER SIZING

Figure 11.1 depicts the flow of our execution interval analysis approach with a post-analysis buffer sizing. We use this analysis flow for the verification of temporal constraints and for the computation of buffer capacities that allow for a satisfaction of these constraints.

Figure 11.1: Overview of the analysis flow with a post-analysis buffer sizing.

The inputs passed to our analysis flow are the same as for the period-and-jitter flow depicted in Figure 11.1, i.e. the single-phase analysis models that are derived from a single-phase application according to the methods described in Chapter 9. The minimum and maximum firing durations in the best-case and worst-case models are initially set to the respective lower and upper bounds of task execution times, that is, the BCETs and WCETs. For buffers whose capacities are to be determined with this flow, the respective numbers of empty buffer locations are set to infinity during analysis.

Two periodic schedules are derived from the best-case and worst-case HSDF graphs in step 1. Based on these schedules we compute lower and upper bounds on the (external) enabling and finish times of the tasks in the underlying response time analysis model. As such, we can use these bounds to define the input intervals of the response time analysis model in step 2 and deduce from them so-called execution intervals that bound the executions of task iterations between minimum enabling and maximum finish times. Based on execution intervals, as well as an explicit consideration of precedence constraints that are derived from the precedence constraint analysis model, we determine maximum response times of tasks. In turn, the maximum response times are used as firing durations for the worst-case analysis model in later iterations of the analysis flow.

In step 3 the best-case and worst-case schedules are checked against temporal constraints and it is verified whether all maximum response times have converged, i.e. did not change since the previous iteration of the algorithm. If a constraint is violated then the algorithm stops. Otherwise, depending on whether the maximum response times have converged, the algorithm either finishes, or repeats the steps 1 to 3 until either convergence or constraint violation is observed.

Given convergence, the determined bounds on enabling and finish times of tasks can be also used to compute buffer capacities in a post-analysis buffer sizing that are sufficiently large to not affect the temporal and functional behavior of the application.

As all our analysis flows, also this flow is iterative, which is both due to the mutual dependency between bounds on schedules and maximum response times (outer loop) and due to the consideration of interference within the maximum response time computations (inner loop). The computational complexity of our flow is in the worst-case non-polynomial. Nevertheless, the run-times are usually small enough for an off-line algorithm, as shown at the end of our case study.

In Appendix B.2 we prove that also this analysis flow terminates, given the specification of upper bounds on end-to-end latencies for all task graphs.

11.1.2 ANALYSIS WITH ITERATIVE BUFFER SIZING

Figure 11.2 presents the analysis flow of our execution interval analysis approach with an iterative buffer sizing.

Figure 11.2: Overview of the analysis flow with an iterative buffer sizing.

For the iterative buffer sizing the computation of buffer capacities is simply drawn inside the iterative loop of the analysis flow, such that estimates of buffer capacities are determined in each flow iteration. In the worst-case analysis model we do not use such estimates, which is required to ensure termination of the flow. The iteratively computed estimates are included in the precedence constraint analysis model, such that they can be used in the maximum response time computations to limit interference. As for period-and-jitter, this method results in a higher analysis accuracy, larger throughput guarantees and smaller buffer capacities. And given that buffers with blocking writes are used, the iterative buffer sizing does not only increase accuracy, but actually optimizes the underlying application to meet a certain throughput guarantee. Finally, all these positive effects can be further amplified if synchronization edges according to Section 10.5 are inserted.

In Appendix B.2 we prove that the execution interval analysis flow with an iterative buffer sizing also terminates if upper bounds on all iteratively determined buffer capacities are specified.

11.2 DETERMINING BOUNDS ON TASK SCHEDULES

178

In this section we present our method to compute bounds on the schedules of tasks in step 1 of the execution interval analysis flows. These bounds define the input intervals in the response time analysis model and thereby form the basis for the execution intervals used to constrain interference. Following the method presented in the previous chapter we derive these bounds from the best-case and worst-case analysis models which are both HSDF graphs for the single-phase applications considered in this chapter.

To determine lower bounds on the schedules, i.e. on the (external) enabling times and finish times of tasks in step 1 of the analysis flows we consider the best-case analysis model in which each task from the response time analysis model is modeled as a dataflow actor, according to Section 9.3.2. The firing durations $\check{\rho}$ of actors in the best-case model are set to the BCETs of the corresponding tasks. Furthermore, we take into account that cyclic data dependencies with positive and finite numbers of tokens can delay periodic start times of actors, while it can occur that the corresponding tasks do not always experience the same delays. Consequently, we only consider edges without initial tokens in the best-case model. Taking all this into account, it has been shown in [HWGB13] that the following LP can be used to determine start times of actors in the best-case model (with $\check{V}$ the set of actors in the best-case analysis model and $\check{E}$ the respective set of edges, which contains all data dependencies from the response time analysis model that do not contain initial tokens):

$$\text{Minimize: } \sum_{\check{v}_a \in \check{V}} \check{s}_a \quad (11.1)$$

$$\text{Subject to: } \check{s}_s = 0, \forall_{e_{a,b} \in \check{E}}: \check{s}_b - \check{s}_a \geq \check{\rho}_a$$

These start times define a periodic schedule for the actors in the best-case model, i.e. the start time of an actor $\check{v}_a$ in iteration i is equal to $\check{s}_a + i \cdot P_a$. By setting the start time of the source actor v_s to $\check{s}_s = 0$ it holds that all start times are computed relative to the first enabling of the source actor. As the external enabling time of a task is also defined relative to the first execution of its strictly periodic source it follows according to Section 9.3.2 that we can use the start time $\check{s}_a$ of an actor v_a in the best-case model to determine a periodic lower bound on the external enabling times $\varepsilon_a^{ext}(i)$ of the corresponding task t_a in the response time analysis model, i.e.:

$$\forall_{i \geq 0}: \check{\varepsilon}_a^{ext}(i) = \check{s}_a + i \cdot P_a \leq \varepsilon_a^{ext}(i)$$

As the total enabling times of tasks cannot become smaller than their external ones we can also use the start times of actors as a lower bound on the total enabling times $\varepsilon_a(i)$, such that:

$$\forall_{i \geq 0}: \check{\varepsilon}_a(i) = \check{s}_a + i \cdot P_a \leq \varepsilon_a(i)$$

Analogously, we use the worst-case model to compute upper bounds on the (external) enabling and finish times of tasks in step 1 of the analysis flows. According to Figure 8.1 we consider buffer capacities as cyclic data dependencies in the worst-case model, except for numbers of initial tokens determined with our analysis flows that are set to infinity to ensure both conservativeness and flow termination.

In the first iterations of the analysis flows we set the firing durations $\hat{\rho}$ of actors in the worst-case model to the WCETs of the corresponding tasks. In subsequent flow iterations the firing durations are set to the maximum response times computed in the previous iterations of the analysis flows. According to [HWGB13] the start times of actors in the worst-case model can be computed by solving the following LP (with $\hat{V}$ and $\hat{E}$ the sets of actors and edges in the worst-case analysis model):

$$\begin{aligned} & \text{Minimize: } \sum_{\hat{v}_a \in \hat{V}} \hat{s}_a & (11.2) \\ & \text{Subject to: } \hat{s}_s = 0, \forall_{e_{a,b} \in \hat{E}}: \hat{s}_b - \hat{s}_a \geq \hat{\rho}_a - \delta_{a,b} \cdot P_a \end{aligned}$$

Also note that self-edges with one token are usually used in dataflow models to capture that a corresponding task cannot be enabled before its previous execution has finished. However, we omit self-edges in the worst-case analysis model, which has the advantage that the firing durations of actors (and thus also maximum response times of tasks) are allowed to be larger than source periods. Due to that removal of self-edges the start times computed in the worst-case analysis model do not relate to the total enabling times of tasks in the response time analysis model, but to the respective external enabling times. As all other data dependencies between tasks are exactly captured by the edges between the corresponding actors, we can bound the external enabling times $\varepsilon_a^{ext}(i)$ of a task t_a in the response time analysis model from above as follows:

$$\forall_{i \geq 0}: \varepsilon_a^{ext}(i) \leq \hat{\varepsilon}_a^{ext}(i) = \hat{s}_a + i \cdot P_a$$

With the periodic bounds on external enabling times and the respective constant best-case and worst-case firing durations we can also determine periodic bounds on the finish times $f_a(i)$ of a task t_a , such that:

$$\forall_{i \geq 0}: \check{f}_a(i) = \check{s}_a + \check{\rho}_a + i \cdot P_a \leq f_a(i) \leq \hat{f}_a(i) = \hat{s}_a + \hat{\rho}_a + i \cdot P_a$$

Based on the maximum external enabling times and maximum finish times we can finally also compute a periodic upper bound on the total enabling times $\hat{\varepsilon}_a(i)$, considering that every task execution is either enabled externally or internally by the finish of its preceding execution. With $f_a(-1) = -\infty$ and $\hat{f}_a(-1) = \hat{s}_a + \hat{\rho}_a - P_a$ we obtain:

$$\forall_{i \geq 0}: \varepsilon_a(i) \leq \hat{\varepsilon}_a(i) = \max(\hat{\varepsilon}_a^{ext}(i), \hat{f}_a(i-1)) = \hat{s}_a + \max(0, \hat{\rho}_a - P_a) + i \cdot P_a$$

Given that the firing durations in the best-case and worst-case analysis models adhere to the firing duration criterion in Equation 9.6, the best-case and worst-case analysis models are valid bounding abstractions of the response time analysis model. Later we prove that the firing duration criterion is satisfied if BCETs are used for the best-case model and the maximum response times computed in the next section as firing duration in the worst-case model. This allows us to use the bounds on (external) enabling and finish times determined above as input intervals in the response time analysis model. Finally, note that on convergence of the analysis flows also the schedule criterion in Equation 9.3 is satisfied, which implies that the best-case and worst-case analysis models are also valid bounding abstractions of the application model. In that case all bounds on (external) enabling and finish times do not only apply for the response time analysis model, but also for the underlying application model.

11.3 COMPUTING MAXIMUM RESPONSE TIMES

In this section we present the computation of maximum response times based on the response time analysis model introduced in Section 9.3.1, i.e. an analysis model that is identical to the application model, except for the input intervals that restrict the accepted inputs of all task components.

At first we bound the finish times of single iterations of tasks t_a from above, by considering interference from higher priority tasks that are executed on the same processor as a task t_a in a temporally conservative manner. This means that external enabling and execution times of higher priority tasks are considered in such way that interference on a task t_a is maximized. To bound interference of higher priority tasks we make use of WCETs, so-called execution intervals that define bounds on the minimum enabling and maximum finish times of higher priority task executions, as well as precedence constraints. Then we extend the derived single-iteration bounds to periodic bounds that hold for any task iterations. Finally we use the periodic upper bounds on finish times to derive maximum response times which incorporate WCETs, self-delay and higher priority interference. Hence we can use such maximum response times as firing durations in the worst-case analysis model.

The outline of the remainder of this section is as follows: In Section 11.3.1 we discuss the basic idea of bounding interference using both execution intervals and precedence constraints on an example. In Section 11.3.2 it is stressed that it is not sufficient to consider all interference of higher priority tasks that can occur between the external enabling time of an iteration i and its finish time, since earlier iterations $i - 1, i - 2, \dots$ can also delay the execution of iteration i . A temporally conservative method of considering such self-delay is presented subsequently. In Section 11.3.3 we explain how execution intervals can be used to bound interference and in Section 11.3.4 how the accuracy of this bound on interference can be improved by an explicit consideration of precedence constraints. In Section 11.3.5 a distinction between interference from tasks belonging to the

same and other task graphs is made, which allows for a simplification of the derived interference bound. The transformation of a bound on the finish time of a single iteration to a periodic bound valid for any iteration of task t_a is explained in Section 11.3.6, as is the subsequent computation of maximum response times.

11.3.1 BASIC IDEA

In this section we illustrate the basic idea of combining execution intervals with precedence constraints to bound interference between tasks on shared processors. At first, we show that there exist two different ways of conservatively bounding interference using execution intervals and that only a combination of the two leads to usable results. Then we illustrate how this bound can be combined with an explicit consideration of precedence constraints in order to determine a more accurate interference characterization.

Figure 11.3: Task graph example.

Consider the task graph depicted in Figure 11.3. All tasks are triggered by the periodic source t_s with the period $P_s = 10\mu\text{s}$ and communicate via FIFO buffers. The BCETs and WCETs of the tasks are denoted above them. Moreover, the tasks t_a , t_b and t_c are executed on a shared processor using an SPP scheduler, with task t_a having the lowest priority π_1 and task t_c having the highest priority π_3 . The unlabeled tasks are executed on other, unshared processors and determine the external enabling and finish times of the tasks t_a , t_b and t_c relative to the source. In the following we attempt to compute an as accurate as possible, yet temporally conservative (i.e. pessimistic) bound on the finish time of an iteration i of task t_a . We make use of so-called execution intervals of higher priority task iterations, i.e. intervals being defined by the minimum external enabling and maximum finish times of such task iterations.

Consider the Gantt chart depicted in Figure 11.4. The upper part of the chart shows the execution intervals of all iterations of the higher priority tasks t_b and t_c that lie in the temporal proximity of iteration i of task t_a . The execution intervals are established by the minimum enabling times $\hat{e}$ and maximum finish times $\hat{f}$ of the tasks. As we initially use BCETs and WCETs as firing durations in the best-case and worst-case analysis models, respectively, it follows that the execution intervals are initially determined by the sums of BCETs and WCETs on the paths to the respective tasks, relative to the periodic executions of the source indicated

Figure 11.4: Determining the maximum finish time of an iteration i of task t_a from Figure 11.3.

by $f_s(i)$. For instance, the initial execution interval of iteration i of task t_c is equal to $f_s(i) + [1 + 1, 3 + 1 + 2]\mu s = f_s(i) + [2, 6]\mu s$.

For simplicity, let us assume that iteration i of task t_a is always enabled externally, i.e. $\epsilon_a(i) = \epsilon_a^{ext}(i)$. Furthermore, let us first assume that iteration i of task t_a is externally enabled as late as possible, i.e. it holds that $\epsilon_a^{ext}(i) = \hat{\epsilon}_a^{ext}(i)$. Given this external enabling time, the execution intervals of the tasks t_b and t_c , as well as their WCETs $\hat{\chi}_b$ and $\hat{\chi}_c$, we can bound interference of the tasks in an accurate, yet temporally conservative manner, as we explain in the following.

At first, consider that iteration i of task t_a does not experience any interference, i.e. it holds that its finish time is equal to $\hat{\epsilon}_a^{ext}(i) + \hat{\chi}_a$ (Figure 11.4(a)). Now we see that the execution interval of iteration i of task t_c overlaps with the execution of iteration i of task t_a . Consequently, we incorporate interference from this overlapping iteration maximally, i.e. as the WCET of task t_c (Figure 11.4(b)). Due to this interference the finish time of iteration i of task t_a increases, causing an additional overlap with iteration i of task t_b that we also consider maximally (Figure 11.4(c)). We apply this procedure of iteratively increasing interference until the finish time of iteration i of task t_a converges, i.e. until the point in time at which no new overlaps have to be considered. After considering interference from iteration i of task t_b no other execution intervals of higher priority tasks can overlap with the execution of iteration i of task t_a , making the finish time presented in Figure 11.4(c) an upper bound on the finish time of iteration i of task t_a for the case that it is externally enabled at $\hat{\epsilon}_a^{ext}(i)$.

Now assume that iteration i of task t_a is not externally enabled as late as possible, but as early as possible, i.e. at $\epsilon_a^{ext}(i) = \hat{\epsilon}_a^{ext}(i)$. Applying the same method of iteratively adding interference we find the finish time shown in Figure 11.4(d). As you can see, in this case the finish time of iteration i of task t_a is larger than the finish time for $\hat{\epsilon}_a^{ext}(i)$. This is due to the fact that the additional overlap with

iteration $i - 1$ of task t_b , which is caused by the earlier external enabling, is smaller than the WCET of task t_b .

Such an anomaly on the computed finish time bounds with respect to external enabling times earlier than $\hat{\varepsilon}_a^{ext}(i)$ imposes a problem in terms of computational efficiency: If we want to find a bound on the finish time of a task iteration that is independent of when the iteration is actually externally enabled we do not only have to compute the finish time for one external enabling time, but for all external enabling times between $\check{\varepsilon}_a^{ext}(i)$ and $\hat{\varepsilon}_a^{ext}(i)$. This would make our proposed algorithm practically unusable.

Fortunately, there is another temporally conservative way to bound interference: A higher priority iteration cannot only interfere no more than the WCET of the corresponding task, but also no later than the maximum finish time of that iteration, i.e. no later than until the end of its execution interval. In Figure 11.4(e) this method is applied on the execution intervals of the tasks t_b and t_c : Starting from the external enabling time $\check{\varepsilon}_a^{ext}(i)$ of iteration i of task t_a we iteratively add interference from iterations of the tasks t_b and t_c up to their maximum finish times. In this example, however, there is no time between the execution intervals of the higher priority task iterations. For that reason there is also no execution time left for the WCET of iteration i of task t_a and its finish time consequently converges towards infinity.

Now let us combine the two methods of bounding interference as follows: For all higher priority task iterations whose execution intervals end before or at the maximum external enabling time of iteration i of task t_a we bound interference using their finish times and for all other interfering iterations, i.e. all iterations whose maximum finish times are larger than $\hat{\varepsilon}_a^{ext}(i)$, we bound interference by the corresponding WCETs. For $\varepsilon_a^{ext}(i) = \check{\varepsilon}_a^{ext}(i)$ the finish time depicted in Figure 11.4(f) is computed, which is apparently smaller than the finish time computed for $\varepsilon_a^{ext}(i) = \hat{\varepsilon}_a^{ext}(i)$. As we show later, the latter observation does not only hold for this example and not only for the case that iteration i of task t_a is externally enabled at its minimum external enabling time, but for any external enabling time smaller than $\hat{\varepsilon}_a^{ext}(i)$. For that reason we can compute an upper bound on the finish time of iteration i of task t_a by only computing an upper bound assuming that iteration i of task t_a is externally enabled at $\hat{\varepsilon}_a^{ext}(i)$ and know that this bound is indeed an upper bound for any enabling time smaller or equal to $\hat{\varepsilon}_a^{ext}(i)$.

Moreover, we can combine the consideration of interference based on execution intervals with an explicit consideration of precedence constraints: As it can be seen in Figure 11.3 task t_a precedes task t_b in such way that an iteration i of task t_a must have always finished before iteration i of task t_b is externally enabled. From this follows that iteration i of task t_b cannot interfere with iteration i of task t_a , independent of any overlaps with the execution interval of iteration i of task t_b . Considering this relation in the computation of maximum interference just results in the finish time of iteration i of task t_a presented in Figure 11.4(g). In Section 11.3.4 we show how to use the precedence constraint analysis model

to derive precedence constraints between tasks for the finish time computations in a consistent way, resulting in more accurate analysis results than achievable by only considering execution intervals.

Finally, note that the determined difference between maximum external enabling and maximum finish time of iteration i of task t_a is larger than the WCET of task t_i . This increased response time of iteration i of task t_a results in an additional delay of iteration i of task t_b that has to be accounted for in a larger execution interval. Larger execution intervals, however, can lead to even more interference that must be considered. Therefore we have to compute execution intervals and maximum response times alternately until convergence is achieved, which is the reason why our execution interval analysis flows are iterative.

11.3.2 HANDLING SELF-DELAY

Remember that the finish time of a task iteration in the response time analysis model cannot be only delayed due to interference of higher priority tasks, but also due to previous iterations of the same task which have not finished before the iteration is externally enabled. For that reason it is not sufficient to compute an upper bound on the finish time of an iteration i of a task t_a with respect to its own external enabling time only. Instead, we determine upper bounds on the finish times of an iteration i with respect to its own external enabling time and the external enabling times of all preceding iterations $i - 1, i - 2, \dots$. Computing the maximum of all these bounds then results in an upper bounds on the finish time of iteration i that is temporally conservative regarding both interference of higher priority tasks and self-delay. To obtain such bounds we make use of maximum busy periods again, which can be formally defined as follows:

Definition 55 (Maximum Busy Periods). *A maximum busy period $\hat{w}_a(\epsilon_a^{ext}(i-q), q)$ defines an upper bound on the time between the external enabling of an iteration $i-q$ and the finish time of an iteration i of a task t_a under the following two assumptions:*

- *Iteration $i - q$ is not delayed by its preceding iteration, i.e. it holds that $f_a(i - q - 1) \leq \epsilon_a^{ext}(i - q)$.*
- *All $q + 1$ iterations $i - q \dots i$ of task t_a are in consecutive execution, i.e. it holds for all iterations i' with $i - q \leq i' < i$ that $f_a(i') \geq \epsilon_a^{ext}(i' + 1)$.*

Note that this definition of maximum busy periods is in line with the definition given in [PGH98], with the difference that in [PGH98] maximum busy periods begin with the critical instant, i.e. the earliest point in time before or at the external enabling of a task iteration $i - q$ after which in the worst-case no lower priority task of task t_a can execute until the end of the maximum busy period. In contrast, our maximum busy periods begin at the external enabling time of a task iteration $i - q$, which is potentially later than the critical instant. This is allowed as we use execution intervals instead of the external enabling intervals considered in [PGH98] to characterize interference.

If for a maximum busy period $\hat{w}_a(\varepsilon_a^{ext}(i-q), q)$ both assumptions hold it follows by definition that $\varepsilon_a^{ext}(i-q) + \hat{w}_a(\varepsilon_a^{ext}(i-q), q)$ is a temporally conservative upper bound on the finish time $f_a(i)$ of task t_a in the response time analysis model. It can be seen that for each task iteration i there must exist at least one $q \geq 0$ for which both assumptions hold. Moreover, it holds for the external enabling time $\varepsilon_a^{ext}(i-q)$ of a task t_a in the response time analysis model:

$$\varepsilon_a^{ext}(i-q) \in [\check{\varepsilon}_a^{ext}(i-q), \hat{\varepsilon}_a^{ext}(i-q)]$$

Note that this range of external enabling times is defined by the input intervals obtained from the best-case and worst-case analysis models. Considering this range it can be seen that a temporally conservative bound on the finish time of iteration i of task t_a with respect to both interference of higher priority tasks and self-delay can be computed as follows:

$$\hat{f}_a(i) = \max_{q \geq 0} (\max_{\varepsilon_a^{ext}(i-q) \in [\check{\varepsilon}_a^{ext}(i-q), \hat{\varepsilon}_a^{ext}(i-q)]} (\varepsilon_a^{ext}(i-q) + \hat{w}_a(\varepsilon_a^{ext}(i-q), q))) \quad (11.3)$$

Now that we have established a relation between maximum busy periods and maximum finish times we focus on the derivation of an accurate, yet temporally conservative function to compute maximum busy periods for any external enabling times and numbers of consecutive executions. After deriving such a function we show that Equation 11.3 can be simplified in such way that it becomes computable, even though the number of possible external enabling times within $[\check{\varepsilon}_a^{ext}(i-q), \hat{\varepsilon}_a^{ext}(i-q)]$ may be very large in general.

11.3.3 BOUNDING INTERFERENCE USING EXECUTION INTERVALS

A maximum busy period $\hat{w}_a(\varepsilon_a^{ext}(i-q), q)$ has to be temporally conservative. This means that its value must be large enough to accommodate both the WCETs of $q+1$ consecutive executions of task t_a , as well as all interference of higher priority tasks that can occur between the external enabling time of the first of the $q+1$ consecutive executions, $\varepsilon_a^{ext}(i-q)$, and the finish time of the last, $\varepsilon_a^{ext}(i-q) + \hat{w}_a(\varepsilon_a^{ext}(i-q), q)$. We account for the execution time of task t_a by simply adding $q+1$ times its WCET $\hat{\chi}_a$ to the maximum busy period. To account for interference of higher priority tasks $t_b \in hp_a$ we make use of execution intervals that we obtain from the input intervals. Remember that the latter are defined by the bounds resulting from the best-case and worst-case analyses, as discussed in Section 11.2. Formally, we can define execution intervals as follows:

Definition 56 (Execution Intervals). *The execution interval $\mathcal{E}_b(j)$ of an iteration j of a task t_b is defined as the interval between its minimum enabling time and its maximum finish time, i.e.:*

$$\mathcal{E}_b(j) = [\check{\varepsilon}_b(j), \hat{f}_b(j)] = [\check{s}_b + j \cdot P_b, \hat{s}_b + \hat{\rho}_b + j \cdot P_b]$$

In the following we use the notation $t_b(j) \in \hat{w}_a(\varepsilon_a^{ext}(i - q), q)$ to indicate that an iteration j of a task t_b is considered as interference in $\hat{w}_a(\varepsilon_a^{ext}(i - q), q)$.

As execution intervals are defined by the temporally conservative minimum enabling and maximum finish times of task iterations it can be seen that only task iterations can interfere with any of the $q + 1$ iterations of task t_a whose execution intervals overlap with the maximum busy period. Therefore we only have to account for higher priority task iterations that adhere to the following two constraints:

$$\begin{aligned} t_b(j) &\in \hat{w}_a(\varepsilon_a^{ext}(i - q), q) \\ \Rightarrow \hat{f}_b(j) &> \varepsilon_a^{ext}(i - q) \wedge \check{e}_b(j) < \varepsilon_a^{ext}(i - q) + \hat{w}_a(\varepsilon_a^{ext}(i - q), q) \end{aligned}$$

In Section 11.3.1 we have discussed that there are two ways of conservatively modeling interference of higher priority tasks using execution intervals. Following the method devised in that section we divide all iterations j of tasks t_b that can interfere with $\hat{w}_a(\varepsilon_a^{ext}(i - q), q)$ into two groups: Interference from task iterations whose maximum finish times are smaller or equal to $\hat{\varepsilon}_a^{ext}(i - q)$ is considered via maximum finish times, whereas interference from all other iterations is considered via the WCETs of the corresponding tasks. Using this distinction we can also split the busy period $\hat{w}_a(\varepsilon_a^{ext}(i - q), q)$ into two parts, the first only considering interference from the first group and the second only considering interference from the second group:

$$\hat{w}_a(\varepsilon_a^{ext}(i - q), q) = \hat{w}_a^*(\varepsilon_a^{ext}(i - q)) + \hat{w}_a^\diamond(q) \quad (11.4)$$

For the iterations in the first group it holds by definition that their maximum finish times are all smaller or equal to the maximum external enabling time $\hat{\varepsilon}_a^{ext}(i - q)$ of task t_a , i.e.:

$$\hat{f}_b(j) \leq \hat{\varepsilon}_a^{ext}(i - q)$$

Therefore none of these iterations can interfere with any of the $q + 1$ iterations of task t_a after $\hat{\varepsilon}_a^{ext}(i - q)$. If we additionally substitute $\hat{\varepsilon}_a^{ext}(i - q)$ with the periodic bound from the worst-case analysis model we can bound interference from the first group of higher priority task iterations with the following function:

$$\hat{w}_a^*(\varepsilon_a^{ext}(i - q)) = \hat{\varepsilon}_a^{ext}(i - q) - \varepsilon_a^{ext}(i - q) = \hat{s}_a + (i - q) \cdot P_a - \varepsilon_a^{ext}(i - q) \quad (11.5)$$

Note that by using $\hat{w}_a^*(\varepsilon_a^{ext}(i - q))$ to bound interference we implicitly assume that the $q + 1$ iterations of task t_a , as well as any higher priority task iterations with a finish time larger than $\hat{\varepsilon}_a^{ext}(i - q)$ do not execute before $\hat{\varepsilon}_a^{ext}(i - q)$. For that reason we have to define the function $\hat{w}_a^\diamond(q)$ in such way that it takes the WCETs of $q + 1$ iterations of task t_a into account, as well as all interference of higher priority task iterations whose maximum finish times are larger than the maximum external enabling time of iteration $i - q$. For the higher priority task iterations that must be considered in $\hat{w}_a^\diamond(q)$ it follows:

$$t_b(j) \in \hat{w}_a^\diamond(q) \Rightarrow \hat{f}_b(j) > \varepsilon_a^{ext}(i - q) \wedge \check{e}_b(j) < \hat{\varepsilon}_a^{ext}(i - q) + \hat{w}_a^\diamond(q)$$

Replacing the minimum and maximum external enabling times and the maximum finish times with the periodic bounds obtained from the best-case and worst-case analysis models results in the following criterion for the higher priority task iterations from the second group:

$$\begin{aligned}
 t_b(j) &\in \hat{w}_a^\diamond(q) & (11.6) \\
 \Rightarrow \hat{s}_b + \hat{\rho}_b + j \cdot P_b &> \hat{s}_a + (i - q) \cdot P_a \wedge \check{s}_b + j \cdot P_b < \hat{s}_a + \hat{w}_a^\diamond(q) + (i - q) \cdot P_a \\
 \Leftrightarrow \left\lfloor \frac{\hat{s}_a + (i - q) \cdot P_a - \hat{s}_b - \hat{\rho}_b}{P_b} \right\rfloor &< j < \left\lceil \frac{\hat{s}_a + \hat{w}_a^\diamond(q) + (i - q) \cdot P_a - \check{s}_b}{P_b} \right\rceil
 \end{aligned}$$

Note that the last equivalence holds because j must be integer.

Using Equation 11.6 we can summarize all iterations of higher priority tasks $t_b \in hp_a$ that have to be considered in $\hat{w}_a^\diamond(q)$ due to their execution intervals in the so-called execution interval set:

$$\begin{aligned}
 E_{t_b \rightarrow \hat{w}_a^\diamond(q)} \\
 = \{j \mid \left\lfloor \frac{\hat{s}_a + (i - q) \cdot P_a - \hat{s}_b - \hat{\rho}_b}{P_b} \right\rfloor < j < \left\lceil \frac{\hat{s}_a + \hat{w}_a^\diamond(q) + (i - q) \cdot P_a - \check{s}_b}{P_b} \right\rceil\}
 \end{aligned}$$

Before we use such sets to derive a temporally conservative function $\hat{w}_a^\diamond(q)$ we introduce the notion of precedence constraints in the context of interference in the next section, which enables the computation of more accurate bounds on interference than by only considering execution intervals.

11.3.4 BOUNDING INTERFERENCE USING PRECEDENCE CONSTRAINTS

In this section we present a bound on interference that is based on precedence constraints. We make use of so-called precedence sets. Here we only derive these sets intuitively, for a formal derivation of precedence sets please refer to Section 10.3.2.

Reconsider the simple task graph depicted in Figure 8.1 which contains a producing task t_a that is connected to a consuming task t_b via a FIFO buffer of the capacity $\delta_{a,b} + \delta_{b,a}$. Let us assume that both tasks execute on the same processor and that task t_b has a higher priority than task t_a . Based on these assumptions we derive all iterations of task t_b that can occur during one iteration i of task t_a . We do this derivation based on the precedence constraint analysis model. Hence we also do not speak of tasks, task executions and buffers with certain capacities in the following, but of the respective HSDF actors, actor firings and edges with initial tokens.

According to the semantics of HSDF graphs, actor v_b can fire $\delta_{a,b}$ times before it must be enabled by a completed firing of actor v_a . From this follows that all firings j of actor v_b with $j \geq i + \delta_{a,b}$ cannot be enabled before firing i of actor v_a finishes. Moreover, it also holds that actor v_a can fire $\delta_{b,a}$ times before it must be

enabled by a completed firing of actor v_b . From this follows that all firings j of actor v_b with $j + \delta_{b,a} \leq i$ must have finished before firing i of actor v_a is enabled.

Negating both constraints gives us the firings j of actor v_b that can occur during a firing i of actor v_a , despite the precedence constraints between the two:

$$j < i + \delta_{a,b} \wedge j + \delta_{b,a} > i$$

As derived in Section 10.3.2 we can generalize this observation from single edges to paths of edges: We define $\mathcal{P}_{a,b}$ as the set of all directed paths of edges from an actor v_a to an actor v_b and $\delta(\mathcal{P}_{a,b})$ as the minimum number of tokens on any path in $\mathcal{P}_{a,b}$, with $\delta(\mathcal{P}_{a,b}) = \infty$ if $\mathcal{P}_{a,b} = \emptyset$. According to Section 10.3.2 we can then capture all firings j of actor v_b that can interfere with an iteration i of actor v_a despite precedence constraints between the corresponding tasks in the following so-called incoming and outgoing precedence sets:

$$\begin{aligned} P_{v_b \rightarrow v_a(i)}^{\text{out}} &= \{j \mid j < i + \delta(\mathcal{P}_{a,b})\} \\ P_{v_b \rightarrow v_a(i)}^{\text{in}} &= \{j \mid j > i - \delta(\mathcal{P}_{b,a})\} \end{aligned}$$

Note that adding an edge with an infinite number of initial tokens to an HSDF graph does not affect the start times of any actors. This relation is used in the definition of $\delta(\mathcal{P}_{a,b})$ for $\mathcal{P}_{a,b} = \emptyset$ to allow for a temporally conservative consideration of actors without directed paths of edges between them. $\delta(\mathcal{P}_{b,a})$ and $\delta(\mathcal{P}_{a,b})$ can be computed efficiently by applying the Floyd-Warshall algorithm on the precedence constraint analysis model.

Using the analogy between task graphs and dataflow graphs according to Figure 8.1 we can determine the iterations j of task t_b that can execute during an iteration i of task t_a by simply taking the union of incoming and outgoing precedence sets, therewith bounding interference. However, to bound interference on $\hat{w}_a^\diamond(q)$ we do not only require all iterations j of task t_b that can interfere with one iteration i of task t_a , but that can interfere with all $q + 1$ consecutive iterations of task t_a , with iteration $i - q$ being the first. A set containing all these interfering iterations can be obtained by taking the union of the respective outgoing and incoming precedence sets of the iterations $i - q \dots i$ of task t_a and subsequently taking the intersection of the two resulting sets, i.e.:

$$\begin{aligned} P_{t_b \rightarrow \hat{w}_a^\diamond(q)} &= \bigcup_{i'=i-q}^i P_{v_b \rightarrow v_a(i')}^{\text{out}} \cap \bigcup_{i'=i-q}^i P_{v_b \rightarrow v_a(i')}^{\text{in}} \\ &= \{j \mid i - q - \delta(\mathcal{P}_{b,a}) < j < i + \delta(\mathcal{P}_{a,b})\} \end{aligned}$$

Both $E_{t_b \rightarrow \hat{w}_a^\diamond(q)}$ and $P_{t_b \rightarrow \hat{w}_a^\diamond(q)}$ are conservative upper bounds on all iterations of a task t_b that can interfere with $\hat{w}_a^\diamond(q)$. To compute a more accurate bound on interference we can therefore simply draw the intersection of the two sets. Before we do that, however, let us first investigate how the lower bounds of both sets relate to each other.

First, if there is no directed path from an actor v_b to an actor v_a in the precedence constraint analysis model then $\delta(\mathcal{P}_{b,a})$ is infinite, making the lower bound of $E_{t_b \rightarrow \hat{w}_a^\circ(q)}$ always larger than the lower bound of $P_{t_b \rightarrow \hat{w}_a^\circ(q)}$. The lower bound of $E_{t_b \rightarrow \hat{w}_a^\circ(q)}$ therefore becomes the dominant bound when the intersection between the two sets is drawn. And second, if there is at least one directed path $p \in \mathcal{P}_{b,a}$ then it holds that the periods P_a, P_b and the periods of all other tasks on path p must be equal, since the tasks must all belong to the same task graph. By recursively substituting the inequalities from the worst-case LP in Equation 11.2 that are defined for the edges on a path p it follows:

$$\forall p \in \mathcal{P}_{b,a}: \hat{s}_a \geq \hat{s}_b + \sum_{e_{c,d} \in p} (\hat{\rho}_c - \delta_{c,d} \cdot P_d) \geq \hat{s}_b + \hat{\rho}_b - \left(\sum_{e_{c,d} \in p} \delta_{c,d} \right) \cdot P_b$$

As this inequality holds for any path from actor v_b to v_a it must also hold for the path p^* with the minimum number of tokens $\delta(\mathcal{P}_{b,a}) = \sum_{e_{c,d} \in p^*} \delta_{c,d}$.

Moreover, the inequality also applies for numbers of initial tokens on the path that are determined in an iterative buffer sizing, although such numbers are assumed as infinite in the worst-case analysis model. This is due to the fact that iteratively sized numbers of initial tokens are just computed large enough such that the schedules determined with the worst-case LP remain admissible, i.e. they could be added to the LP without increasing any of the calculated start times. From this follows:

$$\begin{aligned} \hat{s}_a \geq \hat{s}_b + \hat{\rho}_b - \delta(\mathcal{P}_{b,a}) \cdot P_b &\Leftrightarrow -\delta(\mathcal{P}_{b,a}) \leq \frac{\hat{s}_a - \hat{s}_b - \hat{\rho}_b}{P_b} \\ &\Leftrightarrow i - q - \delta(\mathcal{P}_{b,a}) \leq \frac{\hat{s}_a + (i - q) \cdot P_a - \hat{s}_b - \hat{\rho}_b}{P_b} \\ &\Leftrightarrow i - q - \delta(\mathcal{P}_{b,a}) \leq \left\lfloor \frac{\hat{s}_a + (i - q) \cdot P_a - \hat{s}_b - \hat{\rho}_b}{P_b} \right\rfloor \end{aligned}$$

Note that the last equivalence holds because the left hand side of the inequality is integer.

This lets us conclude that the lower bound of $E_{t_b \rightarrow \hat{w}_a^\circ(q)}$ is always larger or equal to the lower bound of $P_{t_b \rightarrow \hat{w}_a^\circ(q)}$, independent of whether there is a path from actor v_b to actor v_a or not. Taking this observation into account by ignoring the lower bound of $P_{t_b \rightarrow \hat{w}_a^\circ(q)}$ we can draw the intersection between the two sets in the so-called interference set:

$$\begin{aligned} I_{t_b \rightarrow \hat{w}_a^\circ(q)} &= E_{t_b \rightarrow \hat{w}_a^\circ(q)} \cap P_{t_b \rightarrow \hat{w}_a^\circ(q)} \\ &= \{j \mid \left\lfloor \frac{\hat{s}_a + (i - q) \cdot P_a - \hat{s}_b - \hat{\rho}_b}{P_b} \right\rfloor < j < \min \left(\left\lceil \frac{\hat{s}_a + \hat{w}_a^\circ(q) + (i - q) \cdot P_a - \hat{s}_b}{P_b} \right\rceil, i + \delta(\mathcal{P}_{a,b}) \right) \} \end{aligned}$$

Finally, we are not interested in the particular iterations of tasks $t_b \in hp_a$ that can interfere with $\hat{w}_a^\circ(q)$, but only in the number of iterations. For that reason we define a function $\psi_{b \rightarrow a}(\hat{w}_a^\circ(q), q)$ returning the maximum number of iterations

of a task t_b that can interfere with $\hat{w}_a^\diamond(q)$. Such a function can be determined by computing the number of elements in $I_{t_b \rightarrow \hat{w}_a^\diamond(q)}$ (with $-[k] = \lceil -k \rceil$):

$$\psi_{b \rightarrow a}(\hat{w}_a^\diamond(q), q) = |I_{t_b \rightarrow \hat{w}_a^\diamond(q)}| = \min\left(\left\lceil \frac{\hat{s}_a + \hat{w}_a^\diamond(q) + (i - q) \cdot P_a - \check{s}_b}{P_b} \right\rceil, i + \delta(\mathcal{P}_{a,b})\right) + \left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a - (i - q) \cdot P_a}{P_b} \right\rceil - 1$$

11.3.5 DIFFERING BETWEEN INTERFERENCE FROM THE SAME $\mathcal{G}$ OTHER TASK GRAPHS

We differ between the two cases that a task t_a and a higher priority task $t_b \in hp_a$ belong to the same task graph or not. This additional information can be used to simplify the function $\psi_{b \rightarrow a}(\hat{w}_a^\diamond(q), q)$. With the shorthand notation G_a^T for the task graph of a task t_a we can rewrite $\psi_{b \rightarrow a}(\hat{w}_a^\diamond(q), q)$ as follows:

$$\psi_{b \rightarrow a}(\hat{w}_a^\diamond(q), q) = \begin{cases} \psi_{b \rightarrow a}^*(\hat{w}_a^\diamond(q), q) & G_b^T = G_a^T \\ \psi'_{b \rightarrow a}(\hat{w}_a^\diamond(q)) & \text{otherwise} \end{cases} \quad (11.7)$$

Let us first consider the case that $G_b^T = G_a^T$. As both tasks belong to the same task graph they must also have the same period, i.e. $P_b = P_a$. Hence it holds that the summand $\frac{(i-q) \cdot P_a}{P_b} = i - q$ is integer and we can draw it outside of both ceiling functions. From this follows:

$$\psi_{b \rightarrow a}^*(\hat{w}_a^\diamond(q), q) = \min\left(\left\lceil \frac{\hat{s}_a + \hat{w}_a^\diamond(q) - \check{s}_b}{P_b} \right\rceil, \delta(\mathcal{P}_{a,b}) + q\right) + \left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a}{P_b} \right\rceil - 1$$

Note that with this simplification $\psi_{b \rightarrow a}^*(\hat{w}_a^\diamond(q), q)$ becomes independent of a specific iteration i of task t_a . It is therefore a valid upper bound on the number of interfering iterations of a task t_b for any $\hat{w}_a^\diamond(q)$, no matter for which q particular iterations of task t_a it is determined.

For tasks t_b and t_a belonging to different task graphs $\delta(\mathcal{P}_{a,b})$ is infinite, such that only the execution interval set needs to be considered. However, the external enabling times of tasks belonging to different task graphs are generally uncorrelated, i.e. it is unknown how $\check{s}_b$ and $\hat{s}_b$ on the one hand and $\hat{s}_a$ on the other hand relate to each other. In the worst-case we have to assume that the external enabling times are correlated in such way that the inequality $\lceil k \rceil + \lceil l \rceil - 1 \leq \lceil k + l \rceil$ becomes an equality when applied on $\psi_{b \rightarrow a}(\hat{w}_a^\diamond(q))$. From this follows:

$$\psi'_{b \rightarrow a}(\hat{w}_a^\diamond(q)) = \left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \check{s}_b + \hat{w}_a^\diamond(q)}{P_b} \right\rceil$$

As you can see it holds that $\psi'_{b \rightarrow a}(\hat{w}_a^\diamond(q))$ is also independent of a particular iteration i of task t_a .

11.3.6 FROM A SINGLE ITERATION BOUND TO A PERIODIC BOUND

After having derived the function $\psi_{b \rightarrow a}(\hat{w}_a^\diamond(q), q)$ that returns the maximum number of interfering iterations of a task t_b during $\hat{w}_a^\diamond(q)$ we can now also determine the function $\hat{w}_a^\diamond(q)$ itself. As mentioned before, the function shall return a temporally conservative bound on the time between the maximum external enabling time of an iteration $i - q$ and the finish time of an iteration i of task t_a . For that reason it must consist of both an upper bound on the execution times of task t_a itself, as well as an upper bound on the execution times of all higher priority tasks that can preempt and delay any of the executions of t_a within the interval $[\hat{\varepsilon}_a^{ext}(i - q), \hat{\varepsilon}_a^{ext}(i - q) + \hat{w}_a^\diamond(q)]$. Using the WCETs of the task t_a , of the tasks $t_b \in hp_a$ and the function $\psi_{b \rightarrow a}(\hat{w}_a^\diamond(q), q)$ we can define $\hat{w}_a^\diamond(q)$ as follows:

$$\hat{w}_a^\diamond(q) = (q + 1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \psi_{b \rightarrow a}(\hat{w}_a^\diamond(q), q) \cdot \hat{\chi}_b$$

Note that $\hat{w}_a^\diamond(q)$ is computed iteratively until a fixed point is found. This is required because the larger $\hat{w}_a^\diamond(q)$ becomes, the more higher priority tasks $t_b \in hp_a$ can interfere with $\hat{w}_a^\diamond(q)$, which in turn can result in an even larger $\hat{w}_a^\diamond(q)$.

Next we can substitute $\hat{w}_a(\varepsilon_a^{ext}(i - q), q)$ in Equation 11.3 using Equation 11.4 and $\hat{w}_a^*(\varepsilon_a^{ext}(i - q))$ using Equation 11.5, which results in the following upper bound on the finish time of an iteration i of task t_a in the response time analysis model (with ε a shorthand notation for $\varepsilon_a^{ext}(i - q)$):

$$\begin{aligned} \hat{f}_a(i) &= \max_{q \geq 0} (\max_{\varepsilon \in [\hat{\varepsilon}_a^{ext}(i - q), \hat{\varepsilon}_a^{ext}(i - q)]} (\varepsilon + \hat{w}_a(\varepsilon, q))) = \max_{q \geq 0} (\max_{\varepsilon \in [\hat{\varepsilon}_a^{ext}(i - q), \hat{\varepsilon}_a^{ext}(i - q)]} (\varepsilon + \hat{s}_a + (i - q) \cdot P_a - \varepsilon + \hat{w}_a^\diamond(q))) \\ &= \hat{s}_a + \max_{q \geq 0} (\hat{w}_a^\diamond(q) - q \cdot P_a) + i \cdot P_a = \hat{f}_a + i \cdot P_a \end{aligned}$$

Due to the same reasoning as in [TBW94] a $\hat{w}_a^\diamond(q)$ only has to be considered if it holds that $\hat{w}_a^\diamond(q - 1) > q \cdot P_a$. Furthermore, the dependence on specific external enabling times $\varepsilon_a^{ext}(i - q)$ is removed, making the maximum finish time calculation practically usable. As $\hat{w}_a^\diamond(q)$ is independent of a specific iteration i of t_a it follows that $\hat{f}_a(i)$ is indeed a periodic upper bound on the finish time of a task t_a with respect to both interference of higher priority tasks and self-delay. Finally, we can use this bound to compute the maximum response time of a task t_a in the response time analysis model as follows:

$$\hat{R}_a = \hat{f}_a(i) - \hat{\varepsilon}_a^{ext}(i) = \max_{q \geq 0} (\hat{w}_a^\diamond(q) - q \cdot P_a) \quad (11.8)$$

Such a maximum response time $\hat{R}_a$ of a task t_a in the response time analysis model incorporates upper bounds on the execution time of the task, on self-delay, as well as on any higher priority interference that can occur during the execution of task t_a . Hence we can use $\hat{R}_a$ as the corresponding firing duration $\hat{\rho}_a$ in the worst-case analysis model.

Finally, to guarantee that our flows terminate we require that maximum external enabling times increase monotonically throughout iterations of the analysis flows. Considering the LP in Equation 11.2 it can be seen that this holds if the maximum firing durations, and thus maximum response times, also increase monotonically. The term $-\hat{s}_a$ in the computation of $\psi_{b \rightarrow a}^*(\hat{w}_a^\diamond(q), q)$, however, can lead to decreasing maximum response times. Hence we have to clamp maximum response times. Let $\hat{R}_a^{n-1}$ be the maximum response time computed in iterations $n - 1$ of the analysis flows. With $\hat{R}_a^{-1} = \hat{\chi}_a$ we can compute the maximum response time in iteration n as follows:

$$\hat{R}_a^n = \max(\hat{R}_a^{n-1}, \max_{q \geq 0}(\hat{w}_a^\diamond(q) - q \cdot P_a))$$

11.4 BUFFER SIZING

Both post-analysis and iterative buffer sizings in our analysis flows make use of the equations discussed in Section 10.4, which we repeat here for completeness:

$$\delta_{b,a} = \begin{cases} \max(\left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a}{P_b} \right\rceil, 0) & \text{buffer with non-blocking writes} \\ \max(\left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a}{P_b} \right\rceil, 0) & \text{buffer with blocking writes} \end{cases}$$

We differ between buffers with blocking writes, i.e. buffers that block a writing task whenever the buffer is full, and buffers with non-blocking writes. The clamping for iteratively sized buffers with blocking writes used in the previous chapter is not required here, as we already clamp maximum response times, which suffices to ensure monotonicity of the analysis flow with an iterative buffer sizing.

Finally, note that if an iterative buffer sizing is applied, also synchronization edges according to Section 10.5 can be added to the underlying application model. We show in the following case study that both iterative buffer sizing and the usage of synchronization edges lead to superior analysis results, independent of the buffer type.

11.5 PROOF OF CONSERVATIVENESS

After explaining all steps of our analysis flows it remains to be shown that the computed results are not only valid within the respective analysis models, but also apply for the underlying application model.

At first we prove that the maximum response times computed with Equation 11.8 are conservative upper bounds on the differences between maximum external enabling and finish times in the response time analysis model. According to Section 9.3.2 this implies that if these maximum response times are used as firing durations in the worst-case analysis model it follows that the worst-case analysis model becomes a bounding abstraction of the response time analysis model, i.e.:

Lemma 11.1 (Worst-Case Conservativeness). *Let the firing durations $\hat{\rho}_a$ of actor components $\hat{v}_a$ in the worst-case analysis model be set to the respective $\hat{R}_a$ and let $\hat{\varepsilon}_a^{ext}(i)$ be the strictly periodic upper bounds on the respective external enabling times used in the restricted task components t_a in the response time analysis model. Then it holds that $G^{WC} \supseteq G^{RT}$ and $\text{in}_{G^{WC}} \supseteq \text{in}_{G^{RT}}$, i.e. the worst-case analysis model is a valid worst-case bounding abstraction of the response time analysis model.*

Proof. From the firing duration criterion in Equation 9.6 follows that the worst-case bounding abstraction holds if $\varepsilon_a(i) + \chi_a(i) + \mathcal{I}_a[\varepsilon_a(i), f_a(i)] \leq \hat{\varepsilon}_a^{ext}(i) + \hat{\rho}_a$ applies for all restricted tasks and actors. For the proof of this inequality we consider again the actual busy periods in the response time analysis model that were already introduced in the proof of Lemma 10.3:

$$w_a(i) = \sum_{i'=i-q_i}^i \chi_a(i') + \mathcal{I}_a[\varepsilon_a^{ext}(i-q_i), \varepsilon_a^{ext}(i-q_i) + w_a(i)]$$

Using such actual busy periods we can prove the firing duration criterion as follows:

$$\begin{aligned} \varepsilon_a(i) + \chi_a(i) + \mathcal{I}_a[\varepsilon_a(i), f_a(i)] &= \varepsilon_a(i-q_i) + w_a(i) \\ &= \varepsilon_a^{ext}(i-q_i) + \sum_{i'=i-q_i}^i \chi_a(i') + \mathcal{I}_a[\varepsilon_a^{ext}(i-q_i), \varepsilon_a^{ext}(i-q_i) + w_a(i)] \\ &\leq \varepsilon_a^{ext}(i-q_i) + \hat{w}_a^*(\varepsilon_a^{ext}(i-q_i)) + (q_i+1) \cdot \hat{\chi}_a \\ &\quad + \sum_{t_b \in hp_a} \psi_{b \rightarrow a}(w_a(q_i) - \hat{w}_a^*(\varepsilon_a^{ext}(i-q_i)), q_i) \cdot \hat{\chi}_b \\ &\leq \varepsilon_a^{ext}(i-q_i) + \hat{w}_a^*(\varepsilon_a^{ext}(i-q_i)) + (q_i+1) \cdot \hat{\chi}_a \\ &\quad + \sum_{t_b \in hp_a} \psi_{b \rightarrow a}(\hat{w}_a(\varepsilon_a^{ext}(i-q_i), q_i) - \hat{w}_a^*(\varepsilon_a^{ext}(i-q_i)), q_i) \cdot \hat{\chi}_b \\ &= \hat{s}_a + (i-q_i) \cdot P_a + \hat{w}_a^\diamond(q_i) \leq \hat{\varepsilon}_a^{ext}(i) + \hat{R}_a = \hat{\varepsilon}_a^{ext}(i) + \hat{\rho}_a \end{aligned}$$

From the definition of the busy period, which implies $\varepsilon_a(i-q_i) = \varepsilon_a^{ext}(i-q_i)$, follows equality between the first and the second line. The inequality between the second and third line applies due to the following two reasons: First, it holds that $\forall_i : \chi_a(i) \leq \hat{\chi}_a$. And second, $\hat{w}_a^*(\varepsilon_a^{ext}(i-q_i))$ gives an upper bound on the interference that can occur after $\varepsilon_a^{ext}(i-q_i)$, but not after $\hat{\varepsilon}_a^{ext}(i-q_i) = \hat{s}_a + (i-q_i) \cdot P_a$, while the sum to the right gives an upper bound on the interference that can occur between $\hat{\varepsilon}_a^{ext}(i-q_i)$ and $\varepsilon_a^{ext}(i-q_i) + w_a(q_i) = \hat{\varepsilon}_a^{ext}(i-q_i) + w_a(q_i) - \hat{w}_a^*(\varepsilon_a^{ext}(i-q_i))$.

Between the third and the fourth line we make use of an analogous reasoning about fixed points as already used in the proof of Lemma 10.3: We assume that $w_a(i) = g(w_a(i))$ and $\hat{w}_a(\varepsilon_a^{ext}(i-q_i), q_i) = h(\hat{w}_a(\varepsilon_a^{ext}(i-q_i), q_i))$ are the fixed

points of two functions $g(x)$ and $h(x)$ with:

$$g(x) = \sum_{i'=i-q_i}^i \chi_a(i') + \mathcal{I}_a[\varepsilon_a^{ext}(i-q_i), \varepsilon_a^{ext}(i-q_i) + x]$$

$$h(x) = \hat{w}_a^*(\varepsilon_a^{ext}(i-q_i)) + (q_i + 1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \psi_{b \rightarrow a}(x - \hat{w}_a^*(\varepsilon_a^{ext}(i-q_i)), q_i) \cdot \hat{\chi}_b$$

Due to the same reasoning as used between the second and the third line it holds $\forall_x : g(x) \leq h(x)$. Moreover, with $g(x)$ and $h(x)$ being both monotonically increasing and with $g(0) \geq 0$ we can conclude according to the proof of Lemma 10.3 that also $\forall_i : w_a(i) \leq \hat{w}_a(\varepsilon_a^{ext}(i-q_i), q_i)$.

In the fifth line we substitute $\hat{w}_a^*(\varepsilon_a^{ext}(i-q_i))$ with $\hat{s}_a + (i-q_i) \cdot P_a - \varepsilon_a^{ext}(i-q_i)$ and the right-hand expression from the fourth line with $\hat{w}_a^\circ(q_i)$. Thereafter we replace $\hat{s}_a + i \cdot P_a$ with $\hat{\varepsilon}_a^{ext}(i)$ and use the definition of the maximum response time, which implies $\forall_{q_i \geq 0} : \hat{w}_a^\circ(q_i) - q_i \cdot P_a \leq \hat{R}_a$.

Hence if the maximum response times $\hat{R}_a$ determined in this section are used as firing durations $\hat{\rho}_a$ in the worst-case analysis model it follows that the firing duration criterion is satisfied for all restricted tasks and actors. The latter finally allows us to conclude that $G^{WC} \supseteq G^{RT}$ and $in_{G^{WC}} \supseteq in_{G^{RT}}$. $\square$

Furthermore it needs to hold on convergence of our flows that the response time analysis model is a valid inclusion abstraction of the application model, which in turn implies that all bounds computed with our flows are also valid bounds on the respective parameters in the application model. Hence we prove the following theorem:

Theorem 11.1 (Analysis Conservativeness). *On convergence of our analysis flows it holds that $G^{RT} \supseteq G^\Lambda$ and $in_{G^{RT}} \supseteq in_{G^\Lambda}$, i.e. the response time analysis model is a valid inclusion abstraction of the application model. This implies that on convergence also the worst-case, best-case and precedence constraint analysis models are valid worst-case, best-case and inclusion abstractions of the application model, respectively.*

Proof. Analogous to the proof of Theorem 10.1. $\square$

11.6 CASE STUDY

This section demonstrates the benefits of using execution intervals to limit interference in a case study. At first we discuss the performance of our approach for acyclic applications in more detail, using the example depicted in Figure 11.5 that was already briefly introduced in Section 8.2. Then we address a more complex, cyclic application and relate our approach to other approaches applicable for cyclic applications. Finally, we discuss the run-times of our analysis approach in different scenarios.

Figure 11.5: Left: Task graph with variable source jitter. Right: Source jitter J_s vs. maximum end-to-end latency $\hat{L}$ for different temporal analysis approaches.

Let us reconsider the example from Section 8.2, which we depict again in Figure 11.5 for convenience. It is no surprise that our execution interval analysis approach and MAST compute the same end-to-end latencies for the presented task graph, as both rely on the same principles: Both approaches make use of a notion of dynamic offsets that is combined with an explicit exclusion of interference due to precedence constraints. In fact, sometimes MAST can even have slightly more accurate results than our approach for acyclic applications because it determines the critical instant and makes use of external enabling intervals. We make use of execution intervals and consequently do not need to compute critical instants. While this comes at the cost of a slight over-approximation, it also simplifies the combination with arbitrary (cyclic) precedence constraints and thus allows to perform a more accurate analysis for cyclic applications.

The SymTA/S approach has a significantly worse analysis accuracy than our approach for the given example. This is due to the fact that SymTA/S does not have a notion of explicit interference exclusion, which is especially relevant if large jitters, i.e. jitters that are larger than the source period, are involved. In some cases, however, e.g. if the priorities of tasks t_a and t_c were reversed [P⁺08], the SymTA/S approach can still outperform both our approach and MAST. The reason for this peculiarity is that we make use of periodic bounds, which implies the assumption that bursts of input values occurring due to large input jitters are processed periodically with the source period. Therefore our computed end-to-end latencies cannot be smaller than the input jitter plus the time to process one value. In contrast, SymTA/S can take into account that in some cases a burst of multiple input values can be processed faster than with the source period, leading to smaller end-to-end latencies.

Unlike both MAST and SymTA/S, we can additionally apply an iterative buffer sizing on acyclic task graphs. The iterative buffer sizing introduces backward dependencies for all forward dependencies and thereby makes acyclic task graphs

cyclic, such that approaches like MAST or SymTA/S are not applicable anymore. Both our period-and-jitter and execution interval analysis approaches do not only support backward dependencies, but can even exploit them to further limit interference between tasks. This can lead to more accurate analysis results than both MAST and SymTA/S can achieve.

11.6.2 WLAN 802.11P TRANSCEIVER

Figure 11.6: Task graph of the packet decoder of a WLAN 802.11p transceiver.

This section demonstrates the benefits of our approach for the analysis of cyclic real-time streaming applications. We analyze the task graph of a WLAN 802.11p transceiver [AHG07] that we have already introduced in Section 10.7.1. The considered WLAN 802.11p transceiver has several modes and is executed on a multiprocessor system for performance reasons. Moreover, according to the standard usually two identical transceivers are required, one for a control and one for a data channel, that must be executed in parallel. We only consider the part of the task graph that is active during packet decoding mode. The task graph of the packet decoding mode used in both control and data parts is depicted in Figure 11.6.

A periodic source with a period P_{SRC} models the input of this dataflow graph that we again try to minimize using our different analysis approaches. On top of that, we also consider the scenario that the source has an additional jitter of $J_{\text{SRC}} = 2 \cdot P_{\text{SRC}}$. The BCETs and WCETs of the tasks are denoted next to the corresponding vertices in the graph. Moreover, the task graph contains a feedback loop as the settings of the channel equalizer t_{EQ} for the reception of symbol i are based on an estimate of the channel t_{CHEST} . The channel estimate is in turn based on the received symbol $i - 2$ and the reencoded symbol $i - 2$, which is obtained by reencoding the error-corrected bits of symbol $i - 2$ produced by the viterbi channel decoder t_{VIT} .

Our analysis flows allow for the quick verification of different task-to-processor mappings and priority assignments. In the following we assume that the tasks of both control and data parts of the transceiver are mapped to eight different processors. At first, we consider the case that the control and the data transceiver are executed independent from each other, i.e. the tasks of the control part are mapped to a different set of four processors than the tasks of the data part. One

such mapping on four processors is exemplified in Figure 11.6, with different colors of tasks indicating different processors. If multiple tasks are mapped to a shared processor (have the same color) then they are scheduled by an SPP scheduler, with the priorities denoted as π_1 (lowest) to π_4 (highest).

Due to the cyclic data dependency introduced by the feedback loop none of the offset-based approaches discussed in Section 8.2 is applicable on this example.

Consequently, we can only relate our execution interval approach to the approach from [HWGB13], which is based on period-and-jitter, as well as to the ones from the previous chapter, namely the approach that combines a period-and-jitter interference characterization with an explicit consideration of cyclic data dependencies and the approach that additionally replaces the post-analysis buffer sizing with an iterative one. For comparison we also apply our approach, which combines execution intervals with precedence constraints, with or without using the iterative buffer sizing. Lastly, we also consider the addition of synchronization edges as explained in Section 10.5.

The different analysis approaches are applied for the combinations of source period and jitter already considered in Section 10.7.1. The analysis results are then used to compute maximum end-to-end latencies $\hat{L}$ and sums of all buffer capacities Δ . The respective maximum end-to-end latencies and sums of buffer capacities of the control part are presented in Table 11.1, which extends Table 10.2 from the previous chapter (the results from this case study and the one from the previous chapter are comparable as we consider the same task-to-processor mapping here). Note that for some cases the analyzed latencies grow so large that the throughput constraint imposed by the feedback loop is violated, which is indicated by the “cv” entries in the table.

What immediately catches one’s eye is that the combinations of execution intervals, a consideration of precedence constraints and an iterative buffer sizing (cases e.c.i.n and e.c.i.b) produce the most accurate results, whereas the approach that is solely based on period-and-jitter (case p) generates the worst. The usage of execution intervals allows for a more accurate interference characterization than achievable with period-and-jitter. This is due to the fact that in the period-and-jitter characterization all interfering tasks are assumed to be externally enabled simultaneously, whereas our execution interval approach rules out interference between tasks if no overlap between execution intervals and maximum busy periods is detected.

The effect of using execution intervals on analysis accuracy becomes apparent by comparing the cases p and e: Not only does the execution interval algorithm converge in more cases without a violation of temporal constraints (source periods down to $13\mu\text{s}$ for execution intervals compared to $17\mu\text{s}$ for period-and-jitter, which translates to a 31% higher throughput guarantee, with throughput being defined as symbols per period), but also the computed end-to-end latencies are much smaller (50% smaller for $P_{\text{SRC}} = 80\mu\text{s}$ and $J_{\text{SRC}} = 0\mu\text{s}$). Moreover, our approach with execution intervals and precedence constraints (case e.c) produces consistently more accurate results than the approach combining period-and-jitter with

$P_{\text{SRC}} / \mu\text{s}$		17	13	12	10	9	8	32	28	16	12	9	8
$J_{\text{SRC}} / \mu\text{s}$					0			64	56	32	24	18	16
p	$\hat{L}$	34	cv	cv	cv	cv	cv	129	cv	cv	cv	cv	cv
	Δ^n	17						39					
	Δ^b	13						17					
p.c	$\hat{L}$	25	25	25	cv	cv	cv	97	89	65	cv	cv	cv
	Δ^n	14	17	18				33	34	40			
	Δ^b	12	13	13				14	14	17			
p.c.i.n	$\hat{L}^n$	12	19	19	20	cv	cv	93	85	61	cv	cv	cv
	Δ^n	11	14	14	16			33	33	39			
p.c.i.s.n	$\hat{L}^n$	12	12	12	20	cv	cv	93	85	61	cv	cv	cv
	Δ^n	11	12	12	16			33	33	39			
p.c.i.b	$\hat{L}^b$	12	19	19	19	19	cv	76	68	44	43	37	cv
	Δ^b	11	12	12	13	13		13	13	13	14	15	
p.c.i.s.b	$\hat{L}^b$	12	12	12	12	12	14	76	68	44	36	30	30
	Δ^b	11	12	12	12	12	13	13	13	13	14	14	15
e	$\hat{L}$	17	21	cv	cv	cv	cv	117	112	cv	cv	cv	cv
	Δ^n	12	14					37	39				
	Δ^b	12	12					14	16				
e.c	$\hat{L}$	12	12	12	12	12	14	84	76	52	48	cv	cv
	Δ^n	11	12	12	12	12	15	32	32	33	40		
	Δ^b	11	12	12	12	12	13	13	13	14	17		
e.c.i.n	$\hat{L}^n$	12	12	12	12	12	14	84	79	52	48	cv	cv
	Δ^n	11	12	12	12	12	15	32	32	33	40		
e.c.i.s.n	$\hat{L}^n$	12	12	12	12	12	14	84	79	52	48	cv	cv
	Δ^n	11	12	12	12	12	15	32	32	33	40		
e.c.i.b	$\hat{L}^b$	12	12	12	12	12	14	76	68	44	36	30	30
	Δ^b	11	12	12	12	12	13	13	13	13	14	14	15
e.c.i.s.b	$\hat{L}^b$	12	12	12	12	12	14	76	68	44	36	30	30
	Δ^b	11	12	12	12	12	13	13	13	13	14	14	15
sim.n	$\hat{L}^n$	12	12	12	12	12	14	32	33	35	39	51	cv
	Δ^n	11	12	12	12	12	15	32	32	33	40	∞	
sim.b	$\hat{L}^b$	12	12	12	12	12	14	42	42	42	29	29	28
	Δ^b	11	12	12	12	12	13	13	13	13	14	14	15

Table 11.1: Latencies $\hat{L}$ in μs and total buffer capacities Δ in numbers of locations determined for Figure 11.6 (Legend: p = period-and-jitter, e = execution intervals, c = consideration of cyclic dependencies / precedence constraints to limit interference, i = iterative buffer sizing, b = blocking write buffers, n = non-blocking write buffers, s = synchronization edges, sim = simulation, cv = constraint violation).

cyclic data dependencies (case p.c) (up to 50% larger throughput guarantees and up to 52% smaller end-to-end latencies). Likewise, the approaches with execution intervals and iterative buffer sizing (cases e.c.i.n and e.c.i.b) give us a consistently higher analysis accuracy than the respective approaches with period-and-jitter and iterative buffer sizing (cases p.c.i.n and p.c.i.b) (up to 41% larger throughput guarantees and up to 40% smaller end-to-end latencies).

This shows that the advantage of execution intervals over period-and-jitter persists even on combination with other techniques such as considering precedence constraints and an iterative buffer sizing. This is especially true for the tasks t_{DEMAP} , t_{DEINT} and t_{VIT} on the feedback loop, whose finish times are severely overestimated in the cases without execution intervals.

For completeness we also consider the addition of synchronization edges. However, for this particular example, the synchronization edges do not give us any advantage compared to the cases with iterative buffer sizing only. This is due to the fact that the tasks on the same processor are already rather tightly connected via FIFO buffers. For instance, the tasks t_{FFT} and t_{EQ} are directly exchanging data, such that the respective synchronization edges do not give any additional means to limit interference between these tasks. In addition, the execution intervals impose a tighter bound on interference than period-and-jitter, which implies that the room for improvement with other methods such as precedence constraints, iterative buffer sizing and also synchronization edges becomes smaller.

The simulation results obtained with the HAPI simulator described in Chapter 13 (cases sim.n and sim.b) are equal to our analysis results (cases e.c.i.n and e.c.i.b, respectively) if the source jitter is zero. This does not only confirm that our results are temporally conservative, but also that our analysis results are indeed very accurate in this case.

For the scenario with the additional source jitter we can conclude that here the iterative buffer sizing is especially helpful, as it effectively reduces interference between the tasks t_{FFT} and t_{EQ} by limiting the buffer capacity between the two. In comparison with simulation (cases sim.n and sim.b), we find that our results are temporally conservative. The deviation between our best analysis results (cases e.c.i.n and e.c.i.b) and simulation results can be explained by the fact that our analysis uses periodic bounds and hence assumes that bursts of input values are processed periodically, with the period of the source, whereas actually a faster processing may be possible, as already highlighted in Section 11.6.1.

Also note that for the simulation results assuming buffers with blocking writes (case sim.b) we have assumed the buffer capacities obtained via our iterative buffer sizing in the execution interval approach (case e.c.i.b), whereas for the simulation with non-blocking write buffers (case sim.n) we have assumed the buffer capacities to be equal to the ones obtained via execution intervals and iterative buffer sizing for the same buffer type (case e.c.i.n), where applicable, and otherwise unconstrained. That in the scenario with the jittered source simulation results are almost always better if limited buffer capacities are used can be only explained by the fact that the constrained blocking write buffers obtained via

iterative buffer sizing do not only influence analysis results, but effectively shape the schedules of tasks, which enables both higher throughputs (up to 11% higher) and lower end-to-end latencies (up to 43% lower).

We can also conclude that all improvements, i.e. execution intervals, consideration of precedence constraints, iterative buffer sizing and synchronization edges, have a positive influence on required buffer capacities, and that for both buffers with blocking and non-blocking writes. For instance, our best approach for non-blocking write buffers (case e.c.i.n) requires only 11 buffer locations, whereas period-and-jitter only (case p) requires 17 for the source period $P_{\text{SRC}} = 17\mu\text{s}$, which is a reduction in space requirements of 35%.

Figure 11.7: Processor sharing between control and data parts of WLAN 802.11p transceiver packet decoders.

Finally, consider the example depicted in Figure 11.7, which represents the task graphs of both control and data parts of the transceiver decoding modes. Most of the tasks are still executed in isolation, except for the t_{DEMAP} tasks, whose mappings are swapped between the control and data parts.

The analysis results for this example are presented in Table 11.2, again for the control part only (results for the data part are identical since the mappings of the t_{DEMAP} tasks are swapped symmetrically). At first, you can see that although the modification compared to Figure 11.6 is minimal, the results are much worse compared to the case that the task graphs are executed in isolation: For the scenario without source jitter the smallest source period for which we determine convergence is $11\mu\text{s}$, compared to the $8\mu\text{s}$ from before. If a source jitter is considered additionally, we do not even achieve convergence for source periods below $16\mu\text{s}$. Similarly, for the same source periods, consistently larger end-to-end latencies are determined and consistently larger buffer capacities required.

These differences are not merely caused by analysis inaccuracies, but by the circumstance that the t_{DEMAP} tasks become uncorrelated with the other tasks on

$P_{\text{SRC}} / \mu\text{s}$		17	14	13	12	11	32	30	24	22	20	18	16
$J_{\text{SRC}} / \mu\text{s}$					0		64	60	48	44	40	36	32
p	$\hat{L}$	34	cv	cv	cv	cv	129	cv	cv	cv	cv	cv	cv
	Δ^n	17					39						
	Δ^b	13					17						
p.c	$\hat{L}$	29	29	cv	cv	cv	109	105	97	cv	cv	cv	cv
	Δ^n	16	18				38	38	39				
	Δ^b	12	14				14	15	17				
p.c.i.n	$\hat{L}^n$	20	25	25	25	cv	105	101	89	89	cv	cv	cv
	Δ^n	12	17	17	18		36	37	38	39			
p.c.i.s.n	$\hat{L}^n$	16	16	25	25	cv	105	101	89	89	cv	cv	cv
	Δ^n	12	12	17	18		36	37	38	39			
p.c.i.b	$\hat{L}^b$	20	24	24	24	cv	88	84	80	76	72	72	cv
	Δ^b	12	12	13	13		13	13	14	14	14	17	
p.c.i.s.b	$\hat{L}^b$	16	16	16	16	16	88	84	76	72	68	64	64
	Δ^b	12	12	12	12	12	13	13	14	14	14	15	17
e	$\hat{L}$	25	25	25	cv	cv	118	120	cv	cv	cv	cv	cv
	Δ^n	14	17	17			37	39					
	Δ^b	12	12	13			14	17					
e.c	$\hat{L}$	16	16	16	20	20	100	96	84	84	80	cv	cv
	Δ^n	12	12	12	16	18	33	35	38	39	39		
	Δ^b	12	12	12	12	13	14	14	15	15	17		
e.c.i.n	$\hat{L}^n$	16	16	16	20	20	100	96	84	84	80	cv	cv
	Δ^n	12	12	12	16	18	33	35	38	39	39		
e.c.i.s.n	$\hat{L}^n$	16	16	16	20	20	100	96	84	84	80	cv	cv
	Δ^n	12	12	12	16	18	33	35	38	39	39		
e.c.i.b	$\hat{L}^b$	16	16	16	20	20	88	84	76	72	68	68	64
	Δ^b	12	12	12	12	13	13	13	14	14	14	16	17
e.c.i.s.b	$\hat{L}^b$	16	16	16	20	20	88	84	76	72	68	68	64
	Δ^b	12	12	12	12	13	13	13	14	14	14	16	17
sim.n	$\hat{L}^n$	16	16	16	16	16	33	31	33	33	34	35	35
	Δ^n	12	12	12	16	18	33	35	38	39	39	∞	∞
sim.b	$\hat{L}^b$	16	16	16	16	16	47	47	33	32	34	35	35
	Δ^b	12	12	12	12	12	13	13	14	14	14	16	17

Table 11.2: Latencies $\hat{L}$ in μs and total buffer capacities Δ in numbers of locations determined for Figure 11.7 (Legend: p = period-and-jitter, e = execution intervals, c = consideration of cyclic dependencies / precedence constraints to limit interference, i = iterative buffer sizing, b = blocking write buffers, n = non-blocking write buffers, s = synchronization edges, sim = simulation, cv = constraint violation).

their processors. This means that the t_{DEMAP} tasks actually can experience more interference than in Figure 11.6. This observation is supported by comparing the simulation results between Tables 11.1 and 11.2.

Finally, the results for execution intervals are still better than the results for period-and-jitter, although the gap between the results is smaller than in the single task graph case. This is due to the fact that for uncorrelated tasks the period-and-jitter characterization is very similar to the characterization described by $\psi_{b \rightarrow a}^*(\hat{w}_a^\circ(q), q)$. As a matter of fact, in some cases period-and-jitter can be even more accurate for interference over multiple task graphs, as can be seen by comparing the results for the cases p.c.i.s.b and e.c.i.s.b in Table 11.2. This, as well as the conclusion from the following section that run-times of interference computations are small compared to the overall algorithmic run-times, suggests that a combination of period-and-jitter and execution intervals can lead to an even better analysis accuracy if an application consists of multiple task graphs.

11.6.3 ANALYSIS RUN-TIMES

The presented analysis algorithms were executed on a PC with an Intel® Core™ i7 processor. All measured run-times are in the microsecond range, which shows that our approaches are indeed capable of a quick temporal analysis for cyclic real-time streaming applications. Although the measured run-times are neglectably small, we can still make some observations about their influencing factors. The invocations of the algorithms p.c and e.c for the graph in Figure 11.6 all took around 65μs, independent of source frequencies and jitters. This insinuates that the interference computations only take marginal portions of the run-times, since no differences can be observed between period-and-jitter and the computationally more complex execution interval characterization. The combinations with an iterative buffer sizing p.c.i and p.e.i took around 25% longer (around 80μs in all cases), which can be explained by the additional, repetitive invocations of the Floyd-Warshall algorithm.

For the example in Figure 11.7 the run-times are in all cases less than 100% larger compared to the case in Figure 11.6 (around 110μs without and around 160μs with iterative buffer sizing), although the number of analyzed tasks is doubled. This suggests that the run-times scale well in numbers of tasks, allowing for the quick analysis of even larger applications than considered in this case study.

Task Phase Analysis

In this chapter we present our analysis techniques for multi-phase applications. According to Section 2.2, the techniques have the same effectiveness as the corresponding techniques from the previous two chapters, but are capable of analyzing the more entropic multi-phase analysis models. We show that this results in both an increased applicability and an increased analysis accuracy.

The remainder of this chapter is structured as follows: In Section 12.1 we present the analysis flow of our multi-phase analysis. Thereafter, we discuss the different steps of this flow, namely the expansion of CSDF graphs to HSDF graphs in Section 12.2, the determination of periodic bounds on the schedules of task phases in Section 12.3 and the computation of maximum phase response times in Section 12.4. We consider interference characterizations based on both period-and-jitter and on execution intervals. Section 12.5 contains proofs on the conservativeness of the resulting approaches. In Section 12.6 we further present an extension of the task phase analysis with execution intervals, such that not only multi-rate and multi-phase applications can be accurately analyzed, but also applications with multiple shared resources, i.e. combinations of e.g. shared processors, memories, buses or peripherals. In Section 12.7 we evaluate the performance of our multi-phase analysis in a case study.

12.1 ANALYSIS FLOW

Figure 12.1 depicts the flow of our analysis approaches that can be used to give guarantees on the temporal behavior of multi-phase applications. Input to this analysis flow are the multi-phase analysis models that are derived from a multi-phase application model according to Chapter 9. In step 1 the CSDF models are expanded to equivalent HSDF models according to the algorithm described in [BELP96], which enables application of the data dependency analysis techniques introduced in the previous chapters.

Figure 12.1: Overview of the analysis flow.

Step 2 computes two periodic schedules, one defined by the start times of actors in the best-case analysis model and the other by the start times of actors in the worst-case analysis model. The firing durations in the best-case model are set to BCETs, while the firing durations in the worst-case model are initialized to WCETs and in later iterations set to the maximum response times computed in step 3. Lower and upper bounds on the (external) enabling and finish times of task phases are derived from the periodic schedules, which are inserted as input intervals into the response time analysis model.

Using the input intervals, either upper bounds on the enabling jitters of task phases or execution intervals are determined in step 3, depending on whether period-and-jitter or execution intervals are used to characterize interference. In addition, we also consider cyclic data dependencies or precedence constraints obtained from the precedence constraint analysis model to determine another bound on interference than only based on period-and-jitter or execution intervals, respectively. The resulting maximum response times of task phases are normalized such that they can be used as maximum firing durations in the worst-case analysis model considered in the next analysis flow iteration.

Note that the maximum response times computed in step 3 depend on the schedules computed in step 2, which themselves depend on the maximum response times computed in step 3. This mutual dependency is the reason for the iterative character of the analysis flow. Hence we need to check in step 4 whether all maximum response times have converged, i.e. did not change since the last iteration of the analysis flow, or whether any temporal constraints are violated. If none of this is the case, the iterative loop is repeated, starting again with step 2, until either convergence or constraint violation is achieved.

All steps of the analysis flow are monotone, i.e. their results cannot decrease throughout increasing iterations of the flow, which is a sufficient requirement to prove flow termination. The proofs of termination are analogous to the ones for single-phase applications presented in Appendices B.1 and B.2.

12.2 EXPANDING CSDF GRAPHS TO HSDF GRAPHS

In this section we discuss the HSDF expansion of CSDF graphs that we use to make the analysis techniques presented in the previous two chapters applicable for multi-phase applications. As a detailed discussion of this expansion, which is e.g. presented in [BELP96], is beyond the scope of this thesis, we confine ourselves to the presentation of an intuitive example.

Figure 12.2: HSDF expansions of precedence constraint, best-case and worst-case CSDF models as used in our analysis flow (with $\langle 0, 1^3 \rangle$ a shorthand notation for $\langle 0, 1, 1, 1 \rangle$ and $\langle 1, 0 \rangle^2$ for $\langle 1, 0, 1, 0 \rangle$).

Consider the CSDF graph in the middle, upper half of Figure 12.2. This graph consists of two CSDF actors v_a and v_b that communicate via two edges, one in the forward and one in the backward direction. As such, the graph could correspond to (a part of) an precedence constraint analysis model as described in Section 9.3.3, with the two actors including two tasks t_a and t_b from the response time analysis model and the two edges corresponding to a FIFO buffer with zero initially full and two initially empty locations. Actor v_a has two phases, with the first phase consuming one and producing no token and the second consuming also one and producing two tokens per firing. Actor v_b has two phases as well, with the first phase consuming and producing one token per firing and the second neither consuming nor producing tokens. The firing durations of the actors are defined per phase and can further vary in each iteration. Finally, both actors have a self-edge with one token and consumption and production rates of one

on these self-edges, which ensures that a phase cannot fire before the previous one has finished.

Let us now try to expand this CSDF graph to an HSDF graph that has exactly the same behavior. In a first step we create one HSDF actor per CSDF phase, which would be two phases for actor v_a and two for actor v_b . However, we additionally need to consider that a rate conversion occurs between the two actors: Per firing of all phases of actor v_a two tokens are consumed and two produced, whereas per firing of all phases of actor v_b only one token is consumed and produced. This means that one firing of actor v_a enables and is enabled by two firings of actor v_b . As HSDF is single-rate and hence does not support rate conversions inherently, we need to replicate actor v_b twice, once for each of the two enablings by a single firing of actor v_a . In total, this results in two HSDF actors $v_{a,0}$ and $v_{a,1}$ for CSDF actor v_a and four HSDF actors $v_{b,0}$, $v_{b,1}$, $v_{b,2}$ and $v_{b,3}$ for CSDF actor v_b , as depicted in the middle, lower half of Figure 12.2.

The firing durations of the HSDF actors are set to the firing durations of the corresponding phases. For the replicated actor v_b this means that the first replication in the HSDF expansion is assigned all firing durations of CSDF actor v_b in odd iterations and the second replication all firing durations in even iterations.

Next we map the edges of the CSDF model to the HSDF expansion. We simply take into account which phases communicate tokens to which phases and draw the edges in the expansion accordingly. The second phase of v_a produces two tokens, of which one is consumed by the odd and the other by the even iterations of the first phases of actor v_b . Consequently we draw one edge from phase $v_{a,0}$ to phase $v_{b,0}$ and another from $v_{a,0}$ to $v_{b,2}$. To draw the backward edges from the phases of actor v_b to the phases of actor v_a we additionally need to consider the distribution of initial tokens. There are two initial tokens, of which the first enables the first phase of actor v_a and the second the second phase of v_a . Consequently, the first finished firing of the first replication of the first phase of actor v_b enables the first phase of actor v_a again, the second replication of the first phase of v_b the second phase of v_a , and so on. Thus we draw an edge from $v_{b,0}$ to $v_{a,0}$ and another from $v_{b,2}$ to $v_{a,1}$. Finally, we distribute the initial tokens on these edges, according to the phases they enable, such that one token is placed on the edge from $v_{b,0}$ to $v_{a,0}$, for the first enabling of the first phase of v_a , and one token on the edge from $v_{b,2}$ to $v_{a,1}$, for the first enabling of the second phase of v_a . This completes the HSDF expansion of the middle CSDF graph to the HSDF graph. It can be seen that both graphs have exactly the same behavior, as required.

Note that it can also occur that the source actors v_s are replicated r_s times in the HSDF expansion. In this case it follows that the source period in the HSDF model is not P_s , but actually $r_s \cdot P_s$. In the following we always assume such multiplied source periods, even though we just use P_s for brevity.

Now let us also consider the best-case and worst-case HSDF models that are depicted left and right to the precedence constraint HSDF model in Figure 12.2. As discussed in Section 9.3.2, the self-edges with one token are removed from both

best-case and worst-case analysis models, whereas all other self-edges between the phases are preserved to keep the internal order between phases. Moreover, in the best-case model also all other edges with initial tokens are removed, which are for our example the edges leading from actor v_b back to v_a in the precedence constraint analysis model.

We can also contract the best-case and worst-case HSDF models back to CSDF models, as indicated in the upper half of Figure 12.2. Note that the actors $\check{v}_b$ and $\hat{v}_b$ now have four instead of two CSDF phases, which is required due to the self-edge removal that is conducted only between the second phase of the second replication and the first phase of the first replication, but not between the second phase of the first replication and the first phase of the second replication - representing this removal would not be possible with a CSDF actor of only two phases.

Also note that such HSDF expansions and contractions are always possible, i.e. we can expand any CSDF graph to an equivalent HSDF graph and back. In the following we use the HSDF expansions of the precedence constraint analysis models to analyze data dependencies between the corresponding tasks for a limitation of interference, while we employ the HSDF expansions of the best-case and worst-case analysis models to compute lower and upper bounds on the schedules of tasks.

Before we continue with the computation of such bounds on schedules in the next section, we must stress that a different indexing pattern than in Chapter 9 is used in this chapter. The differences between the new indexing pattern and the old one are as follows: First, the index i in Chapter 9 refers to the iterations of actor phases, i.e. each phase firing increases the index i by one. In this chapter we use the index i to indicate iterations of replicated actor firings instead, that is, the index i increases only after the firing of the last phase of the last replication of an actor. Second, we add another index x to all minimum and maximum firing durations, external enabling, total enabling and finish times, as well as enabling jitters to indicate the (replicated) phase to which the respective parameter refers to, with $0 \leq x < \Theta^r$.

For example, with i^{old} the index i used in Chapter 9 and i^{new} the index i used in this chapter we obtain for a maximum enabling time of a phase x of a task t_a with Θ^r replicated phases (i.e. phases in the HSDF expansions):

$$\forall_{i^{old}}: i^{new} = \left\lfloor \frac{i^{old}}{\Theta^r} \right\rfloor, x = i^{old} \% \Theta^r \Rightarrow \hat{e}_{a,x}(i^{new}) = \hat{e}_a(i^{old})$$

Finally, we do not differ between numbers of phases before expansion Θ and numbers of replicated phases $\Theta^r = r \cdot \Theta$ in the remainder of this chapter, but always mean the replicated number of phases whenever we speak of phases.

12.3 DETERMINING BOUNDS ON TASK PHASE SCHEDULES

In this section we present our method to compute bounds on the schedules of task phases in step 2 of the analysis flow. In the previous chapters we have discussed

that temporally conservative bounds on the schedules of tasks in the response time analysis model can be derived from the periodic start times of actors in the best-case and worst-case analysis models according to Section 9.3.2. In this section we apply the same concept to bound schedules of task phases.

First, note that there is a fundamental difference between the best-case and worst-case models for single-phase applications used in the previous chapters and the ones for multi-phase applications that we make use of here: In the former only edges between different actors are considered, but no self-edges. For such models the LPs used to compute start times determine lower and upper bounds on the external enabling times of tasks. For multi-phase applications, however, we only remove the self-edges between the last and the first phases of actors, but the other phases remain connected via self-edges to model internal enabling, as shown in Figure 12.2. This implies that if the same LPs are applied on such models, then the resulting start times for the first phases are bounds on the external enabling times of the corresponding task phases and all other start times bounds on total enabling times, i.e. bounds on the maxima or minima of external and internal enabling times. As we require bounds on the external enabling times of all task phases and not only of the first ones, we consequently modify the LPs from the previous chapters such that they can also compute so-called external start times of actors, i.e. start times without consideration of self-edges.

For the computation of lower bounds on the schedules of task phases we make use of the HSDF expansion of the best-case analysis model that is exemplified in Figure 12.2. The start times, as well as external start times of actor phases in such a best-case HSDF graph can be determined with the following LP (with $\check{V}$ the set of actors in the best-case analysis model and $\check{E}$ the respective set of edges, which contains all data dependencies from the application model that do not contain initial tokens, with $\check{E}^{ext} \subseteq \check{E}$ the set of all edges in the best-case model that are no self-edges and with the firing durations $\check{\rho}_{a,x}$ of actor phases $\check{v}_{a,x}$ assigned to the BCETs of the corresponding task phases):

$$\begin{aligned} \text{Minimize: } & \sum_{\check{v}_{a,x} \in \check{V}} \check{s}_{a,x}^{ext} + \check{s}_{a,x} \\ \text{Subject to: } & \check{s}_{s,0} = 0, \quad \forall_{e_{a,x,b,y} \in \check{E}^{ext}}: \quad \check{s}_{b,y}^{ext} - \check{s}_{a,x} \geq \check{\rho}_{a,x}, \\ & \forall_{e_{a,x,b,y} \in \check{E}}: \quad \check{s}_{b,y} - \check{s}_{a,x} \geq \check{\rho}_{a,x} \end{aligned}$$

The total start times $\check{s}_{a,x}$ define a periodic schedule of the actor phases in the best-case analysis model, i.e. it holds that the start time of an actor phase $\check{v}_{a,x}$ in firing i is equal to $\check{s}_{a,x} + i \cdot P_a$. By setting the start times of the first source actor phases $v_{s,0}$ to $\check{s}_{s,0} = 0$ it holds that all start times are computed relative to the first enabling of the respective first source actor phases. As we also define the enabling times of task phases in the response time analysis model relative to the first executions of their strictly periodic sources it follows that we can use the start times $\check{s}_{a,x}$ of actor phases $v_{a,x}$ in the best-case model to determine periodic

lower bounds on the enabling times $\varepsilon_{a,x}(i)$ of task phases $t_{a,x}$, i.e.:

$$\forall i \geq 0: \check{\varepsilon}_{a,x}(i) = \check{s}_{a,x} + i \cdot P_a \leq \varepsilon_{a,x}(i)$$

Note that such lower bounds also holds for the first phases that do not have incoming self-edges, as also these phases cannot be enabled earlier than at their respective external enabling times. Using the external start times of actors we can further define periodic lower bounds on the external enabling times $\varepsilon_{a,x}^{ext}(i)$ of task phases $t_{a,x}$, such that:

$$\forall i \geq 0: \check{\varepsilon}_{a,x}^{ext}(i) = \check{s}_{a,x}^{ext} + i \cdot P_a \leq \varepsilon_{a,x}^{ext}(i)$$

To compute upper bounds on the enabling times of task phases in the response time analysis model we make use of the HSDF expansion of the worst-case analysis model that is also exemplified in Figure 12.2. In the first iteration of the analysis flow the firing durations of actor phases are set to the WCETs of the task phases, whereas in subsequent iterations the maximum response times derived in step 3 of the analysis flow are used. The maximum response times of the first phases are defined relative to the external enabling times of these phases, whereas all other maximum response times are defined relative to total enabling times, matching the structure of the worst-case analysis model. We can use the following LP to determine start times and external start times of actors in the worst-case model:

$$\begin{aligned} \text{Minimize: } & \sum_{\hat{v}_{a,x} \in \hat{V}} \hat{s}_{a,x}^{ext} + \hat{s}_{a,x} \\ \text{Subject to: } & \hat{s}_{s,0} = 0, \quad \forall_{e_{a,x,b,y} \in \hat{E}^{ext}}: \quad \hat{s}_{b,y}^{ext} - \hat{s}_{a,x} \geq \hat{\rho}_{a,x} - \delta_{a,x,b,y} \cdot P_b, \\ & \forall_{e_{a,x,b,y} \in \hat{E}}: \quad \hat{s}_{b,y} - \hat{s}_{a,x} \geq \hat{\rho}_{a,x} - \delta_{a,x,b,y} \cdot P_b \end{aligned}$$

Note that for the first phases of an actor it holds that $\hat{s}_{a,0} = \hat{s}_{a,0}^{ext}$. With the results of the LP we can bound both external and total enabling times of task phases $t_{a,x}$ in the response time analysis model from above as follows:

$$\forall i \geq 0: \varepsilon_{a,x}^{ext}(i) \leq \hat{\varepsilon}_{a,x}^{ext}(i) = \hat{s}_{a,x}^{ext} + i \cdot P_a \wedge x \neq 0 \Rightarrow \varepsilon_{a,x}(i) \leq \hat{\varepsilon}_{a,x}(i) = \hat{s}_{a,x} + i \cdot P_a$$

With the bounds on (external) enabling times and the respective firing durations we can further also bound the finish times $f_{a,x}(i)$ of task phases $t_{a,x}$:

$$\forall i \geq 0: \check{f}_{a,x}(i) = \check{s}_{a,x} + \check{\rho}_{a,x} + i \cdot P_a \leq f_{a,x}(i) \leq \hat{f}_{a,x}(i) = \hat{s}_{a,x} + \hat{\rho}_a + i \cdot P_a$$

What is still missing is an upper bound on the total enabling times of the first phases $\hat{v}_{a,0}$ which can be obtained by additionally considering delays from the last phases v_{a,Θ_a^r-1} , i.e.:

$$\begin{aligned} \forall i \geq 0: \varepsilon_{a,0}(i) & \leq \hat{\varepsilon}_{a,0}(i) = \max(\hat{\varepsilon}_{a,0}^{ext}(i), \hat{f}_{a,\Theta_a^r-1}(i-1)) \\ & = \max(\hat{s}_{a,0}, \hat{s}_{a,\Theta_a^r-1} + \hat{\rho}_{a,\Theta_a^r-1} - P_a) + i \cdot P_a \end{aligned}$$

Given that the firing durations used in the best-case and worst-case analysis models adhere to the multi-phase firing duration criterion in Equation 9.8 it holds that both best-case and worst-case models are valid bounding abstractions of the response time analysis model and as such we can use all bounds from this section to define the input intervals of tasks in the response time analysis model.

12.4 COMPUTING MAXIMUM RESPONSE TIMES

In this section we discuss our methods to compute upper bounds on the response times of task phases in the response time analysis model. The basic idea of the methods is presented in Section 12.4.1. In Section 12.4.2 the period-and-jitter technique to compute upper bounds on response times of entire tasks is recapped and brought to an algorithmic form that allows for an easier extension to a technique for task phases. Thereafter, we present our algorithm to compute maximum response times of task phases using period-and-jitter in Section 12.4.3, whose stop criterion is proven in Appendix B.3. The method to limit interference with cyclic data dependencies between task phases is detailed in Section 12.4.4. Section 12.4.5 finally presents an algorithm to compute maximum response times of task phases that uses execution intervals instead of period-and-jitter to bound interference.

12.4.1 BASIC IDEA

Figure 12.3: Basic idea (with $P_a = P_b \geq \hat{\chi}_a + 2 \cdot \hat{\chi}_b$, $\hat{J}_b = 0$, $\hat{\chi}_a = \hat{\chi}_{a,0} + \hat{\chi}_{a,1}$).

Consider the task graph depicted on the left side of Figure 12.3. We assume that both tasks are mapped to the same processor and that task t_b has a higher priority than task t_a . The WCETs of the tasks are denoted next to them. We apply the period-and-jitter analysis technique described in Chapter 10 to compute an upper bound on the finish time of task t_a . From the properties given in the caption of the figure it follows that we only have to consider $\hat{w}_a(0)$ and that higher priority task t_b can only interfere once with lower priority task t_a , resulting in the maximum finish time denoted below the graph.

Let us now split the task t_a into two phases, as depicted on the right side of the figure. As the analysis technique from Chapter 10 does not have a notion

of phases it treats each phase as a separate task. This results in a maximum finish time of $\hat{f}_{a,0} = \hat{\varepsilon}_a^{ext} + \hat{\chi}_{a,0} + \hat{\chi}_b$ for $t_{a,0}$, which is at the same time the maximum external enabling time of task phase $t_{a,1}$. As the phases are treated as separate tasks also the maximum response times are computed separately, such that $\hat{f}_{a,1} = \hat{f}_{a,0} + \hat{\chi}_{a,1} + \hat{\chi}_b$, resulting in the upper maximum finish time in the figure. Due to the separate computation of maximum response times one can see that the interference of task t_b is considered twice and thus over-approximated.

To resolve this problem we propose to compute maximum busy periods over multiple phases. For phase $t_{a,0}$ we compute $\hat{w}_{a,0 \rightarrow a,0} = \hat{\chi}_{a,0} + \hat{\chi}_b$. For phase $t_{a,1}$ we then extend this maximum busy period to $\hat{w}_{a,0 \rightarrow a,1}$ by $\hat{\chi}_{a,1}$ and any interference that can occur during this extension, i.e. interference that can occur during the whole maximum busy period over both $t_{a,0}$ and $t_{a,1}$ minus the interference already considered for $t_{a,0}$. This is allowed as the phases $t_{a,0}$ and $t_{a,1}$ are always in consecutive execution, which implies that if an interference of $\hat{\chi}_b$ occurred during the execution of $t_{a,1}$ the internal enabling time of $t_{a,1}$ would be at the same time reduced by $\hat{\chi}_b$. As no additional interference can occur in the extension we derive the lower maximum finish time in the figure, correctly considering only one interference of task t_b .

Note that if phase $t_{a,1}$ were additionally externally enabled by another task t_c we would have to consider a second maximum busy period $\hat{w}_{a,1 \rightarrow a,1}$ starting at external enabling time $\hat{\varepsilon}_{a,1}^{ext}$. For $\hat{w}_{a,1 \rightarrow a,1}$ we would not be allowed to exclude interference considered in $\hat{w}_{a,0 \rightarrow a,0}$ as $\hat{\varepsilon}_{a,1}^{ext}$ does not get smaller if the interference of t_b is assumed to occur during $t_{a,1}$ instead of $t_{a,0}$. This would result in $\hat{w}_{a,1 \rightarrow a,1} = \hat{\chi}_{a,1} + \hat{\chi}_b$ and a finish time of phase $t_{a,1}$ being equal to $\hat{f}_{a,1} = \max(\hat{\varepsilon}_{a,0}^{ext} + \hat{w}_{a,0 \rightarrow a,1}, \hat{\varepsilon}_{a,1}^{ext} + \hat{w}_{a,1 \rightarrow a,1})$. Moreover, for smaller periods we would also need to consider self-delay over phases from different task iterations, which can be achieved by extending both maximum busy periods $\hat{w}_{a,0 \rightarrow a,1}$ and $\hat{w}_{a,1 \rightarrow a,1}$ over additional executions of $t_{a,0}$ and $t_{a,1}$.

The derivation of algorithms that are capable of computing maximum response times using such maximum busy periods over multiple phases is subject to the next sections.

12.4.2 MAXIMUM RESPONSE TIMES OF TASKS BASED ON PERIOD-AND-JITTER

Before we introduce our first algorithm to compute maximum response times of task phases let us first recap the equations from Chapter 10 that are used to derive maximum response times $\hat{\rho}_a$ of entire tasks (or, in other words, tasks that consist of only one phase) based on period-and-jitter. A necessary requirement to derive a maximum response time of a task t_a in the response time analysis model is the existence of a periodic upper bound on the external enabling times $\varepsilon_a^{ext}(i)$ of all executions of that task, i.e.:

$$\forall_{i \geq 0}: \varepsilon_a^{ext}(i) \leq \hat{\varepsilon}_a^{ext}(i) = \hat{s}_a^{ext} + i \cdot P_a$$

Given such a bound we have discussed that a periodic upper bound on the finish times $f_a(i)$ of a task t_a can be determined by computing maximum response times $\hat{R}_a$ as follows:

$$\forall i \geq 0: f_a(i) \leq \hat{f}_a(i) = \hat{\varepsilon}_a^{ext}(i) + \hat{R}_a = \hat{\varepsilon}_a^{ext}(i) + \max_{q \geq 0} (\hat{w}_a(q) - q \cdot P_a) \quad (12.1)$$

with

$$\begin{aligned} \hat{w}'_a(q) &= (q + 1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \eta_b(\hat{w}'_a(q)) \cdot \hat{\chi}_b \\ \hat{w}_a(q) &= (q + 1) \cdot \hat{\chi}_a + \sum_{t_b \in hp_a} \zeta_{b \rightarrow a}(\hat{w}'_a(q), q) \cdot \hat{\chi}_b \end{aligned}$$

and

$$\begin{aligned} \eta_b(\Delta\tau) &= \left\lceil \frac{\hat{J}_b + \Delta\tau}{P_b} \right\rceil, \quad \zeta_{b \rightarrow a}(q) = \delta(\mathcal{P}_{a,b}) + \delta(\mathcal{P}_{b,a}) + q - 1 \\ \gamma_{b \rightarrow a}(\Delta\tau, q) &= \min(\eta_b(\Delta\tau), \zeta_{b \rightarrow a}(q)) \end{aligned}$$

The computation makes use of maximum busy periods $\hat{w}'_a(q)$ and $\hat{w}_a(q)$ that are upper bounds on the total execution time of $q + 1$ executions of a task t_a , including not only $q + 1$ times the WCET $\hat{\chi}_a$ of task t_a , but also the sum of all possible interferences of higher priority tasks t_b with $t_b \in hp_a$ that can occur during these executions. The interference of a task t_b is bounded with its maximum number of enablings that can take place within $\hat{w}_a(q)$, multiplied by the corresponding WCET.

In a first step a maximum busy period $\hat{w}'_a(q)$ is determined based on the period-and-jitter characterization $\eta_b(\Delta\tau)$ only. This interference characterization gives the maximum number of executions of a task t_b in any time interval $\Delta\tau$, using the maximum enabling jitter $\hat{J}_b$ and the period P_b , with the maximum enabling jitter being defined by the input intervals prevalent in the response time analysis model. The maximum busy period $\hat{w}'_a(q)$ is computed iteratively until a fixed point is found, which is required due to the dependency of $\eta_b(\hat{w}'_a(q))$ on $\hat{w}'_a(q)$.

Afterwards the maximum busy period $\hat{w}'_a(q)$ is reduced to $\hat{w}_a(q)$ by computing interference not only based on period-and-jitter, but also based on cyclic data dependencies. The function $\zeta_{b \rightarrow a}(q)$ gives the maximum number of executions of a task t_b during q executions of a task t_a based on cyclic data dependencies between the two in the corresponding precedence constraint analysis model according to Section 9.3.3. For that means the minimum numbers of initial tokens on any path from (to) an actor v_a to (from) an actor v_b are determined, which are denoted as $\delta(\mathcal{P}_{a,b})$ and $\delta(\mathcal{P}_{b,a})$, respectively.

The reason for not only considering one execution of task t_a (i.e. $q = 0$), but multiple executions, is that the maximum finish time $\hat{f}_a(i)$ is defined relative to the maximum external enabling time $\hat{\varepsilon}_a^{ext}(i)$. Consequently any delays of previous executions must be included in $\hat{\rho}_a$, which is achieved by assuming that an

execution of task t_a can be in consecutive execution with any number q executions of its predecessors. To maximize such self-delay it is further assumed that the first of $q+1$ executions is enabled externally at its maximum external enabling time. Recall that the maximum external enabling time is periodic with P_a , which explains the term $-q \cdot P_a$ in Equation 12.1. Finally it holds according to [TBW94] that a $\hat{w}_a(q+1)$ only has to be considered if it holds that $\hat{w}'_a(q) > (q+1) \cdot P_a$.

To compute accurate maximum response times of task phases we translate the maximum finish time computation in Equation 12.1 into an algorithmic representation. It can be seen that the algorithm presented in Figure 12.4 produces the same results as Equation 12.1 if we redefine the interference characterizations for zero inputs, such that $\eta_a(0) = 0$ and $\gamma_{b \rightarrow a}(0, -1) = 0$. This is allowed as no interference has to be considered if there is nothing to interfere with.

```

1  $\hat{f}_a = 0;$ 
2  $q = 0; \hat{w}'_a = \hat{w}_a = 0;$ 
3 do {
4    $\hat{w}'_a^{\oplus} = \hat{w}_a^{\oplus} = 0;$ 
5    $\hat{w}'_a^{\oplus} = \hat{\chi}_a + \sum_{t_b \in hp_a} [\eta_b(\hat{w}'_a + \hat{w}'_a^{\oplus}) - \eta_b(\hat{w}'_a)] \cdot \hat{\chi}_b;$ 
6    $\hat{w}'_a^{\oplus} = \hat{\chi}_a + \sum_{t_b \in hp_a} [\gamma_{b \rightarrow a}(\hat{w}'_a + \hat{w}'_a^{\oplus}, q) - \gamma_{b \rightarrow a}(\hat{w}'_a, q-1)] \cdot \hat{\chi}_b;$ 
7    $\hat{w}'_a = \hat{w}'_a + \hat{w}'_a^{\oplus}; \hat{w}_a = \hat{w}_a + \hat{w}'_a^{\oplus};$ 
8    $\hat{f}_a = \max(\hat{f}_a, \hat{s}_a^{ext} + \hat{w}_a - q \cdot P_a);$ 
9    $q++;$ 
10 } while ( $\hat{w}'_a > q \cdot P_a$ );
```

Figure 12.4: Algorithm to compute upper bounds on finish times of tasks using period-and-jitter.

12.4.3 MAXIMUM RESPONSE TIMES OF TASK PHASES BASED ON PERIOD-AND-JITTER

In this section we describe the maximum response time computation for task phases based on period-and-jitter. We derive maximum response times from the input intervals in the maximum response time model that are determined according to Section 12.3 in step 2 of our analysis flow. More precisely, we require the upper bounds on maximum external enabling times of task phases $\hat{s}_{a,x}^{ext}$, as well as maximum external enabling jitters of phases $\hat{j}_{b,y}$ that can be determined by computing differences between minimum and maximum enabling times, i.e.:

$$\begin{aligned}
\forall_{i \geq 0}: J_{b,y}(i) &\leq \hat{j}_{b,y} = \hat{e}_{b,y}(i) - \check{e}_{b,y}(i) \\
&= \begin{cases} \max(\hat{s}_{b,0}, \hat{s}_{b,\Theta_b^r-1} + \hat{\rho}_{b,\Theta_b^r-1} - P_b) - \check{s}_{b,0} & y = 0 \\ \hat{s}_{b,y} - \check{s}_{b,y} & \text{otherwise} \end{cases}
\end{aligned}$$

Given maximum external enabling times and maximum enabling jitters of task phases we can now extend the finish time bound for entire tasks in Figure 12.4 to the finish time bound for task phases presented in Figure 12.5. The interference characterization $\eta_{b.y}(\Delta\tau)$ used in this algorithm is the same as $\eta_b(\Delta\tau)$ in Equation 12.1, except for the difference that $\hat{J}_b$ is replaced by a maximum enabling jitter per task phase $\hat{J}_{b.y}$. This is allowed as it does not matter whether the parameter $\Delta\tau$ represents a maximum busy period of a single task or of multiple task phases. The derivation of the interference characterization $\gamma_{b.y \rightarrow a}(\Delta\tau, \mathcal{Z})$ from both response time and precedence constraint analysis models is subject to the following Section 12.4.4.

The main difference to the algorithm for entire tasks is that maximum busy periods have to be computed over multiple task phases, as single task phases of tasks with more than one phase can never be in consecutive execution with themselves, they are always preceded and followed by other phases. For that matter we introduce the variable x that is used to traverse the different phases of the same task execution, whereas the variable q is used to distinguish between different executions of the same phases.

```

1   $\forall_{0 \leq x < \Theta_a^r} : \hat{f}_{a,x} = 0;$ 
2  for all  $(x' : e_{b,y,a,x'} \in E^{ext}) \{$ 
3     $x = x'; q = 0; \hat{w}'_{a,x'} = \hat{w}_{a,x'} = 0; \mathcal{Z}_{a,x'} = \emptyset;$ 
4    do {
5       $\hat{w}'_{a,x'} = \hat{w}_{a,x'}^{\oplus} = 0;$ 
6       $\hat{w}'_{a,x'} = \hat{\lambda}_{a,x} + \sum_{t_{b,y} \in hpa} [\eta_{b,y}(\hat{w}'_{a,x'} + \hat{w}'_{a,x'}^{\oplus}) - \eta_{b,y}(\hat{w}'_{a,x'})] \cdot \hat{\chi}_{b,y};$ 
7       $\hat{w}_{a,x'}^{\oplus} = \hat{\lambda}_{a,x} + \sum_{t_{b,y} \in hpa} [\gamma_{b,y \rightarrow a}(\hat{w}'_{a,x'} + \hat{w}_{a,x'}^{\oplus}, \mathcal{Z}_{a,x'} \cup \{(v_{a,x}, q)\}) - \gamma_{b,y \rightarrow a}(\hat{w}'_{a,x'}, \mathcal{Z}_{a,x'})] \cdot \hat{\chi}_{b,y};$ 
8       $\hat{w}'_{a,x'} = \hat{w}'_{a,x'} + \hat{w}_{a,x'}^{\oplus}; \hat{w}_{a,x'} = \hat{w}_{a,x'} + \hat{w}_{a,x'}^{\oplus};$ 
9       $\mathcal{Z}_{a,x'} = \mathcal{Z}_{a,x'} \cup \{(v_{a,x}, q)\};$ 
10      $\hat{f}_{a,x} = \max(\hat{f}_{a,x}, \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'} - q \cdot P_a);$ 
11      $x++;$ 
12     if  $(x = \Theta_a^r) \{$ 
13        $q++; x = 0;$ 
14     }
15   } while  $(x \neq x' \mid \mid \hat{w}'_{a,x'} > q \cdot P_a);$ 
16 }
```

Figure 12.5: Algorithm to compute upper bounds on finish times of task phases using period-and-jitter.

Another fundamental difference between considering separate task phases and entire tasks is that multiple task phases can be externally enabled, i.e. not all phases are necessarily in consecutive execution once the first phase is externally enabled. This means that a maximum busy period cannot only begin with the first phase, but with each phase accessing a FIFO buffer. This is captured by the outer for-loop in Figure 12.5 (line 2), which initializes a maximum busy period for all task phases corresponding to actor phases $v_{a,x'}$ with at least one input edge

in E^{ext} . This set is defined as the set of all edges E in the precedence constraint HSDF expansion according to Section 12.2, but without any self-edges.

Maximum busy periods are initialized under the assumption that the first considered execution $q = 0$ of task phase $t_{a.x} = t_{a.x'}$ is externally enabled at $\hat{s}_{a.x'}^{ext}$ and is not in consecutive execution with its predecessor. Therefore also no previous interference can be excluded, which we capture by defining $\eta_{b.y}(0) = 0$ and $\gamma_{b.y \rightarrow a}(0, \emptyset) = 0$.

After initialization the phases succeeding phase $t_{a.x'}$ in execution $q = 0$ are traversed by increasing x and q (lines 11 to 14). It is assumed that all these succeeding phases are in consecutive execution. This implies on the one hand that the maximum busy periods $\hat{w}'_{a.x'}$ and $\hat{w}_{a.x'}$ are not re-initialized for each additionally considered phase, but extended (line 8). On the other hand, also interference is not considered in separation for each phase, but computed over the entire, extended maximum busy periods and reduced by the interference already considered for the preceding phase executions (lines 6 and 7). The extensions of the maximum busy periods are populated to form the basis for the next iterations (line 8) and the set $\mathcal{Z}_{a.x'}$, which contains tuples representing the already considered corresponding actor firings in the precedence constraint analysis model, is extended analogously (line 9).

Finally the maximum finish times of all traversed task phases $t_{a.x}$ are recomputed for all considered task executions q (line 10), again based on the assumption that phase $t_{a.x'}$ in execution $q = 0$ is externally enabled at $\hat{s}_{a.x'}^{ext}$ and that all succeeding phases are in consecutive execution.

By construction the maximum finish times computed as $\hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'} - q \cdot P_a$ are conservative upper bounds on the finish times of a task phase $t_{a.x}$ if it is in consecutive execution with all its predecessors over $q+1$ executions starting with phase $t_{a.x'}$. Let us now assume that the stop criterion in line 15 were replaced by a simple while (true). Thereby we would consider all phases that can be externally enabled as starting points for maximum busy periods and traverse all phases x over any $q+1$ executions. This would ensure that all possible cases of consecutive execution would be considered for all phases, resulting in periodic upper bounds $\hat{f}_{a.x}$ that would hold for any execution. To be more precise, it would hold for a maximum busy period $\hat{w}_{a.x'}$ over $q+1$ executions that maximizes $\hat{f}_{a.x}$:

$$\forall i \geq 0: f_{a.x}(i) \leq \hat{s}_{a.x'}^{ext} + (i - q) \cdot P_a + \hat{w}_{a.x'} = \hat{f}_{a.x} + i \cdot P_a$$

Now that we have reasoned that the computed maximum finish times would be conservative upper bounds on the finish times of task phases in the response time analysis model if the stop criterion in line 15 was ignored, let us discuss the intuition behind the proof of the stop criterion, which is just analogous to the stop criterion used in Figure 12.4, is valid. A formal proof of the stop criterion can be found in Appendix B.3.

The algorithm terminates if it holds for $x = x'$ that $\hat{w}'_{a.x'} \leq q \cdot P_a$. From this follows for the update of the maximum finish time (line 10) in the next iteration:

$$\hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'} + \hat{w}_{a.x'}^{\oplus} - q \cdot P_a \leq \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^{\oplus}$$

Furthermore it can be seen that the extension $\hat{w}_{a,x'}^\oplus$ is always smaller than the $\hat{w}_{a,x'}$ computed on initialization. Therefore also the newly computed maximum finish time must be smaller than the one from initialization. As the same reasoning can be applied to all other extensions after the stop criterion is met it follows that the stop criterion must be valid as no maximum finish time computed after the stop criterion is met can be larger than the maximum finish time computed before.

This lets us conclude that the maximum finish times computed with our algorithm are indeed periodic upper bounds on the finish times $f_{a,x}(i)$ of the respective task phase executions in the response time analysis model, i.e. it holds:

$$\forall i \geq 0: f_{a,x}(i) \leq \hat{f}_{a,x}(i) = \hat{f}_{a,x} + i \cdot P_a$$

Based on these maximum finish times including interference we derive maximum response times of task phases that we use as maximum firing durations of the corresponding worst-case actor phases in Section 12.3. We have to differ between two cases. As can be seen in Figure 12.2 the edges from last actor phases to first actor phases are removed in the worst-case model. To maintain temporal conservativeness we have to include the delay from such removed self-edges, i.e. the self-delay on the first phases, in the firing durations of these phases. This is achieved by computing maximum response times for first phases as differences between maximum external enabling and maximum finish times, i.e.:

$$\hat{R}_{a,0} = \hat{f}_{a,0}(i) - \hat{e}_{a,0}^{ext}(i) = \hat{f}_{a,0} - \hat{s}_{a,0}^{ext}$$

The other actor phases in the worst-case model have self-edges coming from their predecessors. This implies that these phases are neither enabled before their maximum external enabling times nor before their maximum internal enabling times, i.e. the maximum finish times of their predecessors. Consequently we compute maximum response times for these phases as:

$$\forall 0 < x < \theta_a^r: \hat{R}_{a,x} = \hat{f}_{a,x}(i) - \max(\hat{e}_{a,x}^{ext}(i), \hat{f}_{a,x-1}(i)) = \hat{f}_{a,x} - \max(\hat{s}_{a,x}^{ext}, \hat{f}_{a,x-1})$$

12.4.4 LIMITING INTERFERENCE USING CYCLIC DATA DEPENDENCIES

In this section we first recap the derivation of the function $\zeta_{b \rightarrow a}(q)$ which we defined in Chapter 10 for entire tasks. Then we show that an application of this function for task phases results in a significant over-approximation and subsequently propose a more accurate solution $\zeta_{b,y \rightarrow a}(\mathcal{Z})$. Afterwards we simplify this solution by exploiting the dependencies between phases of a CSDF actor and finally combine $\zeta_{b,y \rightarrow a}(\mathcal{Z})$ with $\eta_{b,y}(\Delta\tau)$ to obtain $\gamma_{b,y \rightarrow a}(\Delta\tau, \mathcal{Z})$.

In the following we base all our observations on the dependencies between actor phases in the HSDF expansion of the precedence constraint analysis model that is exemplified in Figure 12.2. According to Section 9.3.3 all dependencies in the precedence constraint analysis model match the data dependencies in the

corresponding task graphs one-to-one. This allows us to draw conclusions about interference between task phases based on overlaps of actor phases.

We use the following definitions: Let $\mathcal{P}_{a.x,b.y}$ be the set of all directed paths of edges from an actor phase $v_{a.x}$ to an actor phase $v_{b.y}$. Then we define $\delta(\mathcal{P}_{a.x,b.y})$ as the minimum number of initial tokens on any path in $\mathcal{P}_{a.x,b.y}$, with $\delta(\mathcal{P}_{a.x,b.y}) = \infty$ if $\mathcal{P}_{a.x,b.y} = \emptyset$. According to Chapter 10 both $\delta(\mathcal{P}_{b.y,a.x})$ and $\delta(\mathcal{P}_{a.x,b.y})$ can be computed efficiently using the Floyd-Warshall algorithm.

In the same chapter we have defined so-called outgoing and incoming precedence sets containing all firings of an actor v_b that can occur before and after a firing i of an actor v_a , respectively, to derive $\zeta_{b \rightarrow a}(q)$. We redefine these sets for actor phases $v_{b.y}$ and $v_{a.x}$ instead of actors v_b and v_a as follows:

$$P_{v_{b.y} \rightarrow v_{a.x}(i)}^{\text{out}} = \{j \mid j < i + \delta(\mathcal{P}_{a.x,b.y})\}$$

$$P_{v_{b.y} \rightarrow v_{a.x}(i)}^{\text{in}} = \{j \mid j > i - \delta(\mathcal{P}_{b.y,a.x})\}$$

The intuition behind such precedence sets is as follows: Consider the HSDF graph depicted in Figure 12.6. From the semantics of HSDF graphs it can be concluded that phase $v_{b.0}$ can maximally fire $\delta(\mathcal{P}_{a.0,b.0}) = \delta_{a,b.0}^{\min}$ times before it must be enabled by an additional token produced by a finished firing of phase $v_{a.0}$. This implies that any firing $j \geq i + \delta(\mathcal{P}_{a.0,b.0})$ of phase $v_{b.0}$ cannot be enabled before firing i of phase $v_{a.0}$ has finished.

Figure 12.6: Limiting interference with cyclic data dependencies.

Likewise, phase $v_{a.0}$ can maximally fire $\delta(\mathcal{P}_{b.0,a.0}) = \delta_{b.0,a}^{\min} + 1$ times before it requires an additional token produced by a firing of phase $v_{b.0}$. This implies that any firing $j \leq i - \delta(\mathcal{P}_{b.0,a.0})$ of phase $v_{b.0}$ must have finished before firing j of phase $v_{a.0}$ is enabled. Negating these constraints then just gives all firings j of a phase $v_{b.0}$ that can occur during a firing i of a phase $v_{a.x}$, i.e. all firings in the union of both precedence sets.

The precedence set $P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}$ from Chapter 10 contains all iterations of an actor v_b not during one, but during $q + 1$ firings of an actor v_a and is computed by first drawing unions of the outgoing and incoming precedence sets of the firings $i \dots i + q$ of actor v_a and then taking the intersection of the resulting sets. By simply replacing actors with actor phases we could redefine

$P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}$ as:

$$P_{v_b.y \rightarrow \{v_{a.x}(i), \dots, v_{a.x}(i+q)\}} = \bigcup_{i'=i}^{i+q} P_{v_b.y \rightarrow v_{a.x}(i')}^{\text{out}} \cap \bigcup_{i'=i}^{i+q} P_{v_b.y \rightarrow v_{a.x}(i')}^{\text{in}}$$

By taking the number of elements in $P_{v_b \rightarrow \{v_a(i), \dots, v_a(i+q)\}}$ it was further possible to compute $\zeta_{b \rightarrow a}(q)$, an upper bound on the number of executions of a corresponding task t_b that can occur during $q + 1$ executions of a corresponding task t_a . Doing the same for $P_{v_b.y \rightarrow \{v_{a.x}(i), \dots, v_{a.x}(i+q)\}}$ results in an upper bound on the on the number of executions of a task phase $t_{b.y}$ that can occur during $q + 1$ executions of a phase $t_{a.x}$, such that:

$$\zeta_{b.y \rightarrow a.x}(q) = \left| P_{v_b.y \rightarrow \{v_{a.x}(i), \dots, v_{a.x}(i+q)\}} \right| = \delta(\mathcal{P}_{a.x, b.y}) + \delta(\mathcal{P}_{b.y, a.x}) + q - 1$$

Next, reconsider the example depicted in Figure 12.6. By applying $\zeta_{b.y \rightarrow a.x}(q)$ for a consecutive firing of phases $v_{a.0}$ and $v_{a.1}$ we obtain that $\zeta_{b.0 \rightarrow a.0, a.1}$ is equal to $|P_{v_b.0 \rightarrow v_{a.0}(i)}| + |P_{v_b.0 \rightarrow v_{a.1}(i)}|$. This results in the upper equation in the figure in which firings of phase $v_{b.y}$ that occur in both $P_{v_b.0 \rightarrow v_{a.0}(i)}$ and $P_{v_b.0 \rightarrow v_{a.1}(i)}$ are accounted for twice, leading to a significant over-approximation.

To prevent this we can redefine $\zeta_{b.0 \rightarrow a.0, a.1}$ as the number of elements in the intersection of the unions of outgoing and incoming phase precedence sets, i.e.:

$$\zeta_{b.0 \rightarrow a.0, a.1} = |(P_{v_b.y \rightarrow v_{a.0}(i)}^{\text{out}} \cup P_{v_b.y \rightarrow v_{a.1}(i)}^{\text{out}}) \cap (P_{v_b.y \rightarrow v_{a.0}(i)}^{\text{in}} \cup P_{v_b.y \rightarrow v_{a.1}(i)}^{\text{in}})|$$

This results in the lower equation in the figure.

In the following we generalize such interference characterizations for arbitrary sequences of phases.

We define tuples $z = (v_{a.x}, q)$ consisting of phases $v_{a.x}$ of an actor v_a and a firing index q . For instance, the tuple $(v_{a.1}, 0)$ would correspond to the first considered firing of phase $v_{a.1}$ and the tuple $(v_{a.0}, 1)$ to the second considered firing of phase $v_{a.0}$. An ordering relation on such tuples can be defined as follows: If it holds for two tuples $z = (v_{a.x}, q)$ and $z' = (v_{a.x'}, q')$ that $q < q'$ or if $q = q'$ that $x \leq x'$ then we say that $z \leq z'$. Given this ordering relation we define $\check{Z} = (v_{a.\check{x}}, \check{q})$ as the infimum and $\hat{Z} = (v_{a.\hat{x}}, \hat{q})$ as the supremum of a set Z . Furthermore we only consider sets Z containing complete sequences of such tuples, i.e. it must hold that:

$$z, z'' \in Z \wedge z < z' < z'' \Rightarrow z' \in Z$$

On such sets we can now also define precedence sets as the intersections of the unions of all outgoing and incoming precedence sets of the contained tuples, analogous to Section 10.3.2:

$$\begin{aligned} P_{v_b.y \rightarrow Z(i)} &= \bigcup_{z \in Z} P_{v_b.y \rightarrow v_{a.x}(i+q)}^{\text{out}} \cap \bigcup_{z \in Z} P_{v_b.y \rightarrow v_{a.x}(i+q)}^{\text{in}} \\ &= \{j \mid \min_{z \in Z}(i + q - \delta(\mathcal{P}_{b.y, a.x})) < j < \max_{z \in Z}(i + q + \delta(\mathcal{P}_{a.x, b.y}))\} \end{aligned} \quad (12.2)$$

This set can be further simplified as follows: For two tuples $z, z' \in \mathcal{Z}$ with $z < z'$ we can differ between the cases $q < q'$ and $q = q' \wedge x < x'$. According to the HSDF expansion of the precedence constraint analysis model, which is exemplified in Figure 12.2, all phases of an actor lie on a cycle with one token, which implies $\forall_{x, x'}: |\delta(\mathcal{P}_{a.x, b.y}) - \delta(\mathcal{P}_{a.x', b.y})| \leq 1$. For the first case it therefore follows that $q + \delta(\mathcal{P}_{a.x, b.y}) \leq q' + \delta(\mathcal{P}_{a.x', b.y})$. In the second case we have $x < x'$. As exemplified in Figure 12.6 for $v_{a.x_{out}} = v_{a.2}$ we can always find a phase $v_{a.x_{out}}$ such that $\forall_{x \leq x_{out}}: \delta(\mathcal{P}_{a.x, b.y}) = \delta_{a, b.0}^{\min}$ and $\forall_{x > x_{out}}: \delta(\mathcal{P}_{a.x, b.y}) = \delta_{a, b.0}^{\min} + 1$. Consequently it holds that $\delta(\mathcal{P}_{a.x, b.y}) \leq \delta(\mathcal{P}_{a.x', b.y})$ and we can also conclude for the second case that $q + \delta(\mathcal{P}_{a.x, b.y}) \leq q' + \delta(\mathcal{P}_{a.x', b.y})$.

From this follows that for any set $\mathcal{Z}$ the maximum function in Equation 12.2 is maximal for the supremum $\hat{\mathcal{Z}}$ and with an analogous reasoning that the minimum function is minimal for the infimum $\check{\mathcal{Z}}$, i.e.:

$$P_{v_{b.y} \rightarrow \mathcal{Z}(i)} = \{j \mid i + \check{q} - \delta(\mathcal{P}_{b.y, a.\check{x}}) < j < i + \hat{q} + \delta(\mathcal{P}_{a.\hat{x}, b.y})\} \quad (12.3)$$

With that we can define the interference characterization $\zeta_{b.y}(\mathcal{Z})$ on a set $\mathcal{Z}$ as the number of elements in $P_{v_{b.y} \rightarrow \mathcal{Z}(i)}$:

$$\zeta_{b.y \rightarrow a}(\mathcal{Z}) = |P_{v_{b.y} \rightarrow \mathcal{Z}(i)}| = \delta(\mathcal{P}_{a.\hat{x}, b.y}) + \hat{q} + \delta(\mathcal{P}_{b.y, a.\check{x}}) - \check{q} - 1 \quad (12.4)$$

Finally note that both $\eta_{b.y}(\Delta\tau)$ and $\zeta_{b.y \rightarrow a}(\mathcal{Z})$ represent upper bounds on the number of executions of a task phase $t_{b.y}$ that can interfere with the corresponding task phase executions of the firings in $\mathcal{Z}$, provided that $\Delta\tau$ is an upper bound on the time needed to execute the task phases of t_a included in $\mathcal{Z}$. This allows us to combine the two for a tighter bound on interference as follows:

$$\gamma_{b.y \rightarrow a}(\Delta\tau, \mathcal{Z}) = \min(\eta_{b.y}(\Delta\tau), \zeta_{b.y \rightarrow a}(\mathcal{Z}))$$

With that we have presented all components of our period-and-jitter based maximum response time computation for task phases.

12.4.5 MAXIMUM RESPONSE TIMES OF TASK PHASES BASED ON EXECUTION INTERVALS

In this section we replace the period-and-jitter based maximum response time computation for task phases with a computation based on execution intervals. For that matter we first reconsider the execution interval interference characterization for entire tasks from Chapter 11. Then we extend this characterization to a characterization applicable for task phases and lastly modify the algorithm depicted in Figure 12.5 such that it uses the execution interval based characterization instead of the period-and-jitter one.

In Section 11.3.5 we have derived that the interference of a task t_b on the maximum busy period $\hat{w}_a^\diamond(q)$ of a task t_a can be conservatively bounded from above with the following interference characterization:

$$\psi_{b \rightarrow a}(\hat{w}_a^\diamond(q), q) = \begin{cases} \psi_{b \rightarrow a}^*(\hat{w}_a^\diamond(q), q) & G_b^T = G_a^T \\ \psi'_{b \rightarrow a}(\hat{w}_a^\diamond(q)) & \text{otherwise} \end{cases}$$

with

$$\begin{aligned}\psi_{b \rightarrow a}^*(\hat{w}_a^\diamond(q), q) &= \min\left(\left\lceil \frac{\hat{s}_a + \hat{w}_a^\diamond(q) - \hat{s}_b}{P_b} \right\rceil, \delta(\mathcal{P}_{a,b}) + q\right) + \left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a}{P_b} \right\rceil - 1 \\ \psi'_{b \rightarrow a}(\hat{w}_a^\diamond(q)) &= \left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_b + \hat{w}_a^\diamond(q)}{P_b} \right\rceil\end{aligned}$$

Remember that we have obtained this interference characterization from two sets, one based on execution intervals and the other on precedence constraints, which were defined as follows:

$$\begin{aligned}E_{t_b \rightarrow \hat{w}_a^\diamond(q)} &= \{j \mid \left\lfloor \frac{\hat{s}_a + (i - q) \cdot P_a - \hat{s}_b - \hat{\rho}_b}{P_b} \right\rfloor < j < \left\lceil \frac{\hat{s}_a + \hat{w}_a^\diamond(q) + (i - q) \cdot P_a - \hat{s}_b}{P_b} \right\rceil\} \\ P_{t_b \rightarrow \hat{w}_a^\diamond(q)} &= \{j \mid i - q - \delta(\mathcal{P}_{b,a}) < j < i + \delta(\mathcal{P}_{a,b})\}\end{aligned}$$

Based on these sets, let us now derive an interference characterization suitable for task phases. The execution interval set $E_{t_b \rightarrow \hat{w}_a^\diamond(q)}$ includes all iterations j of a task t_b that can occur during the maximum busy period $\hat{w}_a^\diamond(q)$ of a task t_a beginning at $\hat{e}_a^{ext}(i - q) = \hat{s}_a + (i - q) \cdot P_a$. We can transform this set to a set $E_{t_{b,y} \rightarrow \hat{w}_{a,x'}^\diamond}$ that gives us all iterations j of a task phase $t_{b,y}$ that can take place within the maximum busy period $\hat{w}_{a,x'}^\diamond$ of a task t_a beginning at $\hat{e}_{a,x'}^{ext}(i - q) = \hat{s}_{a,x'}^{ext} + (i - q) \cdot P_a$ by simply replacing all task parameters with the parameters of the respective phases, i.e.:

$$\begin{aligned}E_{t_{b,y} \rightarrow \hat{w}_{a,x'}^\diamond} &= \{j \mid \left\lfloor \frac{\hat{s}_{a,x'}^{ext} + (i - q) \cdot P_a - \hat{s}_{b,y} - \hat{\rho}_{b,y}}{P_b} \right\rfloor < j < \left\lceil \frac{\hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^\diamond + (i - q) \cdot P_a - \hat{s}_{b,y}}{P_b} \right\rceil\}\end{aligned}$$

Analogously we require a set that gives the iterations j of a phase $t_{b,y}$ which can occur during the executions of all phases of task t_a included as tuples in a set $\mathcal{Z}$, based on precedence constraints. The included tuples must refer to the same phases of t_a that are also considered in $\hat{w}_{a,x'}^\diamond$. The first task phase execution considered in $\hat{w}_{a,x'}^\diamond$ is execution $i - q$ of phase $t_{a,x'}$, whereas the last is execution i of a phase $t_{a,x}$. As it further holds for the set $\mathcal{Z}_{a,x'}$ considered in the algorithm in Figure 12.5 that its infimum is $\check{\mathcal{Z}}_{a,x'} = (v_{a,x'}, 0)$ and that its supremum equals $\check{\mathcal{Z}}_{a,x'} = (v_{a,x}, q)$ it follows that we have to shift $P_{v_{b,y} \rightarrow \mathcal{Z}(i)}$ from Equation 12.3 by $-q = -\hat{q}$. This results in the following precedence set:

$$P_{v_{b,y} \rightarrow \mathcal{Z}_{a,x'}(i - \hat{q})} = \{j \mid i - \hat{q} - \delta(\mathcal{P}_{b,y,a,\hat{x}}) < j < i + \delta(\mathcal{P}_{a,\hat{x},b,y})\}$$

Both $E_{t_{b,y} \rightarrow \hat{w}_{a,x'}^\diamond}$ and $P_{v_{b,y} \rightarrow \mathcal{Z}_{a,x'}(i - \hat{q})}$ represent conservative upper bounds on the interference of a task phase $t_{b,y}$ that can occur within $\hat{w}_{a,x'}^\diamond$ defined between

iteration $i - q$ of phase $t_{a.x'}$ and iteration i of phase $t_{a.x}$. Moreover, both are of the same form as $E_{t_b \rightarrow \hat{w}_a^\diamond(q)}$ and $P_{t_b \rightarrow \hat{w}_a^\diamond(q)}$ from Section 11.3, such that we can also apply the same reasoning as in that section to derive an execution interval based interference characterization for task phases. The resulting characterization is as follows:

$$\psi_{b.y \rightarrow a}(\hat{s}_{a.x'}^{ext}, \hat{w}_{a.x'}^\diamond, \mathcal{Z}_{a.x'}) = \begin{cases} \psi_{b.y \rightarrow a}^*(\hat{s}_{a.x'}^{ext}, \hat{w}_{a.x'}^\diamond, \mathcal{Z}_{a.x'}) & G_b^T = G_a^T \\ \psi'_{b.y \rightarrow a}(\hat{w}_{a.x'}^\diamond) & \text{otherwise} \end{cases}$$

with

$$\begin{aligned} \psi_{b.y \rightarrow a}^*(\hat{s}_{a.x'}^{ext}, \hat{w}_{a.x'}^\diamond, \mathcal{Z}_{a.x'}) &= \min \left(\left\lceil \frac{\hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^\diamond - \check{s}_{b.y}}{P_b} \right\rceil, \delta(\mathcal{P}_{a.\hat{x}, b.y}) + \hat{q} \right) + \\ &\quad \left\lceil \frac{\hat{s}_{b.y} + \hat{\rho}_{b.y} - \hat{s}_{a.x'}^{ext}}{P_b} \right\rceil - 1 \\ \psi'_{b.y \rightarrow a}(\hat{w}_{a.x'}^\diamond) &= \left\lceil \frac{\hat{s}_{b.y} + \hat{\rho}_{b.y} - \check{s}_{b.y} + \hat{w}_{a.x'}^\diamond}{P_b} \right\rceil \end{aligned}$$

Using this interference characterization we can finally redefine the algorithm in Figure 12.5 such that it uses execution intervals instead of period-and-jitter to bound interference, as depicted in Figure 12.7 (with $\psi_{b.y \rightarrow a}(\hat{s}_{a.x'}^{ext}, 0, \emptyset) = 0$). We make use of a maximum busy period computation analogous to the one in Section 11.3, only applied on task phases instead of tasks. Note that we can replace the terms $\mathcal{Z}_{a.x'} \cup \{(v_{a.x}, q)\}$ from the algorithm in Figure 12.5 with $\{(v_{a.x}, q)\}$ because the interference characterization $\psi_{b.y \rightarrow a}$ only requires the suprema of sets $\mathcal{Z}_{a.x'}$ and $\mathcal{Z}_{a.x'} \cup \{(v_{a.x}, q)\}$, which are always equal to the last added tuples $\{(v_{a.x}, q)\}$.

From the determined maximum finish times of task phases we can then also compute maximum response times, exactly in the same way as described in Section 12.4.3. The only difference is that we additionally need to clamp maximum response times between different flow iterations as described in the end of Section 11.3.6, which is required to ensure monotonicity of the analysis flow.

12.5 PROOF OF CONSERVATIVENESS

Now that we have described all steps of our analysis flow in Figure 12.1 we still need to prove that the computed results do not only hold for the analysis models, but also for the underlying application model.

Like in the previous chapters we first show that the maximum response times as computed in Sections 12.4.3 and 12.4.5 are conservative upper bounds for the task phases in the response time analysis model, i.e. upper bounds on the differences between maximum external enabling and finish times of the first phases of tasks and on the differences between maximum total enabling and finish times of all other phases, respectively. From Section 9.3.2 follows that the

```

1   $\forall_{0 \leq x < \Theta_a^r} : \hat{f}_{a,x} = 0;$ 
2  for all  $(x' : e_{b,y,a,x'} \in E^{ext}) \{$ 
3     $x = x'; q = 0; \hat{w}_{a,x'}^\circ = 0; \mathcal{Z}_{a,x'} = \emptyset;$ 
4    do  $\{$ 
5       $\hat{w}_{a,x'}^{\circ\oplus} = 0;$ 
6       $\hat{w}_{a,x'}^{\circ\oplus} = \hat{\chi}_{a,x} + \sum_{t_{b,y} \in hpa} [\psi_{b,y \rightarrow a}(\hat{s}_{a,x'}^{ext}, \hat{w}_{a,x'}^\circ + \hat{w}_{a,x'}^{\circ\oplus}, \{(v_{a,x}, q)\}) -$ 
7         $\psi_{b,y \rightarrow a}(\hat{s}_{a,x'}^{ext}, \hat{w}_{a,x'}^\circ, \mathcal{Z}_{a,x'})] \cdot \hat{\chi}_{b,y};$ 
8       $\hat{w}_{a,x'}^\circ = \hat{w}_{a,x'}^\circ + \hat{w}_{a,x'}^{\circ\oplus};$ 
9       $\mathcal{Z}_{a,x'} = \{(v_{a,x}, q)\};$ 
10      $\hat{f}_{a,x} = \max(\hat{f}_{a,x}, \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^\circ - q \cdot Pa);$ 
11      $x ++;$ 
12     if  $(x = \Theta_a^r) \{$ 
13        $q ++; x = 0;$ 
14      $\}$ 
15    $\}$  while  $(x \neq x' \mid \mid \hat{w}_{a,x'}^\circ > q \cdot Pa);$ 

```

Figure 12.7: Algorithm to compute upper bounds on finish times of task phases using execution intervals.

worst-case analysis model is a valid bounding abstraction of the response time analysis model if such maximum response times are used as firing durations. This relation is captured by the following lemma:

Lemma 12.1 (Worst-Case Conservativeness). *Let the firing durations $\hat{\rho}_{a,x}$ of phases of actor components v_a in the worst-case analysis model be set to the respective $\hat{R}_{a,x}$ and let $\hat{e}_{a,x}(i)$ ($\hat{e}_{a,x}^{ext}(i)$) be the strictly periodic upper bounds on the respective (external) enabling times used in the restricted task components t_a in the response time analysis model. Then it holds that $G^{WC} \supseteq G^{RT}$ and $in_{G^{WC}} \supseteq in_{G^{RT}}$, i.e. the worst-case analysis model is a valid worst-case bounding abstraction of the response time analysis model.*

Proof. To prove $G^{WC} \supseteq G^{RT}$ and $in_{G^{WC}} \supseteq in_{G^{RT}}$ we need to show that the right inequalities of the firing duration criterion for task phases in Equation 9.8 applies, given that the maximum response times derived according to Section 12.4.3 or Section 12.4.5 are used as firing durations. For that matter we first reformulate the relevant conditions of the firing duration criterion using the notation introduced in the end of Section 12.2. For the first phases $t_{a,0}$ in the response time analysis model it must hence hold for all iterations i of task t_a :

$$\varepsilon_{a,0}(i) + \chi_{a,0}(i) + \mathcal{I}_a[\varepsilon_{a,0}(i), f_{a,0}(i)] \leq \hat{e}_{a,0}^{ext}(i) + \hat{\rho}_{a,0}$$

And for all other phases $t_{a,x}$ with $x \neq 0$ it is required that:

$$\varepsilon_{a,x}(i) + \chi_{a,x}(i) + \mathcal{I}_a[\varepsilon_{a,x}(i), f_{a,x}(i)] \leq \hat{e}_{a,x}(i) + \hat{\rho}_{a,x}$$

In the former inequality let us substitute $\hat{\rho}_{a,0}$ with the corresponding maximum response time $\hat{R}_{a,0} = \hat{f}_{a,0} - \hat{s}_{a,0}^{ext}$ and in the latter inequality $\hat{\rho}_{a,x}$ with $\hat{R}_{a,x} = \hat{f}_{a,x} - \max(\hat{s}_{a,x}^{ext}, \hat{f}_{a,x-1})$. With $\hat{\varepsilon}_{a,0}^{ext}(i) = \hat{s}_{a,0}^{ext} + i \cdot P_a$ and $\hat{\varepsilon}_{a,x}(i) = \max(\hat{s}_{a,x}^{ext}, \hat{f}_{a,x-1}) + i \cdot P_a$ we derive from both inequalities:

$$\varepsilon_{a,x}(i) + \chi_{a,x}(i) + \mathcal{I}_a[\varepsilon_{a,x}(i), f_{a,x}(i)] \leq \hat{f}_{a,x} + i \cdot P_a$$

For the proof of this inequality we bring its left side to a different form by using the actual busy periods of tasks in the response time analysis model that we define analogously to the ones in the proof of Lemma 10.3. Let $w_{a,x'}(i)$ be an actual busy period that precedes an iteration i of a task phase $t_{a,x}$ in the response time analysis model. Furthermore, let $\mathcal{W}$ be a set that includes all phases forming the actual busy period, with $t_{a,x'}(i') \in \mathcal{W}$ the phase initiating the busy period and with $t_{a,x}(i) \in \mathcal{W}$ the phase concluding the busy period (with $t_{a,x}(i)$ being a shorthand notation for iteration i of phase $t_{a,x}$). We require that the first phase iteration is externally enabled, while all other phases in the busy period are self-delaying each other, i.e. it must hold that $\varepsilon_{a,x'}(i') = \varepsilon_{a,x'}^{ext}(i')$ and $\forall t_{a,x^*}(i^*) \in \mathcal{W} \setminus \{t_{a,x'}(i')\} : \varepsilon_{a,x^*}(i^*) > \varepsilon_{a,x^*}^{ext}(i^*)$. Using the definition of self-delay and Equation 9.2 we can formulate the actual busy period as follows:

$$w_{a,x'}(i) = \sum_{t_{a,x^*}(i^*) \in \mathcal{W}} \chi_{a,x^*}(i^*) + \mathcal{I}_a[\varepsilon_{a,x'}^{ext}(i'), \varepsilon_{a,x'}^{ext}(i') + w_{a,x'}(i)]$$

By definition, it holds that $\varepsilon_{a,x'}^{ext}(i') + w_{a,x'}(i)$ is equal to the left side of above criterion. With that it follows for the period-and-jitter method to compute maximum response times of task phases:

$$\begin{aligned} & \varepsilon_{a,x}(i) + \chi_{a,x}(i) + \mathcal{I}_a[\varepsilon_{a,x}(i), f_{a,x}(i)] = \varepsilon_{a,x'}(i') + w_{a,x'}(i) \\ &= \varepsilon_{a,x'}^{ext}(i') + \sum_{t_{a,x^*}(i^*) \in \mathcal{W}} \chi_{a,x^*}(i^*) + \mathcal{I}_a[\varepsilon_{a,x'}^{ext}(i'), \varepsilon_{a,x'}^{ext}(i') + w_{a,x'}(i)] \\ &\leq \hat{\varepsilon}_{a,x'}^{ext}(i') + \sum_{t_{a,x^*}(i^*) \in \mathcal{W}} \hat{\chi}_{a,x^*} + \sum_{t_{b,y} \in hp_a} \gamma_{b,y \rightarrow a}(\hat{w}'_{a,x'}, \mathcal{Z}_{a,x'}) \cdot \hat{\chi}_{b,y} \\ &= \hat{s}_{a,x'}^{ext} + i' \cdot P_a + \hat{w}_{a,x'} \leq \hat{f}_{a,x} + i \cdot P_a \end{aligned}$$

Equality between the first and second lines follows immediately from the definition of the actual busy period, which implies $\varepsilon_{a,x'}(i') = \varepsilon_{a,x'}^{ext}(i')$. The inequality between the second and the third line applies because of four reasons: First, the phases x' of restricted task components t_a in the response time analysis model can only have external enabling times $\varepsilon_{a,x'}^{ext}(i')$ smaller or equal to $\hat{\varepsilon}_{a,x'}^{ext}(i')$. Second, it holds that $\forall x^*, i^* : \chi_{a,x^*}(i^*) \leq \hat{\chi}_{a,x^*}$. Third, the bound on interference in line 3 is by construction conservative to the actual interference that can occur in the response time analysis model, given the same busy periods as parameters. And fourth, we can conclude with a similar discussion on fixed points as in the proof of Lemma 10.3 that $w_{a,x'}(i) \leq \hat{w}'_{a,x'}$.

In the fifth line we replace the upper bound on the external enabling time $\hat{\varepsilon}_{a.x'}^{ext}(i')$ with $\hat{s}_{a.x'}^{ext} + i' \cdot P_a$ and the right-hand expression from the fourth line with $\hat{w}_{a.x'}$. Finally we use $\hat{f}_{a.x} \geq \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'} - q \cdot P_a$ according to the algorithm in Figure 12.5, as well as $q = i - i'$ by definition.

For the execution interval based method we further define analogously to Section 11.3:

$$\hat{w}_{a.x'} = \hat{w}_{a.x'}^* + \hat{w}_{a.x'}^\diamond \text{ with } \hat{w}_{a.x'}^* = \hat{\varepsilon}_{a.x'}^{ext}(i') - \varepsilon_{a.x'}^{ext}(i') = \hat{s}_{a.x'}^{ext} + i' \cdot P_a - \varepsilon_{a.x'}^{ext}(i')$$

With that it follows:

$$\begin{aligned} & \varepsilon_{a.x}(i) + \chi_{a.x}(i) + \mathcal{I}_a[\varepsilon_{a.x}(i), f_{a.x}(i)] = \varepsilon_{a.x'}(i') + w_{a.x'}(i) \\ & = \varepsilon_{a.x'}^{ext}(i') + \sum_{t_{a.x^*}(i^*) \in \mathcal{W}} \chi_{a.x^*}(i^*) + \mathcal{I}_a[\varepsilon_{a.x'}^{ext}(i'), \varepsilon_{a.x'}^{ext}(i') + w_{a.x'}(i)] \\ & \leq \varepsilon_{a.x'}^{ext}(i') + \hat{w}_{a.x'}^* + \sum_{t_{a.x^*}(i^*) \in \mathcal{W}} \hat{\chi}_{a.x^*} + \sum_{t_{b.y} \in hp_a} \psi_{b.y \rightarrow a}(\hat{s}_{a.x'}^{ext}, w_{a.x'}(i) - \hat{w}_{a.x'}^*, \mathcal{Z}_{a.x'}) \cdot \hat{\chi}_{b.y} \\ & \leq \varepsilon_{a.x'}^{ext}(i') + \hat{w}_{a.x'}^* + \sum_{t_{a.x^*}(i^*) \in \mathcal{W}} \hat{\chi}_{a.x^*} + \sum_{t_{b.y} \in hp_a} \psi_{b.y \rightarrow a}(\hat{s}_{a.x'}^{ext}, \hat{w}_{a.x'} - \hat{w}_{a.x'}^*, \mathcal{Z}_{a.x'}) \cdot \hat{\chi}_{b.y} \\ & = \hat{s}_{a.x'}^{ext} + i' \cdot P_a + \hat{w}_{a.x'}^\diamond \leq \hat{f}_{a.x} + i \cdot P_a \end{aligned}$$

The reasoning between the first and the second line is the same as above, whereas the explanation of the remainder is analogous to the explanation given in the proof of Lemma 11.1.

Thus we can conclude that if the maximum response times $\hat{R}_{a.x}$ determined in this chapter are used as firing durations $\hat{\rho}_{a.x}$ in the worst-case analysis model it follows that the firing duration criterion for multi-phase analysis models is satisfied for all restricted tasks and actors. From the latter follows $G^{WC} \supseteq G^{RT}$ and $in_{G^{WC}} \supseteq in_{G^{RT}}$. $\square$

Moreover we require that the response time analysis model is a valid inclusion abstraction of the application model on convergence of our flow, which implies that all bounds determined with our multi-phase flow are also valid bounds on the respective parameters in the application model. Hence we prove the following theorem:

Theorem 12.1 (Analysis Conservativeness). *On convergence of our analysis flow it holds that $G^{RT} \supseteq G^\Lambda$ and $in_{G^{RT}} \supseteq in_{G^\Lambda}$, i.e. the response time analysis model is a valid inclusion abstraction of the application model. This implies that on convergence also the worst-case, best-case and precedence constraint analysis models are valid worst-case, best-case and inclusion abstractions of the application model, respectively.*

Proof. Analogous to the proof of Theorem 10.1. $\square$

After having presented our approaches to analyze the temporal behavior of multi-phase tasks, which enable to capture both native multi-phase tasks as well as single-phase tasks with non-atomic acquires and releases in an accurate manner, we present an extension of the execution interval technique in this section. With this extension we allow to model and analyze applications in which resource sharing is not limited to shared processors, but can involve multiple shared resources, such as for instance shared memories, shared buses or shared peripherals.

For simplicity we assume that all shared resources are scheduled with an SPP scheduler (an extension to different scheduler types such as RR or TDM would be rather straightforward, but would also unnecessarily complicate the presented formulations). Besides, we consider the same multi-phase application model as before, except for the following two differences:

First, we allow different phases of a task t_a to not only have different higher priority tasks, but different higher priority task phases. For example, this modification enables to model task phases that experience interference with higher priority phases executed on the same shared processor, as well as with entirely different higher priority phases that are accessing the same shared memory. Likewise, the modification allows to express other task phases of the same task which do not access shared memory, such that they only experience higher priority interference due to processor sharing. To take this into account we do not use a set hp_a anymore that returns all task phases that can interfere with any of the phases of task t_a , but define sets $hp_{a,x}$ that return all phases having a higher priority than task phase $t_{a,x}$ instead.

And second, we consider that while higher priority task phases on the same processor need to be considered with their entire WCETs, for different types of resource sharing smaller amounts of interference may be sufficient. For instance, it may occur that a higher priority phase $t_{b,y}$ takes maximally $\hat{\chi}_{b,y}$ on the processor it is executing, but only half of this time for the access of a shared bus, i.e. $0.5 \cdot \hat{\chi}_{b,y}$. For a lower priority task phase on the same bus this would mean that also only half the WCET needs to be taken into account as interference. We capture this circumstance by introducing potentially reduced WCETs $\hat{\chi}_{b,y}^{a,x}$ with $0 < \hat{\chi}_{b,y}^{a,x} \leq \hat{\chi}_{b,y}$, representing the maximum time that the execution of a phase $t_{b,y}$ can interfere with the execution of a phase $t_{a,x}$ on the respective shared resource.

With this modified application model in mind, we can construct an analysis technique that can be used for multi-resource applications. We consider the same analysis flow as introduced in Section 12.1 and keep all steps the same, except for the algorithm to compute maximum finish times considering interference. Before we present our new algorithm, which is based on the algorithm using execution intervals in Figure 12.7, let us first discuss the basic idea of computing maximum busy periods of tasks that are subject to interference from multiple shared resources.

The main difference between the use-cases with processor sharing only and with multiple shared resources is that in the latter case not necessarily all higher priority phases can interfere with all lower priority phases. To understand the implications of this consider the example depicted in Figure 12.8. The Gantt chart in this figure illustrates the computation of a maximum busy period $\hat{w}_{a,0}^\circ$ that takes interference of two higher priority phases $t_{b,y}$ and $t_{c,z}$ into account. We assume that the two higher priority phases cannot interfere with all phases of task t_a , but only with the phases $t_{a,0}$ and $t_{a,2}$.

Figure 12.8: Example of a maximum busy period computation using interference periods for the consideration of multiple shared resources.

Let us first examine what would happen if we applied the method to compute maximum busy periods presented in Figure 12.7 on this example and if we only considered interference for each maximum busy period extension that can actually interfere with the respective phases, i.e. if we only considered higher priority phases $t_{b,y} \in hp_{a,x}$ instead of $t_{b,y} \in hp_a$ (with $hp_a = \bigcup_{0 \leq x < \Theta_a^r} hp_{a,x}$).

With respect to interference of phase $t_{b,y}$, the maximum busy period $\hat{w}_{a,0}^\circ$ over phase executions $t_{a,0}(0)$ and $t_{a,1}(0)$ would include two interferences of phase $t_{b,y}$, due to the overlaps with the execution intervals $\mathcal{E}_{b,y}(0)$ and $\mathcal{E}_{b,y}(1)$. The overlaps with $\mathcal{E}_{b,y}(2)$ and $\mathcal{E}_{b,y}(3)$ would be ignored because phase $t_{b,y}$ cannot interfere with phase $t_{a,1}$. For the computation of interference with phase execution $t_{a,2}(0)$ we would then consider all overlaps of execution intervals of phase $t_{b,y}$ with the whole, extended maximum busy period from the beginning of phase execution $t_{a,0}(0)$ until the end of phase execution $t_{b,2}(0)$, i.e. overlaps of $\mathcal{E}_{b,y}(0)$ to $\mathcal{E}_{b,y}(5)$, and deduct from that the number of overlaps of execution intervals of $t_{b,y}$ with the maximum busy period up to the beginning of phase execution $t_{a,2}(0)$, i.e. deduct overlaps of $\mathcal{E}_{b,y}(0)$ to $\mathcal{E}_{b,y}(3)$. This would result in an optimistic consideration of interference since the overlap with $\mathcal{E}_{b,y}(3)$ would be neither considered in the maximum busy period up to the end of phase execution $t_{a,1}(0)$, nor taken into account in the extension over phase execution $t_{a,2}(0)$.

To resolve this issue we introduce so-called interference periods $\hat{w}^{b,y}$ and $\hat{w}^{c,z}$ that keep track over which segments of the maximum busy period $\hat{w}_{a,0}^\circ$ interference of the respective phases is actually considered. With such interference periods we can conservatively and accurately bound interference of higher priority phases as follows.

First, let us consider the interference period $\hat{w}^{c,z}$ of higher priority phase $t_{c,z}$. Initially, we set the starting point of the interference period $\hat{s}_{a,0}^{c,z}$ to $\hat{s}_{a,0}^{ext}$ and its length to 0. In the computation of the first maximum busy period extension $\hat{w}_{a,0}^{\circ\oplus}$ over phase execution $t_{a,0}(0)$, which is illustrated with the first black arrow in Figure 12.8, we consider interference of phase $t_{c,z}$ fully, taking into account the overlap with $\mathcal{E}_{c,z}(0)$. Hence we know that up to the end of phase execution $t_{a,0}(0)$ we have conservatively included interference of phase $t_{c,z}$, allowing us to extend $\hat{w}^{c,z}$ by $\hat{w}_{a,0}^{\circ\oplus}$, as indicated by the first red arrow. In the computation of the second maximum busy period extension illustrated with the second black arrow we do not consider interference of phase $t_{c,z}$, such that $\hat{w}^{c,z}$ is also not extended. In the third extension we consider interference of phase $t_{c,z}$ over the entire, extended maximum busy period from the beginning of phase execution $t_{a,0}(0)$ until the end of $t_{a,2}(0)$ and deduct from it the already considered interference, i.e. deduct the interference considered throughout the interference period $\hat{w}^{c,z}$. With that we take all additional interference into account that can occur between the end of phase execution $t_{a,0}(0)$ and the end of $t_{a,2}(0)$ (which is no additional interference in this case). This allows us to also extend $\hat{w}^{c,z}$ until the end of execution $t_{a,2}(0)$, as indicated by the second red arrow. The resulting extended interference period then forms the basis to consider interference of phase $t_{c,z}$ for all later maximum busy period extensions.

This method of extending interference periods can be used to conservatively bound interference in the multiple shared resource case. But by itself it is not necessarily the most accurate method, as can be seen for the other higher priority phase $t_{b,y}$.

Consider the interference of phase $t_{b,y}$ with the maximum busy period extension $\hat{w}_{a,0}^{\circ\oplus}$ for phase execution $t_{a,2}(0)$. With the extension method all interference of phase $t_{b,y}$ would be included that could occur between the end of the last interference period $\hat{w}^{b,y}$, which is indicated by the first green arrow, and the end of phase execution $t_{a,2}(0)$, which is indicated by the third black arrow. The extension method would thus account for overlaps with the execution intervals $\mathcal{E}_{b,y}(3)$ to $\mathcal{E}_{b,y}(5)$, but also for the overlap with $\mathcal{E}_{b,y}(2)$. However, the latter consideration would result in a significant over-approximation since the phase execution $t_{b,y}(2)$ captured with $\mathcal{E}_{b,y}(2)$ cannot interfere with any of the phase executions $t_{a,0}(0)$, $t_{a,1}(0)$ and $t_{a,2}(0)$.

A more accurate method would be to not consider interference over the whole maximum busy period from phase execution $t_{a,0}(0)$ to $t_{a,2}(0)$ and to deduct from it the interference already considered in $\hat{w}^{b,y}$, but to reinitialize the consideration of interference instead. For that matter we simply take all interference of phase $t_{b,y}$ that can occur during the maximum busy period extension over phase execution $t_{a,2}(0)$, i.e. interference due to overlaps with execution intervals $\mathcal{E}_{b,y}(3)$ to $\mathcal{E}_{b,y}(5)$, and do not deduct any previously considered interference. Thereby we correctly account for interference due to overlaps with $\mathcal{E}_{b,y}(0)$, $\mathcal{E}_{b,y}(1)$, $\mathcal{E}_{b,y}(3)$, $\mathcal{E}_{b,y}(4)$ and $\mathcal{E}_{b,y}(5)$, but not $\mathcal{E}_{b,y}(2)$. For future extensions of the maximum busy period, e.g. for the extension $\hat{w}_{a,0}^{\circ\oplus}$ over phase execution $t_{a,0}(1)$ we also need

to reinitialize the corresponding interference period accordingly, by setting the starting point $\hat{s}^{b,y}$ to the beginning of phase execution $t_{a,2}(0)$ and the length to the respective maximum busy period extension $\hat{w}_{a,0}^{\diamond\oplus}$, as indicated by the second green arrow. For the consideration of interference over phase execution $t_{a,0}(1)$ we can then extend this interference period instead of reinitializing it, as shown with the third green arrow. Otherwise, we would consider the overlap with execution interval $\mathcal{E}_{b,y}(5)$ twice.

Both methods of extending and reinitializing interference periods result in temporally conservative interference characterizations. This allows us to simply apply both methods for each higher priority phase on each maximum busy period extension and to consider the minimum of the resulting amounts of interference thereafter.

Taking these observations into account we can now discuss our algorithm for the computation of maximum finish times of task phases in the multiple shared resource case, which is presented in Figure 12.9. The differences to the execution interval algorithm for the single shared resource case in Figure 12.7 are as follows. In line 4 of the algorithm, the interference periods $\hat{w}^{b,y}$, their starting points $\hat{s}^{b,y}$ and the accompanying sets $\mathcal{Z}^{b,y}$, which are used to keep track up until which point in time interference of a phase $t_{b,y}$ has been considered, are initialized to $\hat{s}_{a,x'}^{ext}$, 0 and $\emptyset$, respectively. Remember that the set hp_a includes all higher priority phases that can interfere with any of the phases of task t_a .

In line 7 maximum busy period extensions are computed. The expression $t_{b,y} \in hp_{a,x}$ captures that higher priority interference is only considered if the respective higher priority phase cannot only interfere with a phase of task t_a , but can interfere with the currently analyzed phase $t_{a,x}$.

The two aforementioned methods to characterize higher priority interference are expressed by the minimum function in line 7 of our algorithm. The first argument of the minimum function corresponds to the reinitialization method in which interference of a higher priority phase $t_{b,y}$ is determined over the currently considered maximum busy period extension and without a reduction of previously derived interference. Hence $\psi_{b,y \rightarrow a}$ is only applied on the interval between $\hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{\diamond}$ and $\hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{\diamond} + \hat{w}_{a,x'}^{\diamond\oplus}$.

The second argument considers interference of a higher priority phase $t_{b,y}$ with the extension method. For that matter an upper bound on the interference of $t_{b,y}$ is determined over the entire timespan between the starting point of the latest interference period $\hat{s}^{b,y}$ and the end of the extended maximum busy period extension at $\hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{\diamond} + \hat{w}_{a,x'}^{\diamond\oplus}$. This interference bound is then reduced by the previously considered interference, i.e. the interference between $\hat{s}^{b,y}$ and $\hat{s}^{b,y} + \hat{w}^{b,y}$.

The usage of $\hat{\chi}_{b,y}^{a,x}$ instead of $\hat{\chi}_{b,y}$ reflects that a shared resource is not necessarily used for the entire execution time of a phase $t_{b,y}$, given that the shared resource is not a shared processor.

Depending on which method produces a more accurate result for a higher priority phase $t_{b,y}$, the respective interference periods are either reinitialized or

```

1   $\forall_{0 \leq x < \Theta'_a} : \hat{f}_{a,x} = 0;$ 
2  forall  $(x' : e_{b,y,a,x'} \in E^{ext}) \{$ 
3     $x = x'; q = 0; \hat{w}_{a,x'}^\circ = 0;$ 
4     $\forall_{t_{b,y} \in hp_a} : \hat{s}^{b,y} = \hat{s}_{a,x'}^{ext}, \hat{w}^{b,y} = 0, \mathcal{Z}^{b,y} = \emptyset;$ 
5    do  $\{$ 
6       $\hat{w}_{a,x'}^{\circ\oplus} = 0;$ 
7
8      
$$\hat{w}_{a,x'}^{\circ\oplus} = \hat{\chi}_{a,x} + \sum_{t_{b,y} \in hp_{a,x}} \min[ \overbrace{\psi_{b,y \rightarrow a}(\hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^\circ, \hat{w}_{a,x'}^{\circ\oplus}, \{(v_{a,x}, q)\})}^{m_0^{b,y}},$$

9
10     
$$\overbrace{\psi_{b,y \rightarrow a}(\hat{s}^{b,y}, \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^\circ - \hat{s}^{b,y} + \hat{w}_{a,x'}^{\circ\oplus}, \{(v_{a,x}, q)\})}^{m_1^{b,y}} -$$

11
12     
$$\overbrace{\psi_{b,y \rightarrow a}(\hat{s}^{b,y}, \hat{w}^{b,y}, \mathcal{Z}^{b,y})}^{m_1^{b,y}} ] \cdot \hat{\chi}_{b,y}^{a,x};$$

13
14     forall  $(t_{b,y} \in hp_{a,x}) \{$ 
15       if  $(m_0^{b,y} < m_1^{b,y}) \{$ 
16          $\hat{s}^{b,y} = \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^\circ, \hat{w}^{b,y} = \hat{w}_{a,x'}^{\circ\oplus};$ 
17       } else  $\{$ 
18          $\hat{w}^{b,y} = \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^\circ - \hat{s}^{b,y} + \hat{w}_{a,x'}^{\circ\oplus};$ 
19       }
20        $\mathcal{Z}^{b,y} = \{(v_{a,x}, q)\};$ 
21     }
22      $\hat{w}_{a,x'}^\circ = \hat{w}_{a,x'}^\circ + \hat{w}_{a,x'}^{\circ\oplus};$ 
23      $\hat{f}_{a,x} = \max(\hat{f}_{a,x}, \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^\circ - q \cdot P_a);$ 
24      $x++;$ 
25     if  $(x = \Theta'_a) \{$ 
26        $q++;$ 
27        $x = 0;$ 
28     }
29   } while  $(x \neq x' \mid \mid \hat{w}_{a,x'}^\circ > q \cdot P_a);$ 
30 }

```

Figure 12.9: Algorithm to compute upper bounds on finish times of task phases using execution intervals, with support for multiple shared resources.

extended in lines 8 to 15. If interference of a phase $t_{b,y}$ is considered in $\hat{w}_{a,x'}^{\circ\oplus}$ via a reinitialization (case $m_0^{b,y} < m_1^{b,y}$) then also the respective interference period is reinitialized in line 10, by setting the starting point to the beginning of the maximum busy period extension $\hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^\circ$ and the length to $\hat{w}_{a,x'}^{\circ\oplus}$. For the extension case considered in line 12 (case $m_0^{b,y} \geq m_1^{b,y}$) the starting point remains the same, but the length is extended up to and including the end of the maximum busy period extension $\hat{w}_{a,x'}^{\circ\oplus}$.

The set $\mathcal{Z}^{b,y}$ is set to $\{(v_{a,x}, q)\}$ in line 14 for both cases, which reflects that the respective interference periods all include interference up to and including the finish of task phase $t_{a,x}$ in iteration q , no matter which method is used for their determination. The remainder of the algorithm is equal to the one in Figure 12.7.

With that we have introduced an analysis technique suitable for the analysis of multi-resource applications, i.e. we can express and analyze both processor

sharing and the sharing of other resources, such as shared memories, buses or peripherals, in a single, integrated analysis flow.

Finally, note that there is a reason for not presenting an algorithm for multiple shared resources which uses period-and-jitter as interference characterization. The problem is that the interference characterizations $\eta_{b,y}(\Delta\tau)$ only give conservative upper bounds on the numbers of enablings of phases $t_{b,y}$ if all phases that can interfere with $t_{b,y}$ are included in the same maximum busy period computations as well. For single shared resources with SPP scheduling this condition is always satisfied. For multiple shared resources it can however occur that e.g. a task phase $t_{b,y}$ can interfere with a phase $t_{a,x}$ on a shared memory and that another phase $t_{c,z}$ can interfere with phase $t_{b,y}$ on a shared processor, while phase $t_{c,z}$ cannot (directly) interfere with phase $t_{a,x}$. In the maximum busy period computation of $t_{a,x}$ we would consequently consider phase $t_{b,y}$, but not phase $t_{c,z}$, violating aforementioned requirement. Of course we could additionally include phase $t_{c,z}$ as well, but this would result in a significant over-approximation.

In contrast, execution intervals give upper bounds on interference of task phases independent of the inclusion of other task phases in the respective maximum busy period computations, which makes them a suitable interference characterization for our multiple shared resource analysis.

12.7 CASE STUDY

In this section we demonstrate the benefits of our multi-phase approaches in a case study. First, we discuss the application of our multi-phase analysis technique on our well-known WLAN transceiver in Section 12.7.1. Thereafter we evaluate the performance of our integrated multi-resource analysis in Section 12.7.2.

12.7.1 MULTI-PHASE ANALYSIS

As in the previous two chapters, we analyze the task graph of a WLAN 802.11p transceiver [AHG07] which is used in safety-critical automotive applications like automated braking systems. A WLAN 802.11p transceiver has several modes and is executed on a multiprocessor system for performance reasons. We only consider the part of the task graph that is active during packet decoding mode. The task graph of the packet decoding mode is once again shown in Figure 12.10.

A strictly periodic source with a period P_{SRC} models the input of this task graph. For illustration purposes we vary the source period and try to minimize it for all our considered use-cases. Additionally, we consider an optional source jitter of $J_{\text{SRC}} = 2 \cdot P_{\text{SRC}}$. The BCET and WCET are denoted next to the tasks. The different colors of the tasks illustrate the task-to-processor mapping and the priorities are written next to the tasks (with π_1 the lowest).

In the following we discuss different modifications of the task graph such that it becomes a multi-phase application. For these modifications we compare the

Figure 12.10: Task graph of the packet decoder of a WLAN 802.11p transceiver.

analysis accuracy of our multi-phase analysis approaches (mp) that are considering task phases explicitly to the accuracy of the single-phase approaches (sp) discussed in the previous chapters, which necessarily consider task phases as separate tasks. We further differ between interference characterization based on period-and-jitter (p), execution intervals (e), as well as the additional consideration of precedence constraints (p.c and e.c).

The analysis results for the unmodified task graph as determined with the analysis approaches from the previous chapters are presented once more in Table 12.1 for convenience. As before, the periods P_{SRC} listed in the table are the minimum periods for which the approaches do not report a constraint violation (cv). For the cases in which no constraint violation occurs, the entries in the table correspond to the maximum end-to-end latencies from the source to the end of task t_{VIT} , constituting a measure of accuracy. Moreover, the minimum periods define the maximum guaranteeable throughputs, making them another accuracy measure. Note that the values in all tables presented in this case study are in μs and rounded up to three digits.

P_{SRC}	17	13	12	8	32	28	16	12
J_{SRC}			0		64	56	32	24
p	34	cv	cv	cv	129	cv	cv	cv
p.c	25	25	25	cv	97	89	65	cv
e	17	21	cv	cv	117	112	cv	cv
e.c	12	12	12	14	84	76	52	48

Table 12.1: End-to-end latencies obtained for unmodified task graph in Figure 12.10 (Legend: p = period-and-jitter, e = execution intervals, c = consideration of cyclic dependencies / precedence constraints to limit interference, cv = constraint violation).

For the first modification we assume to possess additional information about the tasks than only their BCETs and WCETs. Specifically, we consider that the read operations of all tasks take the fixed time of $0.5\mu\text{s}$ right after the beginnings of task executions and the write operations the same time at their ends. This allows us to split all tasks into three phases, as it is exemplified for task t_{EQ} in

Figure 12.11(a). Such a partitioning can in general lead to higher throughput guarantees, smaller required buffer capacities and can be used to relax the requirement on single-phase applications that for a task execution all acquires must happen atomically at its beginning and all releases atomically at its end. The analysis results obtained for this modification are presented in Table 12.2.

Figure 12.11: Considered multi-phase use-cases for the graph in Figure 12.10 (the gray arrows indicate the respective acquisitions and releases of space that are omitted in Figure 12.10, $\langle 1, 0^4 \rangle$ is shorthand for $\langle 1, 0, 0, 0, 0 \rangle$).

P_{SRC}	30	21	17	13	12	8	64	54	38	33	30	28	16	12
J_{SRC}				0			128	108	76	66	60	56	32	24
sp.p	60	cv	cv	cv	cv	cv	248	cv	cv	cv	cv	cv	cv	cv
sp.p.c	42	42	cv	cv	cv	cv	194	174	142	cv	cv	cv	cv	cv
mp.p	25	25	34	cv	cv	cv	179	159	136	129	cv	cv	cv	cv
mp.p.c	25	25	25	25	25	cv	161	141	109	99	93	89	65	cv
sp.e	26	32.5	cv	cv	cv	cv	228	216	cv	cv	cv	cv	cv	cv
sp.e.c	12	12	12	12	12	16	164	144	112	102	100	cv	cv	cv
mp.e	17	17	17	21	cv	cv	175	155	123	119	113	112	cv	cv
mp.e.c	12	12	12	12	12	14	148	128	96	86	80	76	52	48

Table 12.2: End-to-end latencies obtained for case (a) in Figure 12.11 (Legend: sp = single-phase analysis, mp = multi-phase analysis, p = period-and-jitter, e = execution intervals, c = consideration of cyclic dependencies / precedence constraints to limit interference, cv = constraint violation).

The first observation we can make by comparing the results from Tables 12.1 and 12.2 is that for our WLAN example the consideration of task phases does not give us improvements in neither throughput nor end-to-end latencies. This is due to the feedback loop imposing a rather strict throughput constraint on the tasks in the loop, which does not change with the presented modification because all phases remain “*in-the-loop*”. However, the results are also not worse if we apply our multi-phase analysis techniques (cases mp), which shows that we can relax the requirement of all acquires occurring atomically at the beginnings and all releases atomically at the ends of task executions without suffering a degradation of analysis accuracy.

In contrast, the analysis results obtained with the period-and-jitter analysis methods for single-phase applications (cases sp.p and sp.p.c) are significantly impacted by the introduction of task phases (up to 43 % lower throughput guarantees and up to 140 % higher end-to-end latencies for the strictly periodic source and even up to 67 % lower throughput guarantees for the jittered source), as for these analysis approaches interference is tripled by considering three phases per task. The execution interval analyses for single-phase applications achieve almost the same results as the respective analyses for multi-phase applications if they are applied on the scenario without source jitter. This is due to the fact that the joint consideration of execution intervals and precedence constraints is able to exclude most interference and then it also does not matter anymore whether no interference is considered for one or three phases. For the jittered source scenario we get a different picture. Here, more overlaps with execution intervals are determined and thus also more interference that is considered thrice. This results in a significant degradation of analysis accuracy with the single-phase approaches (up to 60% lower throughput guarantees), whereas the multi-phase approaches maintain the same accuracy as before the modification.

P_{SRC}	24	22	20	19	17	16	10	9	46	45	41	35	30	26	20	13
J_{SRC}					0				92	90	82	70	60	52	40	26
sp.p	40.5	cv	cv	cv	cv	cv	cv	cv	166	cv	cv	cv	cv	cv	cv	cv
sp.p.c	31.5	31.5	31.5	cv	cv	cv	cv	cv	133	131	123	111	101	cv	cv	cv
mp.p	40.5	40.5	cv	cv	cv	cv	cv	cv	166	164	163	cv	cv	cv	cv	cv
mp.p.c	31.5	31.5	31.5	31.5	31.5	31.5	cv	cv	133	131	123	111	101	92.5	80.5	cv
sp.e	22	22	22	22	cv	cv	cv	cv	156	154	cv	cv	cv	cv	cv	cv
sp.e.c	12.5	12.5	12.5	12.5	12.5	12.5	12.5	cv	114	112	104	92	81.5	73.5	cv	cv
mp.e	22	22	22	22	28.5	cv	cv	cv	150	153	148	140	cv	cv	cv	cv
mp.e.c	12.5	12.5	12.5	12.5	12.5	12.5	12.5	15	114	112	104	92	81.5	73.5	61.5	52

Table 12.3: End-to-end latencies obtained for case (b) in Figure 12.11 (Legend: sp = single-phase analysis, mp = multi-phase analysis, p = period-and-jitter, e = execution intervals, c = consideration of cyclic dependencies / precedence constraints to limit interference, cv = constraint violation).

The second considered modification can be used to relax the requirement of atomic synchronization on single-phase applications, even if a task cannot be split into phases like in the first use-case. If it can be guaranteed that all acquires happen in the first $0.25\mu\text{s}$ and all releases in the last $0.25\mu\text{s}$ of each task execution, we can model such variances with synchronization jitters that are added on top of the WCETs of the corresponding tasks, as exemplified for task t_{REENC} in Figure 12.11(b). The analysis results determined for this use-case can be found in Table 12.3.

As expectable, applying this relaxation on all tasks comes at the cost of worse results for all analysis approaches. Moreover, we do not observe any differences in

end-to-end latencies between single- and multi-phase approaches for the strictly periodic source scenario, which can be explained by the fact that also with the multi-phase approaches a reduction of interference only occurs on the “*out-of-the-loop*” phases one and three, whereas the externally enabled “*in-the-loop*” phases experience maximal interference. However, better throughput guarantees are determined with the multi-phase than with the single-phase approaches (up to 25% higher for period-and-jitter and up to 11% higher for execution intervals) as for small periods the cycles over the three phases become more critical than the feedback loop cycle, due to the overestimation of interference in the single-phase approaches. For the jittered source scenario the differences become even larger (up to 50% higher throughput guarantees for period-and-jitter and even up to 100% higher guarantees for execution intervals).

The third modification is a technique called task clustering [F⁺08] which can be used to remove synchronization overhead, as well as any interference between clustered tasks, leading to potentially better analysis results. We apply this technique on the tasks t_{DEMAP} , t_{DEINT} and t_{VIT} , resulting in the task cluster t_{DMDIVIT} depicted in Figure 12.11(c). The results obtained for this use-case are presented in Table 12.4.

P_{SRC}	15	11	10	9	8	27	25	17	16	15	12
J_{SRC}				0		54	50	34	32	30	24
sp.p	22	cv	cv	cv	cv	102	cv	cv	cv	cv	cv
sp.p.c	22	22	cv	cv	cv	84	80	64	62	cv	cv
mp.p	18	20	20	cv	cv	86	82	66	cv	cv	cv
mp.p.c	18	18	18	18	cv	80	76	60	58	60	cv
sp.e	12	12	12	12	14	92	92	cv	cv	cv	cv
sp.e.c	12	12	12	12	14	74	70	54	52	54	48
mp.e	12	12	12	12	14	82	78	62	62	cv	cv
mp.e.c	12	12	12	12	14	74	70	54	52	54	48

Table 12.4: End-to-end latencies obtained for case (c) in Figure 12.11 (Legend: sp = single-phase analysis, mp = multi-phase analysis, p = period-and-jitter, e = execution intervals, c = consideration of cyclic dependencies / precedence constraints to limit interference, cv = constraint violations).

Compared to the results for the unmodified graph in Table 12.1 we see indeed an improvement in both throughput and end-to-end latencies for both single-phase and multi-phase analyses. For the period-and-jitter single-phase approaches the improvement is moderate (up to 13 % higher throughput guarantees and up to 35 % lower end-to-end latencies compared to the results for the unmodified graph) as interference of task t_{CHEST} is still accounted for in all three subtasks. For the respective multi-phase approaches only the subtask t_{DEMAP} experiences maximal interference, which leads to a more significant improvement (up to 70 % higher throughput guarantees and up to 53 % lower end-to-end la-

tencies). The differences in results obtained from the execution interval analyses are not that large, which is due to the fact that the execution interval approaches are capable of excluding most interference. For the cases in which precedence constraints are combined with execution intervals (ei.c) even all interference on the cluster can be excluded, and that in both scenarios without and with source jitter, such that the respective single-phase and multi-phase results are both equal to each other and also equal to the execution interval results for the unmodified task graph in Table 12.1. Hence for this particular example the clustering does not result in an accuracy improvement if the most accurate execution interval approaches are applied.

Finally, we show that the different use-cases of multi-phase modeling depicted in Figure 12.11 can be also combined with each other. For that matter we consider the combination of task phases and clustering, such that we have the same three-phase tasks as in case (a), except for the clustered task whose combination with task phases is depicted in Figure 12.11(d). The results obtained for this combination are shown in Table 12.5.

P_{SRC}	23	17	10	9	8	44	41	38	30	17	16	15	12
J_{SRC}			0			88	82	76	60	34	32	30	24
sp.p	44	cv	cv	cv	cv	176	cv	cv	cv	cv	cv	cv	cv
sp.p.c	34	34	cv	cv	cv	146	140	134	cv	cv	cv	cv	cv
mp.p	18	18	20	cv	cv	118	112	106	90	66	cv	cv	cv
mp.p.c	18	18	18	18	cv	114	108	102	86	60	58	60	cv
sp.e	12	12	12	12	cv	158	154	cv	cv	cv	cv	cv	cv
sp.e.c	12	12	12	12	16	124	118	112	100	cv	cv	cv	cv
mp.e	12	12	12	12	14	116	110	104	88	62	62	cv	cv
mp.e.c	12	12	12	12	14	108	102	96	80	54	52	54	48

Table 12.5: End-to-end latencies obtained for case (d) in Figure 12.11 (Legend: sp = single-phase analysis, mp = multi-phase analysis, p = period-and-jitter, e = execution intervals, c = consideration of cyclic dependencies / precedence constraints to limit interference, cv = constraint violation).

If we compare the analysis results in Tables 12.2 and 12.4 for the period-and-jitter single phase approaches (cases sp.p and sp.p.c) we find that the clustering helps to reduce the negative effects of the task phases on this type of analysis (up to 30% higher throughput guarantees and up to 27% lower end-to-end latencies for the scenario without source jitter and up to 60% higher throughput guarantees for the jittered source). In contrast, we do not see any improvement for the single-phase execution interval approaches (sp.e and sp.e.c), which was expectable since we also did not see an improvement for the clustering without the combination with task phases.

Considering the results obtained with our multi-phase analyses for the combination of task phases and clustering we can observe that both period-and-jitter

and execution interval approaches are capable of maintaining the same analysis accuracy as for the case with clustering only. This shows that with our multi-phase approaches we can combine the different phase modeling use-cases freely without any degradation in analysis performance.

Altogether we can hence conclude that the introduction of our maximum response time computations considering multiple task phases results in a significant accuracy improvement compared to state-of-the-art for practically relevant use-cases, and that for all considered interference characterizations. Finally, note that all analysis run-times were in the microsecond range on a standard PC, which shows the applicability of our approaches for a quick verification of temporal constraints.

12.7.2 MULTI-RESOURCE ANALYSIS

Up to this point we have only considered processor sharing in our analyses of the WLAN transceiver application, whereas other types of resource sharing were not explicitly modeled. Specifically, the exchange of data between the different tasks was assumed only implicitly, such that all communication times had to be included within the respective execution times of tasks. In the following we change this paradigm, by taking into account explicitly that all tasks exchange data over a single shared memory.

We consider the task phase model from the previous section, such that all tasks in Figure 12.10 are split into three phases, as exemplified in Figure 12.11(a). In this scenario only the first and the last phases of all tasks access the shared memory, while the middle phases are executed processor-locally, i.e. without accessing other resources than the respective processors. In the last section we have assumed that these read and write phases have WCETs of $0.5\mu\text{s}$, and that including any possible interference on the shared memory.

Now let us model this interference explicitly. We assume that the read and write phases take $0.1\mu\text{s}$ instead of $0.5\mu\text{s}$ if they access the shared memory in isolation, i.e. without having any interference on the memory. Furthermore, we consider an SPP scheduling on the shared memory, with the priorities decreasing with the flow of data. This means that the read and write phases of task t_{FILTER} have the highest priorities on the memory, whereas the read and write phases of task t_{CHEST} have the lowest. The task-to-processor mapping and the processor priorities remain the same as depicted in Figure 12.10. Lastly, we assume that only 50% of the read and write phase execution times of $0.1\mu\text{s}$ are spent on memory accesses, such that only $0.05\mu\text{s}$ need to be considered as interference on the shared resource.

Recall that we can only consider execution interval approaches for multi-resource applications because the period-and-jitter ones are inapplicable. On the resulting multi-resource WLAN transceiver application we consequently apply the single-phase execution interval analysis technique discussed in the previous chapter (sp.e), the multi-phase execution interval analysis approach presented in

this chapter (mp.e), as well as our integrated multiple shared resource analysis technique introduced in Section 12.6 (msr.e). Moreover, we distinguish between analyses without and with the consideration of precedence constraints to limit interference (e and e.c, respectively) and deem the reduction of read and write phase WCETs to 50% on the shared memory optional (r), for a better comparability of analysis results. The minimal source periods and end-to-end latencies determined with these approaches are presented in Table 12.6 (all values in μs and rounded up to three digits).

P_{SRC}	13	12	8	7	148	108	73	55	24	19	14	10
J_{SRC}			0		296	216	146	110	48	38	28	20
sp.e	8.4	cv	cv	cv	469	cv	cv	cv	cv	cv	cv	cv
sp.e.c	7.2	7.2	cv	cv	424	356	cv	cv	cv	cv	cv	cv
mp.e	7.8	cv	cv	cv	388	308	244	cv	cv	cv	cv	cv
mp.e.c	7.2	7.2	cv	cv	372	292	222	188	cv	cv	cv	cv
msr.e	7.8	7.8	7.8	cv	331	251	181	145	87.1	cv	cv	cv
msr.e.c	7.2	7.2	7.2	7.4	317	237	167	131	68.7	58.7	52	cv
msr.e.r	7.8	7.8	7.8	cv	325	245	175	139	77.3	71.4	cv	cv
msr.e.c.r	7.2	7.2	7.2	7.3	312	232	162	126	63.7	53.7	43.7	39

Table 12.6: End-to-end latencies obtained for case (a) in Figure 12.11 with explicit resource sharing between first and last phases of tasks (Legend: sp = single-phase analysis, mp = multi-phase analysis, msr = multiple shared resource analysis, e = execution intervals, c = consideration of precedence constraints to limit interference, r = reduced WCETs on shared resource, cv = constraint violation).

The most apparent conclusion we can draw is that the analysis approaches optimized for multiple shared resources (msr) perform much better than the unoptimized approaches (sp and mp), even if the WCET reduction is not considered (up to 41% higher throughput guarantees for the scenario without source jitter and up to 75% higher throughput guarantees for the scenario with source jitter). This difference occurs because the approaches not optimized for multi-resource applications cannot differ between different phases having different higher priority phases. For instance, the multi-phase approaches consider shared memory interference of all three phases of task t_{FFT} on all three phases of task t_{CHEST} , whereas the multi-resource approaches only consider interference of the first and last phases of task t_{FFT} on the first and last phases of task t_{CHEST} .

If the WCET reduction is additionally taken into account (r) we do not see higher throughput guarantees for the source without jitter, but a slightly better end-to-end latency. For the jittered source case the impact of the WCET reduction is larger, such that the respective approaches compare even more favorably to the unoptimized approaches (up to 82% higher throughput guarantees).

Finally, note that even if the obvious improvement in analysis accuracy is not taken into account (compare e.g. the results for msr.e.c.r in Table 12.6 to the

respective results obtained with e.c in Table 12.1), an explicit consideration of multiple shared resources can be still beneficial for the analysis of real-time systems. This is because it is much more difficult to determine tight and nevertheless conservative WCETs if the respective tasks or task phases cannot be assumed to execute in isolation, but need to implicitly include interference of e.g. shared memories or buses.

This allows us to conclude that our lastly discussed analysis approach is capable of determining accurate analysis results, despite the explicit consideration of multiple shared resources, and that for practically relevant use-cases.

chapter 13

Simulation

In this chapter we present HAPI, an event-driven simulation approach. Of the use-cases described in Section 8.1.8 we focus on the application of simulation on application models as defined in Chapter 9 in the following, which enables to evaluate the correctness of real-time analysis results. The correctness evaluation is enabled by formally guaranteeing that analytical results are pessimistic compared to simulation results. The approach is theoretically capable of simulating any scheduler type, albeit the current implementation of the simulator only supports SPP scheduling, TDM scheduling, as well as cooperative RR scheduling.

The HAPI simulation approach for an evaluation of analysis result correctness is presented in Section 13.1, while the internals of the simulator are discussed in Section 13.2. Section 13.3 concludes this chapter with a case study.

13.1 SIMULATION & ABSTRACTION

In the preceding chapters we have discussed that temporal analysis techniques using dataflow models are implicitly or explicitly based on bounding abstraction theories such as the one described in the first part of this thesis. However, demonstrating that a component A bounds a component A' is regularly done by means of case distinction [WBS07b, SBW09, SB09b, LMC12], which can be non-trivial due to the large number of cases that must be distinguished, as well as due to a potential lack of formal methods guaranteeing that indeed all cases are identified. Therefore it is desirable to simulate the models used at lower levels of abstraction in order to gain confidence that the abstract analysis models and the analysis results are indeed correct. For instance, for a worst-case bounding analysis model, the confidence in the model correctness and the analytically obtained analysis results can be increased if none of the production moments found during simulation are later than the ones computed with a worst-case analysis model at design time. In contrast, for the case that a production moment found

Figure 13.1: The HAPI simulation approach.

during simulation is later than the moment computed by analysis we would like to conclude that the analysis is incorrect.

However, such a conclusion cannot be drawn in general. The reason is that the model used to compute analytical results is not necessarily an abstraction of the application model used in the simulator. More precisely, it is not guaranteed that a dataflow analysis model is a worst-case bounding abstraction of the simulation model, as indicated by the dashed arrow in Figure 13.1. Consequently, simulation results can be more pessimistic than worst-case analysis results, which makes it impossible to draw conclusions by comparing timestamps in data streams obtained via simulation on the one hand and analysis on the other.

In contrast, analytically obtained results can be verified with the HAPI simulator because the HAPI simulator directly works on application models as presented in Chapter 9. As our analysis models are based on the same application models, it holds by construction that e.g. dataflow models used for the worst-case analysis are worst-case bounding abstractions of the models used by the HAPI simulator.

A simulation with the HAPI simulator produces traces that are valid traces in the application model, as shown in Figure 13.1. The application model G^A , which is itself both an exclusion and inclusion abstraction of reality (see Chapter 9), is worst-case bounded by a worst-case dataflow model G^{WC} as used in the analysis approaches in the previous chapters. HAPI produces traces that are valid traces in the application model. Taking these simulation traces together results in a subset of all traces that can occur in the application model S , such that we can conclude that the application model is itself an inclusion abstraction of the traces determined by the simulator, i.e. $S \subseteq G^A$. Because the application model is worst-case bounded by the worst-case data dependency analysis model it follows according to Figure 13.1 that the same holds for the simulation traces as well, i.e. the simulation traces are worst-case bounded by the worst-case data dependency model, $S \leq G^{WC}$. From this we can conclude that if $S \leq G^{WC}$ does not hold that also $G^A \leq G^{WC}$ cannot hold. Putting it differently, HAPI simulation results can be used to falsify analytical results.

Note that the reverse of this conclusion usually does not apply, i.e. we cannot conclude from $S \leq G^{WC}$ that also $G^A \leq G^{WC}$. This means that we can falsify, but

cannot verify analysis results. This is because for verification we would have to conduct an exhaustive simulation in which all traces of an application model are determined, such that $S = G^A$, which is usually impractical and often impossible. Finally, also note that the simulation traces are valid traces in the application model, but it is not guaranteed that the determined traces are also “*real*” traces, i.e. traces that can also occur in reality.

13.2 THE HAPI SIMULATOR

The HAPI application model simulator is created using the SystemC library. The SystemC library provides an event-driven simulation kernel that processes events in the order they are generated by components. While the kernel does not actually perform preemptive scheduling of components, it can be used to simulate preemptive scheduling nevertheless. We discuss in this section that simulating preemptive scheduling does not require the so-called preemption points as mentioned in the related work section. Therefore, the HAPI simulator does not introduce an unpredictable amount of additional delay, which enables to use HAPI to directly simulate application models, thereby allowing for the aforementioned falsification of analytical results.

On top of the SystemC library we define several elements such as tasks and FIFO buffers for the expression of application models according to Chapter 9. The data-dependent behavior of these components follows the semantics of dataflow models as described in Section 4.7. This means that task components are enabled as soon as sufficient inputs are available in their input queues and as soon as previous executions have finished, that tasks immediately begin their executions after their enabling (data-driven behavior) and that outputs are produced in their output queues as soon as task executions finish. The differences between enabling and finish times are determined by both the execution times the tasks can have in isolation, which can vary between BCETs and WCETs, as well as interference that occurs due to preemption of other tasks sharing the same processor.

Using the Application Programming Interface (API) of HAPI, applications consisting of one or more task graphs can be defined in C++. In addition to tasks and FIFO buffers that model the data-dependent behavior of applications, shared processors can be specified, as well as scheduling policies and additional scheduling parameters such as priorities for the case of SPP scheduling. Altogether, this enables the expression of application models with both data and resource dependencies according to Chapter 9.

With a g++ compiler executables are created from the C++ programs in which the application models are specified. Running such executables results in VCD files in which traces with events are stored. These traces can be viewed with a VCD viewer such as GTKWave [Jen]. In the traces it is visible when tasks are enabled, executed, preempted and finished.

The HAPI simulator additionally supports so-called auto-concurrent execution of tasks during which multiple instances of the same task can execute at the

same moment in time. HAPI also supports an input-dependent execution as it is allowed for tasks to check how many inputs are in their incoming queues. With this feature so-called non-sequential firing rules can be defined that can result in schedule-dependent execution behaviors [LP95].

SystemC components generate events by executing the *notify()* SystemC call. Simulation time can only be advanced by executing the *wait(Interval, EventList)* SystemC call. This call does not return until *Interval* simulation time has passed or one of the events in *EventList* has occurred.

Preemptive scheduling of tasks on a processor can be simulated in SystemC by making use of *wait* and *notify* calls. In Figure 13.2 two components are depicted of which one component models a TDM scheduler and the other the execution of a task. Both components exchange start and stop events, which creates a kind of handshaking protocol between the components. The TDM scheduler component performs time-slicing and keeps track of the remaining budget of the task in the current replenishment interval of the scheduler. The *ExecuteTask* component models the execution of a task and handles events from the scheduler indicating that execution of the task should be stopped in order to model the effects of preemption.

The interaction between the *ExecuteTask* component and the *TDMscheduler* component is as follows. The task waits until it can acquire the required input data from its input queues. Thereafter it reads the input data and notifies the *TDMscheduler* component that it is due to execute by sending the *startTaskE* event. If there is a remaining budget for the task the scheduler notifies the task by sending it the *startSchedE* event, which signals that the task can advance time until either the budget is depleted or the task execution is completed. A depleted budget is indicated by the scheduler with the *stopSchedE* event. That the execution of the task has finished is indicated by the *stopTaskE* event issued by the *ExecuteTask* component. If there is remaining execution time then the inner while loop in the *ExecuteTask* component is repeated a number of times. If a task has depleted its budget then it receives a new budget in the next replenishment interval of the TDM scheduler. When the remaining execution time is zero, a computation is performed on the input data and the results are written to the output queues. Thereafter this output data is released, resulting in a notification for the consuming tasks. These operations do not advance the simulation time.

Other types of scheduling policies can be implemented without changing the *ExecuteTask* component, as shown in Figure 13.3.

In the SPP scheduling component *SPPscheduler* all tasks are tested in a descending priority order whether they are enabled. If a task is enabled it is notified with the *startSchedE* event that it can execute until a higher priority task gets enabled or the execution of the task is completed. After that the outer while loop is repeated and it is tested again for the highest priority task that is enabled.

In the non-preemptive RR scheduling component *RRscheduler* the tasks are executed in a predefined order. The scheduler component sends the *startSchedE* event to indicate that the task can execute and waits until the *stopTaskE* event before the next task is allowed to execute.


```

1 ExecuteTask () {
2   while (true) {
3     task.acquire ();
4     task.readData ();
5     set (requested);
6     notify (startTaskE);
7     remainingET=task.ET ();
8     do {
9       wait (startSchedE);
10      startT=currT ();
11      wait (remainingET, stopSchedE);
12      remainingET -= (currT () - startT);
13    } while (remainingET > 0);
14    clear (requested);
15    notify (stopTaskE);
16    task.processData ();
17    task.writeData ();
18    task.release ();
19  }
20 }

1 TDMscheduler () {
2   while (true) {
3     startR=currT ();
4     foreach (task) {
5       B=task.getbudget ();
6       while (B > 0) {
7         startT=currT ();
8         if (task.requested ()) {
9           notify (startSchedE);
10          wait (B, stopTaskE);
11        } else {
12          wait (B, startTaskE);
13        }
14        B=currT () - startT;
15      }
16      notify (stopSchedE);
17    }
18    D=currT () - startR;
19    wait (proc.RI () - D);
20  }
21 }

```

Figure 13.2: TDM scheduling in HAPI.

```

1 SPPscheduler () {
2   while (true) {
3     sortPriority (taskList);
4     foreach (task) {
5       if (task.requested ()) {
6         notify (startSchedE);
7         Elist=task.HPeventList ();
8         Elist=Elist OR stopTaskE;
9         wait (Elist);
10        notify (stopSchedE);
11        break;
12      }
13    }
14  }
15 }

1 RRscheduler () {
2   while (true) {
3     foreach (task) {
4       if (task.requested ()) {
5         notify (startSchedE);
6         wait (stopTaskE);
7       }
8     }
9   }
10 }

```

Figure 13.3: SPP and RR scheduling in HAPI.

13.3 CASE STUDY

In this section we present a number of small didactic examples, as well as our well-known WLAN 802.11p transceiver application in order to illustrate the usability and capabilities of the HAPI simulator.¹

The first example is the HSDF model shown in Figure 13.4 which models a producer and consumer task. The actor v_a models the producer and v_b the consumer. The actors have firing durations of 2ns and 3ns, respectively. Self-edges

¹The HAPI simulator together with the C++ code of the examples in this thesis are made available under the GPL licence.

with a single token prevent auto-concurrent execution, which models that the respective tasks cannot start their executions before their previous executions have finished. The FIFO communication buffer between the tasks is modeled by two unbounded queues. The buffer is assumed to be initially empty and its capacity of three locations is modeled by three initial tokens on the edge $e_{b,a}$.

Figure 13.4: Producer-consumer example.

Both actors execute concurrently during self-timed execution in the HAPI simulator. This is also apparent in the simulation trace shown in Figure 13.5. Initially actor v_a can execute every 2ns because there are a sufficient number of tokens on the edge $e_{b,a}$. However, after 10ns its firings are delayed as it has to wait for productions by actor v_b .

Figure 13.5: Trace of producer-consumer example.

A second example is shown in Figure 13.6 in which actor v_b is forced to execute strictly periodically every period of 4ns. An error is reported in the case that the actor is not enabled before it is executed. It is not enabled if there is less than one token in the queue $e_{a,b}$. The actor v_a has a firing duration which varies randomly every execution between 1ns and 5ns. In this example it can occur that an error is reported because the firing duration of v_a can be larger than the period of v_b .

Figure 13.6: Dataflow graph with a periodic sink.

An example of a task graph in which task t_a is scheduled by a TDM scheduler is shown in Figure 13.7. The task t_a has an execution time χ_a of 3ns, a budget S_a of 9ns and a replenishment interval Q_a of 18ns. Task t_b is executed on a separate processor. The tasks are connected by a blocking write buffer with a capacity of one, which effectively makes executions of the tasks mutually exclusive. Whether task t_a can continue its execution depends on the availability of remaining budget in the current replenishment interval.

Figure 13.7: Task graph scheduled using TDM.

The simulation trace for the task graph in Figure 13.7 is shown in Figure 13.8. This figure shows that the processor becomes idle after task t_a has finished its execution at $\tau = 3\text{ns}$. In the interval between 3ns and 9ns , task t_a cannot execute, although it has budget left, because it is not enabled (indicated by “xxxx”). And in the interval between 9ns and 18ns , task t_a is enabled, but does not have budget left (indicated by “Xxxx”). Therefore it has to wait for a new budget in the second replenishment interval, which begins at $\tau = 18\text{ns}$, to continue its execution.

Figure 13.8: Trace of the TDM example.

The fourth example is the dataflow graph in Figure 13.9 which uses a latency-rate dataflow component that can accurately capture the worst-case behavior of TDM scheduling. Actor $v_{a,0}$ does not contain a self-edge and can therefore execute auto-concurrently. The parameters of the actors are the same as of the tasks in the previous example, while a FIFO buffer of capacity three instead of one is modeled by the edges connecting the actors, enabling concurrency.

Figure 13.9: Latency-rate model of a TDM-scheduled actor.

The trace generated by HAPI given the dataflow graph in Figure 13.9 is shown in Figure 13.10. In this figure we find separate traces for the latency and the rate actors that model the behavior of v_a . The counter of the latency actor $v_{a,0}$ is updated when its next firing starts. At $\tau = 0$ the counter value equals to two because firings 0, 1 and 2 happen concurrently as a result of the three initial tokens.

Figure 13.10: Trace for the latency-rate example.

The fifth example is the task graph shown in Figure 13.11. In this task graph the task t_a has a lower priority than task t_c and both tasks are executed on the same processor. Therefore, task t_c can preempt task t_a .

Figure 13.11: Task graph scheduled using SPP scheduling.

The simulation trace for this task graph is shown in Figure 13.12. The *proc* trace in this figure shows which task is executed on the processor at every point in time. From the trace it becomes apparent that t_a is preempted at $\tau = 9$ ns.

Figure 13.12: Trace of the SPP example.

Another example which makes use of SPP scheduling is shown in Figure 13.13. This example illustrates that HAPI is capable of considering tasks with multiple inputs. The tasks can only execute when there is at least one full location available in all their input buffers.

Figure 13.13: Tasks graph with a task that has multiple inputs.

In our last example we consider the WLAN 802.11p transceiver application that we have already used in the case studies of the previous three sections.

We assume again that the transceiver is executed on a multiprocessor system for performance reasons. The application consists of multiple modes, of which we only consider the packet decoding mode whose task graph is depicted in Figure 13.14.

Figure 13.14: Task graph of the packet decoder of a WLAN 802.11p transceiver.

Here we assume that symbols arrive strictly periodic with a period of $8\mu\text{s}$, which corresponds to a frequency of 125kHz . Received symbols are first passed through a filter and then processed by an FFT, an equalizer, a demapper, a deinterleaver and finally a viterbi decoder. Furthermore, stability is improved by compensation of the domain-specific channel variation. This is realized by adjusting the equalizer for the reception of symbol i by an estimate of the channel during the reception of symbol $i - 2$. Such a channel estimate is obtained by reencoding the error-corrected bits of symbol $i - 2$ and comparing them to the received symbol $i - 2$. This results in the depicted feedback loop, limiting the maximum throughput of the packet decoding mode.

The tasks of the packet decoding mode are executed on four different processors. If multiple tasks share one processor then an SPP scheduler is used. Figure 13.14 depicts one possible task-to-processor mapping. The tasks t_{FILTER} and t_{REENC} are executed on separate processors and consequently cannot get preempted. The other tasks, in contrast, use shared processors, which is indicated by tasks sharing a processor having the same color. The priorities of the tasks executed on shared processors are denoted inside the corresponding vertices, with π_1 being the lowest and π_4 the highest.

Denoted next to the tasks are their BCETs and WCETs. As data and resource dependencies are the same as in the realization, whereas BCETs and WCETs are under-approximated and over-approximated, respectively, it holds that the model depicted in Figure 13.14 represents an application model that is a valid inclusion (and exclusion) abstraction of reality according to the abstraction-refinement theory presented in the first part of this thesis.

Simulating the presented example using HAPI produces traces that can occur in the application model, of which one is depicted in Figure 13.15. As you can see the execution time of task t_{FILTER} varies between $1\mu\text{s}$ and $3\mu\text{s}$, resulting in different preemption patterns and consequently different end-to-end latencies (the times between executions of task t_{SRC} and finishes of task t_{VIT}). The maximum end-to-end latency determined using simulation equals to $14\mu\text{s}$. This latency can be for

instance observed in the sixth iteration in Figure 13.15, with task t_{FILTER} having an execution time of $3\mu\text{s}$ and task t_{DEINT} being preempted for $2\mu\text{s}$ by task t_{CHEST} in its fifth iteration.

Figure 13.15: Trace of the WLAN 802.11p example.

As discussed in Section 13.1, a model used for analysis must be an abstraction of the model used for simulation. The analysis techniques presented in the previous chapters are capable of using the same model as used in simulation. For this example temporal analysis determines the same maximum end-to-end latency as our HAPI simulation (see Table 11.1). This shows on the one hand that in this case analysis is so accurate that its result matches the one obtained with simulation. On the other hand this also implies that HAPI simulation cannot be only used to determine average latencies, but can also trigger the worst-case.

Figure 13.16: Trace of the WLAN 802.11p example with wrong priority of task t_{CHEST} .

Now consider that one makes a mistake when constructing the analysis model from the inclusion abstraction used for simulation. For instance assume that in the analysis model the priority of task t_{CHEST} is not set to the highest, but to the lowest on the shared processor. Then this faulty temporal analysis determines a maximum end-to-end latency of only $12\mu\text{s}$ instead of $14\mu\text{s}$. The reason for this difference can be observed in Figure 13.16, which depicts a simulation trace with the same mistake: Now task t_{CHEST} in its fifth iteration cannot preempt any of the

tasks in iteration six, but is itself preempted. As task t_{CHEST} is not on the critical path a smaller end-to-end latency is determined.

The HAPI simulation using the correct model determines an end-to-end latency of $14\mu\text{s}$, whereas the faulty analysis determines an end-to-end latency of only $12\mu\text{s}$. According to Figure 13.1 the higher simulated latency allows us to conclude that the analysis must be indeed faulty. This demonstrates that HAPI can be used for the falsification of incorrectly obtained analysis results.

Conclusion

In the second part of this thesis we have exploited the theoretical foundation laid in the first for the evaluation of concurrent real-time systems with shared resources.

The difficulties of evaluating such applications arise mainly due to the allowed presence of cyclic data dependencies, which can occur due to both feedback loops and FIFO buffers with finite capacities, as well as due to the usage of SPP scheduling that requires enabling rate characterizations of tasks to limit interference and that can introduce scheduling anomalies, such that earlier events can result in overall worse temporal properties.

At first, we have shown that concurrent real-time systems with shared resources can be accurately captured as application models based on the timed component model from the first part. From such application models we have further derived several analysis models by means of inclusion and bounding abstraction, which demonstrates that the methods from the first part can be indeed used to establish strong relations between application and analysis models. Besides, these relations also allowed us to prove thereafter that the results produced by the techniques applied on analysis models are temporally conservative to the actual events that can occur in concurrent real-time systems.

After the presentation of analysis models we have devised several analysis techniques for single-rate and single-phase applications that were applied on the analysis models in the contexts of iterative analysis flows. We have shown that analysis accuracy of a period-and-jitter analysis for applications with SPP scheduling can be substantially improved by the additional consideration of cyclic data dependencies between tasks to limit interference. The replacement of period-and-jitter with execution intervals led to a further significant improvement in analysis accuracy. And the introduction of an iterative buffer sizing for both buffers with blocking and non-blocking writes increased accuracy even further. Moreover, the usage of an iterative buffer sizing on blocking write buffers did not only result in a better accuracy, but actually optimized the application model

itself to meet a certain throughput constraint, resulting in both better temporal properties and smaller buffer capacities. Finally, the additional introduction of synchronization edges was able to amplify the positive effects of an iterative buffer sizing even more.

Based on the multi-phase analysis approach using execution intervals we have derived an analysis approach that can analyze real-time systems with multiple shared resources in a temporally conservative manner. This enables an overall more accurate analysis of applications with e.g. both shared processors and shared memories, as the communication times determined by the latter can be captured accurately instead of being necessarily hidden in the WCETs of tasks.

According to the terminology introduced in Section 2.2, our first two techniques based on period-and-jitter and execution intervals for single-phase applications improved state-of-the-art in terms of effectiveness, while the multi-phase and multi-resource analyses maintained the same level of effectiveness for the more entropic multi-phase and multi-resource analysis models. Hence all presented approaches led to significant improvements in analysis accuracy, which is also reflected by our case study results.

In the end of the second part we have also introduced a simulation approach for concurrent real-time systems. Unlike other simulation approaches, the HAPI simulator is able to determine simulation traces that can occur in the same application models also used for analysis. With the methods from the first part we were able to show that we can use HAPI to falsify analysis results that are the consequence of mistakes in the construction of analysis models.

Let us conclude the second part of this thesis with some conceptual remarks. First, we have shown that the abstraction techniques from the first part can be used to prove that analysis results are not only valid within analysis models, but also for the underlying application models, even if the latter are non-deterministic, non-monotone and even uncertain in some cases. Second, all the presented analysis techniques demonstrate that it can be advantageous to not only construct a single analysis model by means of abstraction and thereafter apply a single analysis technique, but to devise multiple models and to combine different techniques on these models in iterative procedures instead. In fact, for the SPP scheduling it is even required to use such procedures, due to the interdependence between response times and schedules of tasks. And third, the introduction of the iterative buffer sizing for blocking write buffers indicates that such iterative approaches do not need to be limited to different analysis models, but can also include the underlying application models. In that case the analysis approaches in fact become optimization approaches in which the modeled real-time systems are actively modified to e.g. meet a certain throughput constraint.

Finally, note that a maximally accurate analysis of many real-time systems is inherently undecidable [2], even after application of exclusion abstraction to remove unconsidered corner cases. Hence our ultimate goal also was not to determine the most complex and most accurate analysis approach for all cases, but to devise several methods that allow to seamlessly trade off accuracy and

complexity for a broad class of applications instead. The focus was efficiency, i.e. achieving high accuracies at acceptable run-times.

Future work with respect to the evaluation of concurrent real-time systems could include the consideration of other scheduler types such as RR or TDM, as well as a consideration of both iterative buffer sizings and synchronization edges in the contexts of our multi-phase approaches. Finally, from a more conceptual point of view, especially the combination of different analysis techniques on different abstraction levels still appears to have lots of potential to determine different trade-offs between accuracy and complexity. In particular, combinations of analytical approaches as the ones described in this thesis with model checking approaches, as already hinted at in [2], appear to be promising candidates for further substantial improvements.

chapter 15

Postface

In the very beginning of this thesis we have developed that ever since the development of thought we have lived in two worlds. For one, there is the world of objects, the world of real things that we can directly perceive and interact with. And then there is the world of imagination, the world of models that we use to understand and educatedly influence the reality surrounding us.

The most prominent characteristic of the world of objects is its unimaginable complexity. It is not unlikely that the complexity of reality is infinite, and that both macro- and microscopically, both in time and in space. Everything appears possible but nothing certain and everything seems to depend on everything else. In contrast, we can construct the world of imagination in just the way we prefer. As such, our imagination is usually nice and clean (anything else would be indeed rather masochistic). The number of possibilities is limited, certainty prevails and dependencies are manageable.

Harari argues that the world of imagination “*became ever more powerful*”, that the impact of our imaginations on reality is ever-increasing [Har15]. With the rise of imagination we can also observe the rise of two different schools of thought: A mainly theoretical school that is inclined to the world of imagination and a mainly practical school that tends to the world of objects.

All too often, unfortunately, the two schools become more and more disentangled, even if they roughly apply to the same field. For one, the theorist often remains in her *heaven of models* in which many circumstances are assumed as ideal, abstracting from the difficulties and uncertainties that are prevalent in reality. This enables the theorist to derive strong statements like guarantees, whose relevance for reality is, however, fairly limited. And the practitioner, frustrated with the inapplicability of the theorists’ models on the increasingly complex *hell of implementations* that she is exposed to, is forced to remain as close as possible to reality, which is not rarely manifested in the use of estimates, rules-of-thumb and even plain guesswork.

In light of these considerations we can regard this thesis as an attempt to remarry the two schools of thought once more. Following the principle that “*there is nothing so practical as a good theory*” [Lew51] we present an abstraction-refinement theory that enables the application of strong and simple models on the difficult and complex, real implementations. To this end we do not need to travel the many circles described in Dante’s Divine Comedy to get from the hell of implementations to the heaven of models, but only require three bridges instead: Exclusion, inclusion and bounding.

We have discussed the applicability of our theory for concurrent real-time systems with shared resources in great detail. Besides, we have proposed an extension for CPS’s whose practical relevance remains to be shown in future work. All the more, we believe that the concepts of exclusion, inclusion and bounding abstraction and refinement are so fundamental that they are not limited to these particular use-cases, but can be applied in a wide range of other fields of study as well.

PART III

Appendix

An Abstraction-Refinement Theory for Cyber-Physical Systems

In this appendix we generalize the component model introduced in Chapter 4 and the abstraction-refinement theory in Chapter 5 such that it is not only applicable for real-time systems (discrete-event systems), but also for CPS's (mixed discrete-event and continuous time systems). For that matter we first generalize the concept of streams to so-called signals, on which grounds we redefine the entire component model. Thereafter, we discuss a generalized abstraction-refinement theory for this model, which allows for other kinds of abstractions and refinements than only inclusion, exclusion and bounding.

A.1 INFORMAL DESCRIPTION

Our timed component model as well as all abstraction-refinement theories that we discussed so far are only applicable for discrete-event systems. This is mainly due to the fact that the base units of our theory are streams consisting of infinite sequences of discrete timestamps and values. In an attempt to make our theory applicable for CPS's, we replace our notion of streams with signals in which times and values are expressed as continuous functions. For such signals, we develop a timed component model equivalent to the one for streams.

Thereafter, we generalize our inclusion, exclusion and bounding abstraction-refinement theories to an approximation abstraction-refinement theory.

Recall that a component includes another component if it produces the same outputs for the same inputs. This means that the including component must have at least the same behaviors as the included component, as depicted with the left-hand relation in Figure A.1. Likewise, a component best-case (worst-case) bounds another component if it produces lower-bounding (upper-bounding) outputs for lower-bounding (upper-bounding) inputs. Hence the bounding component must

lower-bound (upper-bound) all behaviors of the bounded component, as it is shown in the middle of Figure A.1.

Figure A.1: Inclusion, bounding and approximation.

In a similar fashion, we define that a component approximates another component if it produces approximating outputs for approximating inputs. The approximation relation between the respective inputs and outputs can be any relation on signals that is both reflexive and transitive. Therefore it holds that inclusion and bounding are both approximation relations, while we can also use approximation to express more exotic relations expedient in the CPS domain, as illustrated on the right-hand side of Figure A.1. Moreover, we do not even require the same approximation relations for inputs and outputs, such that we can also express approximating components as components that e.g. produce bounding outputs for including inputs.

Just like we did for exclusion, inclusion and bounding we first specify approximation for signals, thereafter for signal traces, then for components and finally prove the automatic lifting of approximation from component to graph level. This finally enables to define approximation abstraction and refinement.

A.2 SIGNAL COMPONENT MODEL

This section first introduces the notion of signals as a generalization of streams. Thereafter, traces are redefined such that they do not contain multiple streams anymore, but multiple signals. Based on this redefinition the component model in Chapter 4 becomes a component model that considers components transferring signals instead of streams, such that CPS's become expressible. Lastly, necessary corrections of the definitions in Section 4.6 are presented, which transfer the concepts of input acceptance lifting and replaceability from streams to signals.

A.2.1 PORTS & SIGNALS

We define signals as infinite, continuous functions of events, with each event mapping an index to a timestamp and a value. Analogous to streams, we define the length of signals, as well as signal ports that are used to transfer signals from a specific value domain. Based on that, we establish the relation between streams

and signals. Finally we specify the connectibility of signal ports and define the prefix and earlier-than relations for signals on the same ports.

Definition 57 (Signal). *A signal x on a port p is an infinite, continuous function of indexed events, with each event consisting of the production time of the event in the form of a timestamp and a value from the value domain of the port. Formally x can be described as a total mapping $x : \mathbb{R}_0^+ \rightarrow \mathcal{T} \times O$, with $\mathcal{T}$ a continuous time domain and O a value domain. We require that the time domain $\mathcal{T}$ is a lattice with respect to an ordering relation $\leq$ and that $\mathcal{T}$ has an infimum $0 \in \mathcal{T}$, as well as a supremum $\infty \in \mathcal{T}$, such that $\forall_{\tau \in \mathcal{T}} : 0 \leq \tau \leq \infty$. Analogously we require that the value domain O is a lattice with respect to an ordering relation $\models$ and that O has an infimum $\vartheta^0 \in O$, as well as a supremum $\vartheta^\infty \in O$, such that $\forall_{\vartheta \in O} : \vartheta^0 \models \vartheta \models \vartheta^\infty$. We use $\tau_x : \mathbb{R}_0^+ \rightarrow \mathcal{T}$ and $\vartheta_x : \mathbb{R}_0^+ \rightarrow O$ to retrieve timestamps and values of events by their indices, respectively.*

Although signals are formally defined as infinite functions of events, signals can also be seen as finite. We define the length of signals as follows:

Definition 58 (Signal Length). *We define an event of a signal x at index i to be absent iff $\tau_x(i) = \infty$ and $\vartheta_x(i) = \vartheta^\infty$. The length of a signal can then be defined as the smallest index from which onwards all events are absent, i.e. (with $\inf \emptyset \equiv \infty$):*

$$|x| = \inf \{i \in \mathbb{R}_0^+ \mid \forall_{i' \geq i} : \tau_x(i') = \infty \wedge \vartheta_x(i') = \vartheta^\infty\}$$

We transfer signals over so-called signal ports, which are specified as follows:

Definition 59 (Signal Port). *A signal port p is characterized by a tuple $(x, O_p, \models_p)$ and contains a signal $x \in St(p)$, with $St(p)$ the set of all valid signals on port p . The set $St(p)$ is constructed based on a port-specific value domain O_p , such that $St(p) = \{x \mid x : \mathbb{R}_0^+ \rightarrow \mathcal{T} \times O_p\}$. The port-specific value domain O_p must adhere to the requirements on the value domains of signals, i.e. it must be a lattice with respect to an ordering relation $\models_p$, with ϑ_p^0 the infimum and ϑ_p^∞ the supremum. Based on the ordering relations $\leq$ of $\mathcal{T}$ and $\models_p$ of O_p we further define the null signal $x_p^0 \in St(p)$, such that $\forall_{i \in \mathbb{R}_0^+} : \tau_{x_p^0}(i) = 0 \wedge \vartheta_{x_p^0}(i) = \vartheta_p^0$, and the empty signal $x_p^\infty \in St(p)$, such that $\forall_{i \in \mathbb{R}_0^+} : \tau_{x_p^\infty}(i) = \infty \wedge \vartheta_{x_p^\infty}(i) = \vartheta_p^\infty$, respectively.*

Given these definitions, we can formulate the representation of streams as signals. This allows to mix streams and signals in the timed component model, enabling the creation of CPS models.

Definition 60 (Stream Signal). *Let $x \in St(p)$ be a stream on a port p that is characterized by a tuple $(x, O_p, \models_p)$. Such a stream can be represented as a signal $x' \in St(p')$ on a port p' that is characterized by a tuple $(x', O_{p'}, \models_{p'})$ with:*

$$\forall_{i \in \mathbb{R}_0^+} : \tau_{x'}(i) = \tau_x(\lfloor i \rfloor) \wedge \vartheta_{x'}(i) = \vartheta_x(\lfloor i \rfloor)$$

Analogous to the connectibility of stream ports, we define the connectibility of signal ports as follows:

Definition 61 (Signal Port Connectivity). *Let q and p be two different signal ports, i.e. $q \neq p$. Then port p is connectible to a port q , i.e. $q \rightarrow p$, iff it holds that all valid signals on port q are also valid signals on port p , i.e. $St(q) \subseteq St(p)$.*

The prefix and earlier-than ordering relations for signals on the same port are defined as follows:

Definition 62 (Signal Order). *Let $x, x' \in St(p)$ be two signals on a port p . The prefix ordering relation $\leq$ and the smaller-than ordering relation $\leq_f$ for signals are defined as:*

$$\begin{aligned} x \leq x' &\equiv |x| \leq |x'| \wedge \forall_{i < |x|} : \tau_x(i) = \tau_{x'}(i) \wedge \vartheta_x(i) = \vartheta_{x'}(i) \\ x \leq_f x' &\equiv |x| = |x'| \wedge \forall_{i < |x|} : \tau_x(i) \leq \tau_{x'}(i) \wedge \vartheta_x(i) \models_p \vartheta_{x'}(i) \end{aligned}$$

For the feedback composition of components using signals we need to define limits of sequences of signals. For that matter we require a distance function for signals that converges towards zero if the distances between all timestamps and values approach zero.

First, we combine the timestamp and value distances to a distance between events of signals as follows:

Definition 63 (Signal Event Distance). *Let $x, x' \in St(p)$ be two signals on a port p . The distance between two events of such signals is a function $d_{St(p)} : St(p) \times St(p) \times \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ with:*

$$d_{St(p)}(x, x', i) = \max(d_T(\tau_x(i), \tau_{x'}(i)), d_O(\vartheta_x(i), \vartheta_{x'}(i)))$$

With the event distance we can define a suitable distance function $D_{St(p)}$ for signals on a port p adhering to the property that if two signals converge to each other, such that all timestamps and values converge to each other, the distance function approaches 0. Formally, this means that for two signals $x, x' \in St(p)$ on a port p the following property must hold:

$$\forall_i : d_{St(p)}(x, x', i) \rightarrow 0 \Rightarrow D_{St(p)}(x, x') \rightarrow 0$$

It can be seen that the following distance function satisfies above property:

Definition 64 (Signal Distance). *The distance between two signals $x, x' \in St(p)$ on a port p is a function $D_{St(p)} : St(p) \times St(p) \rightarrow \mathbb{R}_0^+$ with:*

$$D_{St(p)}(x, x') = \max_{i \in \mathbb{R}_0^+} \left(\frac{1}{i+1} \cdot (1 - 2^{-d_{St(p)}(x, x', i)}) \right)$$

Using this signal distance function we can define the limit of signal sequences as follows:

Definition 65 (Signal Sequence Limit). *Let $x_k \in St(p)$ be a sequence of signals and $x \in St(p)$ a signal on a port p . The sequence x_k converges to x for $k \rightarrow \infty$, i.e. $\lim_{k \rightarrow \infty} x_k = x$, if it holds that for all $\varepsilon > 0$ there exists a $K \in \mathbb{N}$ such that for all $k \geq K$ it holds that $D_{St(p)}(x_k, x) < \varepsilon$.*

Based on the definition of signals, we redefine the stream interfaces and stream traces from Chapter 4.2 as signal interfaces and signal traces as follows:

Definition 66 (Signal Interface). *A signal interface P is a set of $|P|$ different signal ports.*

Definition 67 (Signal Trace). *A signal trace X is a set of $|X| = |P|$ signals on a signal interface P , such that each signal $x \in X$ is on a different port $p \in P$. In the following we use the shorthand notation $X[p]$ to retrieve the signal on port $p \in P$. The set of all valid signal traces on an interface P is then defined as $Tr(P) = \{X \mid |X| = |P| \wedge \forall_{p \in P} : X[p] \in St(p)\}$, with $X_p^0 \in Tr(P)$ the null signal trace and $X_p^\infty \in Tr(P)$ the empty signal trace, such that $\forall_{p \in P} : X_p^0[p] = x_p^0 \wedge X_p^\infty[p] = x_p^\infty$.*

These definitions allow us to lift connectivity of signal ports to connectivity of signal interfaces:

Definition 68 (Signal Interface Connectivity). *Let Q and P be two disjoint signal interfaces of the same numbers of ports, i.e. $Q \cap P = \emptyset$ and $|Q| = |P|$. Furthermore let $\Xi : Q \rightarrow P$ be a bijective mapping, i.e. $\forall_{q \in Q} \exists_{p \in P} : p = \Xi(q)$ and $\forall_{q, q' \in Q} : q \neq q' \Rightarrow \Xi(q) \neq \Xi(q')$. Then signal interface P is connectible to signal interface Q given mapping Ξ , i.e. $Q \rightarrow_\Xi P$, iff it holds that all mapped signal ports are connectible, i.e. $\forall_{q \in Q} : q \rightarrow \Xi(q)$.*

Given connectivity of signal interfaces we can also define the connection of signal interfaces as follows:

Definition 69 (Signal Interface Connection). *Let Q and P be two disjoint signal interfaces of the same numbers of ports. Furthermore let $\Xi : Q \rightarrow P$ be a bijective mapping. If P is connectible to Q given mapping Ξ , i.e. $Q \rightarrow_\Xi P$, then the signal interfaces Q and P can be connected by a signal interface connection C^Ξ .*

A signal interface connection C^Ξ is characterized by a tuple (Q, P, Ξ) and assigns the signals on P to signals on Q , according to mapping Ξ . Formally C^Ξ can be described as a function $C^\Xi : Tr(Q) \rightarrow Tr(P)$ with $X = C^\Xi(Y) \equiv \forall_{q \in Q} : X[\Xi(q)] = Y[q]$.

We can also lift the prefix and earlier-than ordering relations of signals to prefix and earlier-than ordering relations of traces:

Definition 70 (Signal Trace Order). *Let $X, X' \in Tr(P)$ be two signal traces on an interface P . The prefix ordering relation $\leq$ and earlier-than ordering relation $\leq$ for signal traces are defined as:*

$$\begin{aligned} X \leq X' &\equiv \forall_{p \in P} : X[p] \leq X'[p] \\ X \leq X' &\equiv \forall_{p \in P} : X[p] \leq X'[p] \end{aligned}$$

Just like for stream traces, the limit of signal sequences can be lifted to a limit of trace sequences as follows:

Definition 71 (Signal Trace Sequence Limit). *Let $X_k \in \text{Tr}(P)$ be a sequence of signal traces and $X \in \text{Tr}(P)$ a signal trace on an interface P . The sequence X_k converges to X for $k \rightarrow \infty$, i.e. $\lim_{k \rightarrow \infty} X_k = X$, if all signals of the sequence X_k converge to the respective signals of X for $k \rightarrow \infty$, i.e. $\forall_{p \in P} : \lim_{k \rightarrow \infty} X_k[p] = X[p]$.*

The merging and splitting of signal traces is moreover defined as follows:

Definition 72 (Signal Trace Merging & Splitting). *Let X_0 be a signal trace on an interface P_0 and X_1 a trace on an interface P_1 . If $P_0 \cap P_1 = \emptyset$ then these traces can be merged to a signal trace $X = X_0 \cup X_1$ on an interface $P = P_0 \cup P_1$. In the following we call the reverse of signal trace merging signal trace splitting and signal traces such as $X_0 \subseteq X$ and $X_1 \subseteq X$ subtraces of a signal trace X .*

Finally, we have to adjust the notions of partial trace sets and compositional trace relations such that they apply for signals as follows:

Definition 73 (Partial Signal Trace Sets). *Let $\text{Tr} \subseteq \text{Tr}(P)$ be a set of traces on an interface P . We denote the set of signals on a port $p \in P$ that are part of traces in Tr as:*

$$\text{Tr}[p] = \{x \in \text{St}(p) \mid \exists_{X \in \text{Tr}} : x = X[p]\}$$

Likewise, we denote the set of traces on an interface $P' \subseteq P$ that are part of traces in Tr as:

$$\text{Tr}[P'] = \{X' \in \text{Tr}(P') \mid \exists_{X \in \text{Tr}} : \forall_{p \in P'} : X'[p] = X[p]\}$$

Definition 74 (Compositional Signal Trace Relation). *A relation $\propto$ on traces is compositional iff it is also defined for signals, such that it holds on any interface P that $X \propto X' \Leftrightarrow \forall_{p \in P} : X[p] \propto X'[p]$.*

In the following we assume that all definitions in Sections 4.3 to 4.6 do not apply with respect to the definitions of stream traces and interfaces from Chapter A.2.2, but with respect to the definitions of signal traces and interfaces from this chapter. In particular, this implies that components and graphs of components that relate signals are now defined in just the same way as components and component graphs that relate streams, enabling an expression of CPS's in our component model. Moreover, it can easily be seen that all proofs in Section 4.6 also hold for signal traces, enabling an automatic lifting of input acceptance and replaceability for our signal component model.

A.3 GENERALIZED ABSTRACTION & REFINEMENT

Being an attentive and thorough reader, you may have noticed that Sections 5.2 and 5.1 are very similar in structure, definitions, lemmas, theorems and even proofs, and that even though exclusion, inclusion and bounding describe fundamentally different concepts. The reason for this similarity is that the functionality of exclusion, inclusion and bounding abstraction and refinement can be traced back to two properties that both used trace relations, trace equality and trace bounding, have in common: Reflexivity and transitivity.

For CPS's exclusion and inclusion abstraction and refinement remain viable concepts. However, in many cases it is desirable to do different kinds of abstractions than exclusion, inclusion, best-case or worst-case bounding, which, with respect to time, denote the same, earlier or later traces. Moreover, in the context of CPS's that mix both digital and analog parts, it may be required to combine different relations between traces.

In this section we generalize the concepts introduced in Chapter 5 such that they become applicable for components that approximate each other with respect to different trace relations. First, we introduce the notion of generic signal and trace approximation relations, which have reflexivity and transitivity as their sole requirements. Then we define component approximation for such relations and subsequently prove lifting of approximation from component to graph level, as well as abstraction and refinement.

A.3.1 SIGNAL & TRACE APPROXIMATION

We define a generic signal approximation relation as follows:

Definition 75 (Signal Approximation). *We call a relation α_p an approximative signal relation on a port p iff it holds that the relation is reflexive and transitive for all signals in $St(p)$, i.e. $\forall x \in St(p): x \alpha_p x$ and $\forall x, x', x'' \in St(p): x \alpha_p x' \alpha_p x'' \Rightarrow x \alpha_p x''$. We say that x under-approximates x' with respect to α_p iff $x \alpha_p x'$.*

In the following we relate traces, components and graph of components to each other whose signals approximate each other with respect to different such approximative signal relations. To ease the notation of different relations we define a so-called pool of relations as follows:

Definition 76 (Pool of Approximation Relations). *A pool of approximation relations $\mathcal{A}$ on an interface P is defined as a set of approximative signal relations α_p , such that for each port of the interface exactly one relation exists in the set, i.e. $\forall p \in P \exists! \alpha_p \in \mathcal{A}$.*

In the remainder of this chapter (in all upcoming definitions, lemmas and theorems) we assume that there is always an $\mathcal{A}$ defined which contains approximative signal relations for all involved ports. Given this, we can lift one or multiple approximation relations for signals to an approximation relation for traces as follows:

Definition 77 (Trace Approximation). *Let $X, X' \in \text{Tr}(P)$ be two traces on an interface P . The approximation relation $\approx_{\mathcal{A}}$ for traces with respect to $\mathcal{A}$ is defined as:*

$$X \approx_{\mathcal{A}} X' \equiv \forall_{p \in P} \exists_{\alpha_p \in \mathcal{A}}: X[p] \approx_p X'[p]$$

Next to that, we define the approximation relation for traces that are not necessarily on the same interface, but on interfaces that only share a subset of ports:

Definition 78 (Partial Trace Approximation). *Let $X \in \text{Tr}(P), X' \in \text{Tr}(P')$ be two traces on interfaces P and P' , respectively. The partial approximation relation $\approx_{\mathcal{A}}$ for traces with respect to $\mathcal{A}$ is defined as:*

$$X \approx_{\mathcal{A}} X' \equiv \forall_{p \in P \cap P'} \exists_{\alpha_p \in \mathcal{A}}: X[p] \approx_p X'[p]$$

Note that trace approximation is transitive, whereas partial trace approximation is not. Trace approximation and partial trace approximation are closely related, as shown with the following lemma:

Lemma A.1 (Trace Approximation vs. Partial Trace Approximation). *Let X be a trace on an interface P , let X' be a trace on an interface P' . Then it holds:*

$$X \approx_{\mathcal{A}} X' \Leftrightarrow P = P' \wedge X \approx_{\mathcal{A}} X'$$

Proof. Trivial. □

Crucial to our usage of (partial) trace approximation is further the merging and splitting of traces, for which we prove the following four lemmas:

Lemma A.2 (Partial Trace Approximation Preservation on Merging). *Let X_0 be a trace on an interface P_0 , X_1 a trace on an interface P_1 , X'_0 a trace on an interface P'_0 and X'_1 a trace on an interface P'_1 , with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$. For such traces it holds iff $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ that partial trace approximation is preserved on merging of these traces to $X = X_0 \cup X_1$ and $X' = X'_0 \cup X'_1$, respectively, i.e.:*

$$X_0 \approx_{\mathcal{A}} X'_0 \wedge X_1 \approx_{\mathcal{A}} X'_1 \Rightarrow X \approx_{\mathcal{A}} X'$$

Proof. By definition it holds that X is a trace on interface $P = P_0 \cup P_1$, whereas X' is a trace on $P' = P'_0 \cup P'_1$. Thus it holds that:

$$P \cap P' = (P_0 \cup P_1) \cap (P'_0 \cup P'_1) = (P_0 \cap P'_0) \cup (P_0 \cap P'_1) \cup (P_1 \cap P'_0) \cup (P_1 \cap P'_1)$$

From $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ further follows $P \cap P' = (P_0 \cap P'_0) \cup (P_1 \cap P'_1)$. If $X_0 \approx_{\mathcal{A}} X'_0$ then it holds by definition that $\forall_{p \in P_0 \cap P'_0} \exists_{\alpha_p \in \mathcal{A}}: X_0[p] \approx_p X'_0[p]$ and if $X_1 \approx_{\mathcal{A}} X'_1$ then it holds that $\forall_{p \in P_1 \cap P'_1} \exists_{\alpha_p \in \mathcal{A}}: X_1[p] \approx_p X'_1[p]$. In this case it hence holds that also $\forall_{p \in P \cap P'} \exists_{\alpha_p \in \mathcal{A}}: X[p] \approx_p X'[p]$, i.e. $X \approx_{\mathcal{A}} X'$.

We prove by contradiction that partial trace approximation is not preserved on merging of traces if $P_0 \cap P'_1 \neq \emptyset$ and / or $P'_0 \cap P_1 \neq \emptyset$. Consider traces X_0

on interface $P_0 = \{p_0, p_1\}$, X_1 on interface $P_1 = \{p_2\}$, X'_0 on interface $P'_0 = \{p_0\}$ and X'_1 on interface $P'_1 = \{p_1, p_2\}$. Then it holds that $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$, as well as $P_0 \cap P'_1 = \{p_1\} \neq \emptyset$ and $P'_0 \cap P_1 = \emptyset$. With $\alpha_{p_0}, \alpha_{p_1}, \alpha_{p_2} \in \mathcal{A}$ further let $X_0[p_0] \alpha_{p_0} X'_0[p_0]$, $X_0[p_1] \not\alpha_{p_1} X'_1[p_1]$ and $X_1[p_2] \alpha_{p_2} X'_1[p_2]$. Then it follows that $X_0 \alpha_{\mathcal{A}} X'_0$, $X_1 \alpha_{\mathcal{A}} X'_1$, but $X_0 \cup X_1 \not\alpha_{\mathcal{A}} X'_0 \cup X'_1$. $\square$

Lemma A.3 (Partial Trace Approximation Preservation on Splitting). *Let X be a trace on an interface P , $X_0 \subseteq X$ a subtrace of X on an interface $P_0 \subseteq P$ and $X_1 \subseteq X$ a subtrace of X on interface $P_1 = P \setminus P_0$. Likewise, let X' be a trace on an interface P' , $X'_0 \subseteq X'$ a subtrace of X' on an interface $P'_0 \subseteq P'$ and $X'_1 \subseteq X'$ a subtrace of X' on interface $P'_1 = P' \setminus P'_0$. Then it holds that partial trace approximation is preserved on splitting of trace X to X_0 and X_1 and of X' to X'_0 and X'_1 , respectively, i.e.:*

$$X \alpha_{\mathcal{A}} X' \Rightarrow X_0 \alpha_{\mathcal{A}} X'_0 \wedge X_1 \alpha_{\mathcal{A}} X'_1$$

Proof. If $X \alpha_{\mathcal{A}} X'$ it holds that $\forall p \in P \cap P' \exists \alpha_p \in \mathcal{A} : X[p] \alpha_p X'[p]$. From $P_0 \subseteq P$ and $P'_0 \subseteq P'$ further follows that $P_0 \cap P'_0 \subseteq P \cap P'$. Thus it also holds that $\forall p \in P_0 \cap P'_0 \exists \alpha_p \in \mathcal{A} : X_0[p] \alpha_p X'_0[p]$, i.e. $X_0 \alpha_{\mathcal{A}} X'_0$. $X_1 \alpha_{\mathcal{A}} X'_1$ follows from $X \alpha_{\mathcal{A}} X'$ analogously. $\square$

Lemma A.4 (Trace Approximation Preservation on Merging). *Let X_0 be a trace on an interface P_0 , X_1 a trace on an interface P_1 , X'_0 a trace on an interface P'_0 and X'_1 a trace on an interface P'_1 , with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$. For such traces it holds that trace approximation is preserved on merging of these traces to $X = X_0 \cup X_1$ and $X' = X'_0 \cup X'_1$, respectively, i.e.:*

$$X_0 \alpha_{\mathcal{A}} X'_0 \wedge X_1 \alpha_{\mathcal{A}} X'_1 \Rightarrow X \alpha_{\mathcal{A}} X'$$

Proof. $X_0 \alpha_{\mathcal{A}} X'_0$ implies $P_0 = P'_0$ and $X_0 \alpha_{\mathcal{A}} X'_0$, while $X_1 \alpha_{\mathcal{A}} X'_1$ implies $P_1 = P'_1$ and $X_1 \alpha_{\mathcal{A}} X'_1$. From that follows with $P_0 \cap P_1 = P'_0 \cap P'_1 = \emptyset$ that also $P_0 \cap P'_1 = P'_0 \cap P_1 = \emptyset$ and with Lemma A.2 that $X \alpha_{\mathcal{A}} X'$. As X is a trace on interface $P = P_0 \cup P_1$ and X' is a trace on interface $P' = P'_0 \cup P'_1$ it further follows with $P_0 = P'_0$ and $P_1 = P'_1$ that $P = P'$. $X \alpha_{\mathcal{A}} X'$ and $P = P'$ finally let us conclude that $X \alpha_{\mathcal{A}} X'$. $\square$

Lemma A.5 (Trace Approximation Preservation on Splitting). *Let X be a trace on an interface P , $X_0 \subseteq X$ a subtrace of X on an interface $P_0 \subseteq P$ and $X_1 \subseteq X$ a subtrace of X on interface $P_1 = P \setminus P_0$. Likewise, let X' be a trace on an interface P' , $X'_0 \subseteq X'$ a subtrace of X' on an interface $P'_0 \subseteq P'$ and $X'_1 \subseteq X'$ a subtrace of X' on interface $P'_1 = P' \setminus P'_0$. If $P_0 = P'_0$ and $P_1 = P'_1$ then it holds that trace approximation is preserved on splitting of trace X to X_0 and X_1 and of X' to X'_0 and X'_1 , respectively, i.e.:*

$$X \alpha_{\mathcal{A}} X' \Rightarrow X_0 \alpha_{\mathcal{A}} X'_0 \wedge X_1 \alpha_{\mathcal{A}} X'_1$$

Proof. Trivial. $\square$

A.3.2 COMPONENT APPROXIMATION

Given trace approximation we define best-case and worst-case approximation of components as follows:

Definition 79 (Component Approximation). *Let A, A' be two components with input interfaces P_A and $P_{A'}$ and output interfaces Q_A and $Q_{A'}$, respectively.*

Component A is a best-case under-approximation of A' with respect to $\mathcal{A}$, i.e. $A \sqsubseteq_{\mathcal{A}} A'$, iff it holds $\forall_{X \in in_A, X' \in in_{A'}} that:$

$$X \sqsubseteq_{\mathcal{A}} X' \Rightarrow \forall_{X'A'Y'} \exists_{XAY}: Y \sqsubseteq_{\mathcal{A}} Y'$$

Analogously, component A is a worst-case over-approximation of A' with respect to $\mathcal{A}$, i.e. $A \supseteq_{\mathcal{A}} A'$, iff it holds $\forall_{X \in in_A, X' \in in_{A'}} that:$

$$X \supseteq_{\mathcal{A}} X' \Rightarrow \forall_{X'A'Y'} \exists_{XAY}: Y \supseteq_{\mathcal{A}} Y'$$

In words this means that A is a best-case under-approximation (worst-case over-approximation) of A' with respect to $\mathcal{A}$ iff for every input trace X of A that is a partial under-approximation (over-approximation) of an input trace X' of A' there exists an output trace Y with XAY that is a partial under-approximation (over-approximation) of every output trace Y' with $X'A'Y'$.

A.3.3 APPROXIMATION LIFTING

In this section we discuss the automatic lifting of approximation from component to graph level, i.e. that two graphs approximate each other if all their components approximate each other. For that purpose we prove that approximation between individual components is preserved on parallel, serial and feedback compositions.

For parallel composition it holds:

Lemma A.6 (Parallel Approximation Preservation). *With $\nabla = \sqsubseteq_{\mathcal{A}}$ ($\nabla = \supseteq_{\mathcal{A}}$) let $A \nabla A'$ and $B \nabla B'$. Furthermore let $Q_A \cap Q_{B'} = Q_{A'} \cap Q_B = \emptyset$. Then it holds that the respective parallel compositions also approximate each other with respect to $\mathcal{A}$, i.e. $A || B \nabla A' || B'$.*

Proof. Trivial as the input and output interfaces of the composed components are disjoint, as both input and output traces of parallel compositions $A || B$ and $A' || B'$ are merged from input and output traces of A and B and of A' and B' , respectively, and as $Q_A \cap Q_{B'} = Q_{A'} \cap Q_B = \emptyset$ guarantees that partial approximation is lifted from subtraces to merged output traces, according to Lemma A.2. $\square$

For serial composition we obtain analogously:

Lemma A.7 (Serial Approximation Preservation). *With $\nabla = \underline{\alpha}_{\mathcal{A}}$ ($\nabla = \overline{\alpha}_{\mathcal{A}}$) let $A \nabla A'$ and $B \nabla B'$. For the serial compositions $A \Xi B$ and $A' \Xi' B'$ let further $P_B^* \cap P_{B'}^\circ = P_{B'}^* \cap P_B^\circ = Q_A^\circ \cap Q_{B'} = Q_{A'}^\circ \cap Q_B = \emptyset$ and let $\forall_{q \in Q_A^*} : \alpha_q = \alpha_{\Xi}(q) \in \mathcal{A}, \forall_{q' \in Q_{A'}^*} : \alpha_{q'} = \alpha_{\Xi'}(q') \in \mathcal{A}$, as well as $\forall_{\Xi(q) = \Xi'(q') \in P_B^* \cap P_{B'}^*} : q = q'$. Then it holds that the respective serial compositions also approximate each other with respect to $\mathcal{A}$, i.e. $A \Xi B \nabla A' \Xi' B'$.*

Proof Idea. From $A \nabla A'$ follows that for any input that is accepted by $A \Xi B$ and that partially approximates an input of $A' \Xi' B'$, there must be an output of A that partially approximates all respective outputs of A' . As these outputs must be accepted by B , according to the third line in the definition of serial composition, it follows with $B \nabla B'$ that there must also be an output of B that partially approximates all respective outputs of B' . This lets us conclude that for any input of $A \Xi B$ that partially approximates an input of $A' \Xi' B'$ there must be an output of $A \Xi B$ that partially approximates all respective outputs of $A' \Xi' B'$, i.e. $A \Xi B \nabla A' \Xi' B'$.

Proof. Let $X_A \cup X_B^\circ \in in_{A \Xi B}$, $X'_A \cup X_B^{\circ'} \in in_{A' \Xi' B'}$ and with $\Delta = \alpha_{\mathcal{A}}$ ($\Delta = \alpha_{\mathcal{A}}$) let $X_A \cup X_B^\circ \Delta X'_A \cup X_B^{\circ'}$. With Lemma A.3 it holds that also $X_A \Delta X'_A$ and from $A \nabla A'$ follows:

$$\forall_{X'_A A'(Y_A^{\circ'} \cup Y_A^{\circ})} \exists_{X_A A(Y_A^* \cup Y_A^\circ)} : (Y_A^* \cup Y_A^\circ) \Delta (Y_A^{\circ'} \cup Y_A^\circ) \quad (\text{A.1})$$

$(X_A \cup X_B^\circ) \in in_{A \Xi B}$ implies that it holds for all Y_A^* with $X_A A(Y_A^* \cup Y_A^\circ)$ that $(X_B^\circ \cup C^\Xi(Y_A^*)) \in in_B$. Analogously $(X'_A \cup X_B^{\circ'}) \in in_{A' \Xi' B'}$ implies that it holds for all $Y_A^{\circ'}$ with $X'_A A'(Y_A^{\circ'} \cup Y_A^\circ)$ that $(X_B^{\circ'} \cup C^{\Xi'}(Y_A^{\circ'})) \in in_{B'}$.

For $(Y_A^* \cup Y_A^\circ)$ and $(Y_A^{\circ'} \cup Y_A^\circ)$ according to Equation A.1 it follows from $(Y_A^* \cup Y_A^\circ) \Delta (Y_A^{\circ'} \cup Y_A^\circ)$ with Lemma A.3, from $\forall_{q \in Q_A^*} : \alpha_q = \alpha_{\Xi}(q) \in \mathcal{A}, \forall_{q' \in Q_{A'}^*} : \alpha_{q'} = \alpha_{\Xi'}(q') \in \mathcal{A}$ and from $\forall_{\Xi(q) = \Xi'(q') \in P_B^* \cap P_{B'}^*} : q = q'$ that $C^\Xi(Y_A^*) \Delta C^{\Xi'}(Y_A^{\circ'})$. With Lemma A.2 and $P_B^* \cap P_{B'}^\circ = P_{B'}^* \cap P_B^\circ$ it further follows that $(X_B^\circ \cup C^\Xi(Y_A^*)) \Delta (X_B^{\circ'} \cup C^{\Xi'}(Y_A^{\circ'}))$. With $B \nabla B'$ we obtain:

$$\forall_{(X_B^{\circ'} \cup C^{\Xi'}(Y_A^{\circ'})) B' Y_B'} \exists_{(X_B^\circ \cup C^\Xi(Y_A^*)) B Y_B} : Y_B \Delta Y_{B'}$$

From this follows with Lemma A.2 and $Q_A^\circ \cap Q_{B'} = Q_{A'}^\circ \cap Q_B = \emptyset$ that also $(Y_A^\circ \cup Y_B) \Delta (Y_A^{\circ'} \cup Y_{B'})$, such that it can be finally concluded $\forall_{(X_A \cup X_B^\circ) \in in_{A \Xi B}} \text{ and } \forall_{(X'_A \cup X_B^{\circ'}) \in in_{A' \Xi' B'}} :$

$$\begin{aligned} & (X_A \cup X_B^\circ) \Delta (X'_A \cup X_B^{\circ'}) \\ & \Rightarrow \forall_{(X'_A \cup X_B^{\circ'}) A' \Xi' B' (Y_A^{\circ'} \cup Y_B')} \exists_{(X_A \cup X_B^\circ) A \Xi B (Y_A^* \cup Y_B)} : (Y_A^\circ \cup Y_B) \Delta (Y_A^{\circ'} \cup Y_{B'}) \end{aligned}$$

This is just the definition of $A \Xi B \nabla A' \Xi' B'$, q.e.d. $\square$

Note that Lemma A.7 does neither imply $A \Xi B \nabla A \Xi' B'$ nor $A \Xi B \nabla A' \Xi' B$ in general.

For feedback composition it holds:

Lemma A.8 (Feedback Approximation Preservation). *With $\nabla = \underline{\nabla}_{\mathcal{A}}$ ($\nabla = \overline{\nabla}_{\mathcal{A}}$) let $A \nabla A'$. For the feedback compositions $A \Xi A$ and $A' \Xi' A'$ let further $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and let $\forall_{q \in Q_A^*} : \alpha_q = \alpha_{\Xi(q)} \in \mathcal{A}, \forall_{q' \in Q_{A'}^*} : \alpha_{q'} = \alpha_{\Xi'(q')} \in \mathcal{A}$, as well as $\forall_{\Xi(q) = \Xi'(q') \in P_A^* \cap P_{A'}^*} : q = q'$. Then it holds that the respective feedback compositions also approximate each other with respect to $\mathcal{A}$, i.e. $A \Xi A \nabla A' \Xi' A'$.*

Proof Idea. Let there be an input that is accepted by $A \Xi A$ and that partially approximates an input that is accepted by $A' \Xi' A'$. Consequently, the respective combinations of these traces with empty traces on the internal interfaces also partially approximate each other. From $A \nabla A'$ then follows that for these inputs there exists an output of A that partially approximates all respective outputs of A' . With the third line of the definition of feedback composition it follows that these outputs must be accepted by A and A' as inputs, respectively, again resulting in an output of A that partially approximates all outputs of A' . As the third line ensures that for any such partially approximating sequences fixed points are reached, it follows that also the respective fixed points partially approximate each other. This lets us conclude that for any input of $A \Xi A$ that partially approximates an input of $A' \Xi' A'$ there exists an output of $A \Xi A$ that partially approximates all respective outputs of $A' \Xi' A'$, i.e. $A \Xi A \nabla A' \Xi' A'$.

Proof. We denote with $X_k = (X^\diamond \cup X_k^*)$ and $Y_k = (Y_k^* \cup Y_k^\diamond)$ the input and output traces of component A in each feedback iteration k after assignment of an external input trace $X^\diamond$ and with $X'_k = (X^{\diamond'} \cup X_k^{*'})$ and $Y'_k = (Y_k^{*'} \cup Y_k^{\diamond'})$ the input and output traces of A' after assignment of $X^{\diamond'}$ analogously.

Given that an external input of A is a partial under-approximation (over-approximation) of an external input of A' we first show with $\Delta = \underline{\Delta}_{\mathcal{A}}$ ($\Delta = \overline{\Delta}_{\mathcal{A}}$) that for each feedback iteration there exist output traces of A that are partial under-approximation (over-approximations) all output traces of A' , using mathematical induction. Based on this we prove that there exists a fixed point of A for each fixed point of A' , such that the output traces of A are partial under-approximations (over-approximations) of the output traces of A' . We begin with the mathematical induction:

Induction base: Let $X^\diamond \in in_{A \Xi A}$, $X^{\diamond'} \in in_{A' \Xi' A'}$ and $X^\diamond \Delta X^{\diamond'}$. It follows from the definition of components that initially (before $X^\diamond$ and $X^{\diamond'}$ are assigned) the traces on all interfaces are empty traces, such that the input traces of A on assignment of $X^\diamond$ and A' on assignment of $X^{\diamond'}$ are $X_0 = (X^\diamond \cup C^\Xi(Y^{*,\infty}))$ and $X'_0 = (X^{\diamond'} \cup C^{\Xi'}(Y^{*',\infty}))$, respectively. With the definition of feedback composition it follows from $X^\diamond \in in_{A \Xi A}$ that $X_0 \in in_A$ and from $X^{\diamond'} \in in_{A' \Xi' A'}$ that $X'_0 \in in_{A'}$. From $Y^{*,\infty} \Delta Y^{*',\infty}$ (due to the reflexivity of approximative signal relations) follows with $\forall_{q \in Q_A^*} : \alpha_q = \alpha_{\Xi(q)} \in \mathcal{A}, \forall_{q' \in Q_{A'}^*} : \alpha_{q'} = \alpha_{\Xi'(q')} \in \mathcal{A}$ and $\forall_{\Xi(q) = \Xi'(q') \in P_A^* \cap P_{A'}^*} : q = q'$ that also $C^\Xi(Y^{*,\infty}) \Delta C^{\Xi'}(Y^{*',\infty})$ and from $X^\diamond \Delta X^{\diamond'}$ it follows with $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and Lemma A.2 that $X_0 \Delta X'_0$. From $A \nabla A'$ we can then conclude that there exists an $X_0 A Y_0$ for all $X'_0 A' Y'_0$ such that $Y_0 \Delta Y'_0$.

Induction hypothesis: Let $X_k \in in_A$, $X'_k \in in_{A'}$, let $X_k \triangle X'_k$ and let an $X_k A Y_k$ exist for all $X'_k A' Y'_k$ such that $Y_k \triangle Y'_k$.

Induction step: With the definitions of interfaces and feedback composition it follows from $X^\diamond \in in_{A \boxplus A}$ that $X_{k+1} = (X^\diamond \cup C^\Xi(Y_k^*)) \in in_A$ and from $X^{\diamond'} \in in_{A' \boxplus A'}$ that $X'_{k+1} = (X^{\diamond'} \cup C^{\Xi'}(Y_k^{*'})) \in in_{A'}$. From $Y_k \triangle Y'_k$ it follows with $\forall q \in Q_A^* : \alpha_q = \alpha_{\Xi(q)} \in \mathcal{A}$, $\forall q' \in Q_{A'}^* : \alpha_{q'} = \alpha_{\Xi'(q')} \in \mathcal{A}$ and $\forall \Xi(q) = \Xi'(q') \in P_A^* \cap P_{A'}^* : q = q'$ that also $C^\Xi(Y_k) \triangle C^{\Xi'}(Y_k')$ and from $X^\diamond \triangle X^{\diamond'}$ follows with $P_A^\diamond \cap P_{A'}^* = P_{A'}^\diamond \cap P_A^* = \emptyset$ and Lemma A.2 that $X_{k+1} \triangle X'_{k+1}$. $A \nabla A'$ lets us conclude that there exists an $X_{k+1} A Y_{k+1}$ for all $X'_{k+1} A' Y'_{k+1}$ such that $Y_{k+1} \triangle Y'_{k+1}$.

Thus it holds for $X^\diamond \in in_{A \boxplus A}$, $X^{\diamond'} \in in_{A' \boxplus A'}$ and $X^\diamond \triangle X^{\diamond'}$ that:

$$\forall k \in \mathbb{N} : \forall X'_k A' Y'_k \exists X_k A Y_k : X_k \triangle X'_k \wedge Y_k \triangle Y'_k \quad (A.2)$$

According to the definition of feedback composition $X^\diamond \in in_{A \boxplus A}$ implies that for any sequence $(X^\diamond \cup C^\Xi(Y_k^*))A(Y_{k+1}^* \cup Y_{k+1}^\diamond)$ with $Y_0^* = Y^{*,\infty}$ there exists a $Y_\infty = \lim_{k \rightarrow \infty} Y_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^\diamond \cup C^\Xi(Y_\infty^*))A(Y_\infty^* \cup Y_\infty^\diamond)$. Analogously $X^{\diamond'} \in in_{A' \boxplus A'}$ implies that for any sequence $(X^{\diamond'} \cup C^{\Xi'}(Y_k^{*'}))A'(Y_{k+1}^{*'} \cup Y_{k+1}^{\diamond'})$ with $Y_0^{*'} = Y^{*,\infty}$ there exists a $Y'_\infty = \lim_{k \rightarrow \infty} Y'_k$ that defines a fixed point of the sequence, i.e. it holds that $(X^{\diamond'} \cup C^{\Xi'}(Y_\infty^{*'}))A'(Y_\infty^{*'} \cup Y_\infty^{\diamond'})$.

Now consider a sequence $(X^\diamond \cup C^\Xi(Y_k^*))A(Y_{k+1}^* \cup Y_{k+1}^\diamond)$ that for any sequence $(X^{\diamond'} \cup C^{\Xi'}(Y_k^{*'}))A'(Y_{k+1}^{*'} \cup Y_{k+1}^{\diamond'})$ satisfies $X_k \triangle X'_k$ and $Y_k \triangle Y'_k$ for all $k \in \mathbb{N}$ (existence of such a sequence is guaranteed by Equation A.2). Furthermore, let Y_∞ and Y'_∞ be the respective fixed points of such sequences. Then it holds that $Y_\infty \triangle Y'_\infty$ and with Lemma A.3 also $Y_\infty^\diamond \triangle Y_\infty^{\diamond'}$. With $Y^\diamond = Y_\infty^\diamond$ and $Y^{\diamond'} = Y_\infty^{\diamond'}$ this lets us finally conclude $\forall X^\diamond \in in_{A \boxplus A}, X^{\diamond'} \in in_{A' \boxplus A'} :$

$$X^\diamond \triangle X^{\diamond'} \Rightarrow \forall (X^{\diamond'})A'\Xi'(Y^{\diamond'}) \exists (X^\diamond)A\Xi(Y^\diamond) : Y^\diamond \triangle Y^{\diamond'}$$

This is just the definition of $A \boxplus A \nabla A' \boxplus A'$, q.e.d. $\square$

Lemmas A.6 to A.8 prove the automatic lifting of approximation from component to graph level:

Theorem A.1 (Approximation Lifting). *With $\nabla = \underline{\alpha}_{\mathcal{A}}$ ($\nabla = \overline{\alpha}_{\mathcal{A}}$) let G be a graph composed of components A and let G' be a graph composed of components A' . If it holds for all components of G that they are best-case under-approximations (worst-case over-approximations) of the respective components of G' with respect to $\mathcal{A}$, i.e. $A \nabla A'$, and if all compositions satisfy the additional interface requirements from Lemmas A.6 to A.8 then it follows that also the graph G is a best-case under-approximation (worst-case over-approximation) of graph G' with respect to $\mathcal{A}$, i.e. $G \nabla G'$.*

Proof. Trivial from Lemmas A.6 to A.8 and the fact that graphs consist of parallel, serial and feedback compositions. $\square$

A.3.4 TRANSITIVITY & REFLEXIVITY

For the presented definition of component approximation, Definition 79, general transitivity does not hold, i.e. with $\nabla = \underline{\alpha}_{\mathcal{A}}$ ($\nabla = \overline{\alpha}_{\mathcal{A}}$) it does not hold in general that $A \nabla A' \nabla A'' \Rightarrow A \nabla A''$. Nevertheless, we can define and prove a restricted form of transitivity that is sufficient for our needs as follows:

Theorem A.2 (Restricted Approximation Transitivity). *With $\nabla = \underline{\alpha}_{\mathcal{A}}$ ($\nabla = \overline{\alpha}_{\mathcal{A}}$) let $A \nabla A' \nabla A''$, let $Q_A = Q_{A'} = Q_{A''}$ and with $\Delta = \underline{\alpha}_{\mathcal{A}}$ ($\Delta = \overline{\alpha}_{\mathcal{A}}$) let $\forall X \in in_A, X'' \in in_{A''}$:*

$$X \Delta X'' \Rightarrow \exists X' \in in_{A'}: X \Delta X' \Delta X''$$

Then it holds that $A \nabla A''$. This means that transitivity of component approximation applies given that the output interfaces of A , A' and A'' are the same and that for all partially approximating input traces in in_A and $in_{A''}$ there exists a trace in $in_{A'}$ such that partial approximation transitivity holds for these traces. In particular, the latter condition is always satisfied if either $in_A \subseteq in_{A'} \subseteq in_{A''}$ or $in_A \supseteq in_{A'} \supseteq in_{A''}$.

Proof. Let $\Delta = \underline{\alpha}_{\mathcal{A}}$ ($\Delta = \overline{\alpha}_{\mathcal{A}}$). For any $X \in in_A, X'' \in in_{A''}$ with $X \Delta X''$ let there exist an $X' \in in_{A'}$ such that $X \Delta X' \Delta X''$. From $A \nabla A'$ follows that $\forall X', A', Y' \exists XAY': Y \Delta Y'$ and from $Q_A = Q_{A'}$ with Lemma A.1 that also $Y \Delta Y'$. Likewise it follows from $A' \nabla A''$ that $\forall X'', A'', Y'' \exists X'AY'': Y' \Delta Y''$ and from $Q_{A'} = Q_{A''}$ that also $Y' \Delta Y''$. From combining the two statements and taking into account that trace approximation is transitive (as opposed to partial trace approximation) it can be concluded that $Y \Delta Y''$ and with Lemma A.1 that also $Y \Delta Y''$. This implies $A \nabla A''$.

Correctness of the last sentence of the theorem follows immediately from the reflexivity of partial trace approximation. $\square$

The restricted approximation transitivity allows us to create multiple layers of approximating components while still allowing to relate the behaviors of the highest and lowest levels to each other.

Next to transitivity also reflexivity of component approximation does not always hold, i.e. with $\nabla = \underline{\alpha}_{\mathcal{A}}$ ($\nabla = \overline{\alpha}_{\mathcal{A}}$) it does not hold in general that $A \nabla A$. Instead we show in the following that approximation reflexivity is tightly coupled to monotonicity:

Theorem A.3 (Approximation Reflexivity vs. Monotonicity). *With $\nabla = \underline{\alpha}_{\mathcal{A}}$ ($\nabla = \overline{\alpha}_{\mathcal{A}}$) it holds that $A \nabla A$ is equivalent to A being best-case (worst-case) $\underline{\alpha}_{\mathcal{A}}$ -monotone.*

Proof. Trivial from Definitions 19 and 79. $\square$

The most important implication of the lack of general reflexivity is that Lemma A.7 does neither imply $A' \Xi' B \nabla A \Xi B$ nor $A \Xi' B' \nabla A \Xi B$. Instead it must additionally hold that A is best-case (worst-case) $\underline{\alpha}_{\mathcal{A}}$ -monotone if only B is replaced by B' or that B is best-case (worst-case) $\underline{\alpha}_{\mathcal{A}}$ -monotone if only A is replaced by A' .

After defining the generic approximation of signals, traces and components, as well as proving the automatic lifting of approximation from component to graph level we can now also define abstraction and refinement. Subsequently we discuss that abstraction and refinement can be directly concluded from input acceptance preservation and approximation.

Definition 80 (Abstraction). *A graph G is a best-case (worst-case) abstraction of a graph G' with respect to $\mathcal{A}$ iff G accepts at least all input traces that G' also accepts, i.e. $in_G \supseteq in_{G'}$, and iff it holds for all accepted input traces of G' that for these input traces there exists an output trace of G that is an under-approximation (over-approximation) of all output traces of G' for the same input traces, i.e. with $\Delta = \alpha_{\mathcal{A}}$ ($\Delta = \omega_{\mathcal{A}}$) it holds that:*

$$\forall X \in in_{G'}: \forall_{XG'Y'} \exists_{XGY}: Y \Delta Y'$$

Definition 81 (Refinement). *A graph G' is a best-case (worst-case) refinement of a graph G with respect to $\mathcal{A}$ iff G' accepts at least all input traces that G also accepts, i.e. $in_G \subseteq in_{G'}$, and iff it holds for all accepted input traces of G that for these input traces there exists an output trace of G that is an under-approximation (over-approximation) of all output traces of G' for the same input traces, i.e. with $\Delta = \alpha_{\mathcal{A}}$ ($\Delta = \omega_{\mathcal{A}}$) it holds that:*

$$\forall X \in in_G: \forall_{XG'Y'} \exists_{XGY}: Y \Delta Y'$$

Definition 80 applies to the creation of best-case and worst-case models for the analysis of an existing implementation (for which the specification lies in the implementation), while Definition 81 applies to the design process from a best-case and / or worst-case model to an implementation (for which the specification is given by the models).

With above definitions of abstraction and refinement we can conclude the following three important theorems:

Theorem A.4 (Abstraction). *A graph G is a best-case (worst-case) abstraction of a graph G' with respect to $\mathcal{A}$ if $G \underline{\alpha}_{\mathcal{A}} G'$ ($G \overline{\omega}_{\mathcal{A}} G'$), if $P_G = P_{G'}$ and $Q_G = Q_{G'}$ and if $in_G \supseteq in_{G'}$.*

Proof. Follows immediately from Definitions 79 and 80, the fact that the $\alpha_{\mathcal{A}}$ relation is reflexive, i.e. $X \alpha_{\mathcal{A}} X$, as well as Lemma A.1. $\square$

Theorem A.5 (Refinement). *A graph G' is a best-case (worst-case) refinement of a graph G with respect to $\mathcal{A}$ if $G \underline{\alpha}_{\mathcal{A}} G'$ ($G \overline{\omega}_{\mathcal{A}} G'$), if $P_G = P_{G'}$ and $Q_G = Q_{G'}$ and if $in_G \subseteq in_{G'}$.*

Proof. Follows immediately from Definitions 79 and 81, the fact that the $\alpha_{\mathcal{A}}$ relation is reflexive, i.e. $X \alpha_{\mathcal{A}} X$, as well as Lemma A.1. $\square$

Theorem A.6 (Abstraction-Refinement Transitivity). *With $\Delta = \underline{\alpha}_{\mathcal{A}}$ ($\Delta = \overline{\alpha}_{\mathcal{A}}$) let it hold for three graphs G, G' and G'' that $G \Delta G' \Delta G''$, that $P_G = P_{G'} = P_{G''}$ and $Q_G = Q_{G'} = Q_{G''}$ and that $\text{in}_G \supseteq \text{in}_{G'} \supseteq \text{in}_{G''}$. Then it holds that G is a best-case (worst-case) abstraction of G'' with respect to $\mathcal{A}$. If it holds instead that $\text{in}_G \subseteq \text{in}_{G'} \subseteq \text{in}_{G''}$ then it follows that G'' is a best-case (worst-case) refinement of G with respect to $\mathcal{A}$.*

Proof. Follows immediately from Definitions 80 and 81, Theorems A.2, A.4 and A.5, as well as the fact that the $\alpha_{\mathcal{A}}$ relation is reflexive, i.e. $X \alpha_{\mathcal{A}} X$. □

Additional Proofs for Real-Time System Analysis Techniques

In this appendix we provide additional proofs for the analysis techniques presented in the second part of the thesis.

First, we prove that all three analysis techniques, the period-and-jitter analysis, execution intervals and the phase analysis, terminate, given either the specification of maximum end-to-end latencies for all task graphs or of upper bounds on all buffer capacities for the case of an iterative buffer sizing. For that matter we show that the maximum response times are valid convergence criteria, such that all parameters converge if maximum response times converge, that all steps in the analysis flows are monotonically increasing, i.e. all parameters are increasing in increasing iterations of the flows, and finally that all flows either converge or stop on constraint violation. The steps in the single-phase and multi-phase analysis flows are conceptually equivalent, such that we confine ourselves to proving termination for single-phase period-and-jitter in Section B.1 and termination for single-phase execution intervals in Section B.2.

Lastly, we show in Section B.3 that the stop criterion in line 15 of Figure 12.5 is valid, i.e. no larger maximum response times can be found after the stop criterion is met.

B.1 TERMINATION OF THE PERIOD-AND-JITTER ANALYSIS

Previously we have discussed that given convergence temporally conservative analysis results are determined by both our period-and-jitter analysis flows. This leaves two properties to be shown: We need to prove that the maximum response times are a valid convergence criterion, i.e. that if maximum response times converge also all other parameters converge. And it still remains to be shown that the two analysis flows terminate.

The following theorem implies that the maximum response times of tasks are a valid convergence criterion:

Theorem B.1 (Period-and-Jitter Convergence Criterion). *Let $\hat{R}_a^n$, $\hat{\rho}_a^n$, $\hat{J}_a^n$ and $\delta_{b,a}^n$ denote maximum response times, maximum firing durations, maximum enabling jitters and estimated numbers of initial tokens that are calculated in iterations n of the period-and-jitter analysis flows. If all maximum response times converge, i.e. $\forall_{t_a \in T, n \geq 0}: \hat{R}_a^{n+1} = \hat{R}_a^n$, then the same holds for all other parameters.*

Proof. The maximum firing durations are set to the maximum response times determined in the previous iterations of the analysis flows, i.e. it holds that $\forall_{\hat{v}_a \in \hat{V}, n \geq 0}: \hat{\rho}_a^{n+1} = \hat{R}_a^n$. From this follows:

$$\forall_{t_a \in T, n \geq 0}: \hat{R}_a^{n+1} = \hat{R}_a^n \Rightarrow \forall_{\hat{v}_a \in \hat{V}, n \geq 0}: \hat{\rho}_a^{n+2} = \hat{\rho}_a^{n+1}$$

The maximum enabling jitters are computed based on the minimum and maximum enabling times derived from the respective best-case and worst-case analysis models. The minimum enabling times are computed using BCETs of tasks and remain constant throughout all iterations of the analysis flows. The maximum enabling times from the worst-case analysis model can only change due to changed maximum firing durations, as we do not consider buffer capacity estimates in that model (recall that during analysis only upper bounds or infinite numbers of initial tokens are considered in the worst-case analysis model, to ensure flow monotonicity, whereas the computed estimates are inserted in the worst-case model only after convergence, to ensure bounding between response time analysis and worst-case analysis models). Combining this observation with the previous implication results in the following:

$$\forall_{t_a \in T, n \geq 0}: \hat{R}_a^{n+1} = \hat{R}_a^n \Rightarrow \forall_{t_a \in T, n \geq 0}: \hat{J}_a^{n+2} = \hat{J}_a^{n+1}$$

Lastly, for the case of an iterative buffer sizing, the buffer capacity estimates are solely based on minimum and maximum enabling times and maximum firing durations, for which we have already proven that they remain constant on converged maximum response times. Thus we can conclude:

$$\forall_{t_a \in T, n \geq 0}: \hat{R}_a^{n+1} = \hat{R}_a^n \Rightarrow \forall_{e_{b,a} \in E, n \geq 0}: \delta_{b,a}^{n+2} = \delta_{b,a}^{n+1}$$

□

The preceding theorem allows us to discuss termination based on maximum response times only. For that matter we first show that no maximum response time can decrease in successive flow iterations, which is captured by the following lemma:

Lemma B.1 (Period-and-Jitter Analysis Flow Monotonicity). *Let $\hat{R}_a^n$ denote the maximum response times that are calculated in iterations n of the period-and-jitter analysis flows. It holds that all maximum response times are monotonically increasing in increasing flow iterations, i.e.:*

$$\forall_{t_a \in T, n \geq 0}: \hat{R}_a^n \leq \hat{R}_a^{n+1}$$

Proof. We prove this lemma with mathematical induction and use $\hat{\rho}_a^n$, $\hat{J}_a^n$ and $\delta_{b,a}^n$ to denote the maximum firing durations, maximum enabling jitters and estimated numbers of initial tokens that are calculated in iterations n of the analysis flows.

Induction base: In analysis flow iterations 0 all maximum firing durations are set to the respective task WCETs, whereas in iterations 1 the firing durations are set to the maximum response times determined in iterations 0. As we only consider non-negative WCETs and jitters it follows that the maximum response times $\hat{R}_a^n$ can never be smaller than the respective WCETs. Hence it holds that $\forall_{\hat{v}_a \in \hat{V}}: \hat{\rho}_a^0 \leq \hat{\rho}_a^1$.

One fundamental property of dataflow models is that their temporal behavior is monotone, which implies that larger firing durations can never result in smaller enabling times. Consequently, the maximum enabling times determined in iteration 1 cannot be smaller than the ones determined in iteration 0, while the minimum enabling times remain constant. This lets us conclude that also the maximum enabling jitters cannot decrease from iteration 0 to iteration 1, i.e. $\forall_{t_a \in T}: \hat{J}_a^0 \leq \hat{J}_a^1$.

For the case of an iterative buffer sizing, the estimated initial token numbers of non-blocking write buffers are increasing both in increasing maximum firing durations and maximum enabling times. For blocking write buffers, however, it can occur that increasing maximum enabling times result in decreasing numbers of initial tokens. For that reason we clamp these numbers of tokens by the respective numbers from the previous flow iteration. While this can result in a suboptimal buffer sizing in some cases, it also ensures monotonicity of the iterative buffer sizing analysis flow. Hence it holds $\forall_{e_{b,a} \in E}: \delta_{b,a}^0 \leq \delta_{b,a}^1$.

Finally, the maximum busy periods for all q increase both in increasing maximum enabling jitters and estimated numbers of initial tokens, as both can result in more, but never less interference of higher priority tasks. Therefore also the respective maximum response times cannot decrease, i.e. it holds that $\forall_{t_a \in T}: \hat{R}_a^0 \leq \hat{R}_a^1$.

Induction hypothesis: Let:

$$\forall_{\hat{v}_a \in \hat{V}, t_a \in T, e_{b,a} \in E}: \hat{\rho}_a^{n-1} \leq \hat{\rho}_a^n \wedge \hat{J}_a^{n-1} \leq \hat{J}_a^n \wedge \delta_{b,a}^{n-1} \leq \delta_{b,a}^n \wedge \hat{R}_a^{n-1} \leq \hat{R}_a^n$$

Induction step: As it holds that $\forall_{\hat{v}_a \in \hat{V}}: \hat{\rho}_a^n = \hat{R}_a^{n-1} \wedge \hat{\rho}_a^{n+1} = \hat{R}_a^n$ it follows immediately from the induction hypothesis that $\forall_{\hat{v}_a \in \hat{V}}: \hat{\rho}_a^n \leq \hat{\rho}_a^{n+1}$. Based on this, we can apply the same reasoning as in the induction base on the maximum enabling jitters, estimated numbers of initial tokens and maximum response times, such that $\forall_{t_a \in T}: \hat{J}_a^n \leq \hat{J}_a^{n+1}$, $\forall_{e_{b,a} \in E}: \delta_{b,a}^n \leq \delta_{b,a}^{n+1}$ and finally $\forall_{t_a \in T}: \hat{R}_a^n \leq \hat{R}_a^{n+1}$.

This concludes the induction and therewith also this proof. $\square$

With the analysis flow monotonicity we can finally also prove termination of the flows. We begin with the proof of termination given bounds on end-to-end latencies:

Theorem B.2 (Period-and-Jitter Analysis Flow Termination 1). *If the period-and-jitter analysis flows are applied on applications whose task graphs are constrained by maximum end-to-end latencies L , then the analysis flows terminate, i.e. they either converge or stop on a constraint violation.*

Proof. According to Theorem B.1 the analysis flow converges if all maximum response times converge. The results of $\gamma_{b \rightarrow a}(\Delta\tau, q)$ from Equation 10.10 are always integer. Therefore it follows with Lemma B.1 that maximum response times increase in minimum steps defined by the WCETs of higher priority tasks, such that an infinite convergence cannot occur. Furthermore, it holds that the end-to-end latencies of task graphs are defined by sums of maximum response times, according to the dependencies in the worst-case LPs. Hence we can conclude that the maximum response times either converge or eventually a temporal constraint imposed by an end-to-end latency must be violated, which proves termination. $\square$

For the period-and-jitter analysis flow with an iterative buffer sizing we additionally prove termination given upper bounds on buffer capacities:

Theorem B.3 (Period-and-Jitter Analysis Flow Termination 2). *If the period-and-jitter analysis flow with an iterative buffer sizing is applied on an application whose FIFO buffer capacities are all bounded from above by maximum numbers of initially empty locations $\hat{\delta}_{b,a}$, then the analysis flow terminates.*

Proof. For each iteratively determined number of initially empty buffer locations $\delta_{b,a}$ we have a corresponding number of initially full locations $\delta_{a,b}$ with a constraint $\hat{s}_b - \hat{s}_a \geq \hat{\rho}_a - \delta_{a,b} \cdot P_a$ in the worst-case LP. Assuming blocking write buffers, it follows for a number of initially empty locations computed with Equation 10.15 (with $P_a = P_b$ as both tasks belong to the same task graph):

$$\delta_{b,a} = \left\lceil \frac{\hat{s}_b + \hat{\rho}_b - \hat{s}_a}{P_b} \right\rceil \geq \frac{\hat{\rho}_a + \hat{\rho}_b - \delta_{a,b} \cdot P_a}{P_b} \Leftrightarrow (\delta_{a,b} + \delta_{b,a}) \cdot P_b \geq \hat{\rho}_a + \hat{\rho}_b$$

The same inequality also applies for non-blocking write buffers, as for such buffers the iteratively determined numbers of initial tokens are never smaller than the ones of blocking write buffers.

For buffers that are fixed, i.e. that are not determined with our iterative buffer sizing, we have two constraints in the worst-case LP, $\hat{s}_b - \hat{s}_a \geq \hat{\rho}_a - \delta_{a,b} \cdot P_a$ and $\hat{s}_a - \hat{s}_b \geq \hat{\rho}_b - \delta_{b,a} \cdot P_b$. Adding these constraints to each other results in the same inequality as above.

Recall that maximum firing durations are set to maximum response times. For iteratively determined buffer capacities it follows that divergent maximum response times $\hat{R}_a$ or $\hat{R}_b$ let the sum $\delta_{a,b} + \delta_{b,a}$ increase until the specified upper bound is exceeded, i.e. $\delta_{a,b} + \delta_{b,a} > \delta_{a,b} + \hat{\delta}_{b,a}$, which is observed as a constraint violation. For fixed buffer capacities it can occur that divergent maximum

response times $\hat{R}_a$ or $\hat{R}_b$ become so large that the aforementioned inequality is violated, i.e. $\hat{\rho}_a + \hat{\rho}_b > (\delta_{a,b} + \delta_{b,a}) \cdot P_b$, in which case the worst-case LP cannot find a valid solution.

As we assume FIFO buffer communication between all tasks it further holds that all maximum firing durations are part of at least one such constraint. With Lemma B.1 and the fact that maximum response times are increasing in minimum steps we can finally conclude termination of the period-and-jitter analysis flow with an iterative buffer sizing, given the specification of upper bounds on buffer capacities. $\square$

B.2 TERMINATION OF THE EXECUTION INTERVAL ANALYSIS

Also for the execution interval analysis flows it remains to be shown that the maximum response times are a valid convergence criterion, as well as that the analysis flows terminate. For that matter we prove the same lemmas and theorems as in the previous section, only this time for execution intervals.

First we prove that maximum response times are a valid convergence criterion:

Theorem B.4 (Execution Interval Convergence Criterion). *Let $\hat{R}_a^n$, $\hat{\rho}_a^n$, $\mathcal{E}_a^n$ and $\delta_{b,a}^n$ denote maximum response times, maximum firing durations, execution intervals and estimated numbers of initial tokens that are calculated in iterations n of the execution interval analysis flows. If all maximum response times converge, i.e. $\forall_{t_a \in T, n \geq 0}: \hat{R}_a^n \leq \hat{R}_a^{n+1}$, then the same holds for all other parameters.*

Proof. The proof is the same as the proof of Theorem B.1, except that we do not have to show that maximum enabling jitters converge, but the respective execution intervals. Given that the maximum firing durations have converged, i.e. $\forall_{\hat{v}_a \in \hat{V}, n \geq 0}: \hat{\rho}_a^{n+2} = \hat{\rho}_a^{n+1}$, it follows with the same reasoning as for the maximum enabling jitters that also maximum external enabling times converge. As the lower bounds of execution intervals are set to minimum enabling times and as the upper bounds are the sums of maximum external enabling times and maximum firing durations, we can conclude that execution intervals converge on convergence of maximum response times, i.e. $\forall_{t_a \in T, n \geq 0}: \mathcal{E}_a^{n+2} = \mathcal{E}_a^{n+1}$. $\square$

Next we show that the analysis flow is monotone in all its parameters:

Lemma B.2 (Execution Interval Analysis Flow Monotonicity). *Let $\hat{R}_a^n$ denote the maximum response times that are calculated in iterations n of the execution interval analysis flows. It holds that all maximum response times are monotonically increasing in increasing flow iterations, i.e.:*

$$\forall_{n \geq 0, t_a \in T}: \hat{R}_a^n \leq \hat{R}_a^{n+1}$$

Proof. Follows immediately from the clamping of maximum response times. $\square$

With the flow monotonicity we can finally prove flow termination:

Theorem B.5 (Execution Interval Analysis Flow Termination 1). *If the execution interval analysis flows are applied on applications whose task graphs are constrained by maximum end-to-end latencies $\hat{L}$, then the analysis flows terminate, i.e. they either converge or stop on a constraint violation.*

Proof. Analogous to the proof of Theorem B.2. $\square$

Theorem B.6 (Execution Interval Analysis Flow Termination 2). *If the execution interval analysis flow with an iterative buffer sizing is applied on an application whose FIFO buffer capacities are all bounded from above by maximum numbers of initially empty locations $\hat{\delta}_{b,a}$, then the analysis flow terminates.*

Proof. Analogous to the proof of Theorem B.3. $\square$

B.3 VALIDITY OF THE MULTI-PHASE STOP CRITERION

We prove the validity of the stop criterion in line 15 of Figure 12.5 in this section (the stop criterion of the algorithm in Figure 12.7 can be proven analogously). The algorithm is recapitulated in Figure B.1 for convenience. In the following we need to distinguish between the maximum busy periods and maximum finish times determined in different iterations of the algorithm. For that matter let us introduce an index n that we can use to refer to the $\hat{w}'_{a,x'}^{(n)}$, $\hat{w}_{a,x'}^{(n)}$ and $\mathcal{Z}_{a,x'}^{(n)}$ computed in different iterations. We define the relation between x' , $x = x_n$, $q = q_n$ and n as follows:

$$n = q_n \cdot \Theta_a^r + x_n - x'$$

As one can easily see, this definition leads to n being initially zero and increasing by one in each iteration of the while-loop. Using this indexing and taking into account that $x = x'$ must hold for exiting the while-loop we can reformulate the stop criterion more explicitly as follows (with q^* the q for which the stop criterion is met):

$$\hat{w}'_{a,x'}^{(q^* \Theta_a^r - 1)} \leq q^* \cdot P_a \quad (\text{B.1})$$

Note that the term -1 appears as the increase of n to $q^* \Theta_a^r$ (and thus $x = x'$) occurs after the computation of the last maximum busy period. Moreover, the stop criterion for q^* would not be checked if it were already true for a q' with $0 < q' < q^*$. This implies:

$$\forall_{0 < q' < q^*} : \hat{w}'_{a,x'}^{(q' \cdot \Theta_a^r - 1)} > q' \cdot P_a \quad (\text{B.2})$$

In the following we prove that given these two criteria we do not have to consider any $\hat{w}'_{a,x'}^{(q^* \Theta_a^r + k)}$ with $k \geq 0$ as the maximum finish times of task phases cannot become larger for any of these maximum busy periods. We conduct this proof by comparing interference characterizations, then extend these observations to maximum busy periods and finally maximum finish times. We begin with the period-and-jitter only interference characterization.

```

1  $\forall_{0 \leq x < \Theta_a^r} : \hat{f}_{a,x} = 0;$ 
2 forall  $(x' : e_{b,y,a,x'} \in E^{ext})$  {
3    $x = x'; q = 0; \hat{w}'_{a,x'} = \hat{w}_{a,x'} = 0; \mathcal{Z}_{a,x'} = 0;$ 
4   do {
5      $\hat{w}'_{a,x'} = \hat{w}_{a,x'} = 0;$ 
6      $\hat{w}'_{a,x'} = \hat{\chi}_{a,x} + \sum_{t_{b,y} \in hpa} [\eta_{b,y}(\hat{w}'_{a,x'} + \hat{w}'_{a,x'}) - \eta_{b,y}(\hat{w}'_{a,x'})] \cdot \hat{\chi}_{b,y};$ 
7      $\hat{w}'_{a,x'} = \hat{\chi}_{a,x} + \sum_{t_{b,y} \in hpa} [\gamma_{b,y \rightarrow a}(\hat{w}'_{a,x'} + \hat{w}'_{a,x'}, \mathcal{Z}_{a,x'} \cup \{(v_{a,x}, q)\}) - \gamma_{b,y \rightarrow a}(\hat{w}'_{a,x'}, \mathcal{Z}_{a,x'})] \cdot \hat{\chi}_{b,y};$ 
8      $\hat{w}'_{a,x'} = \hat{w}'_{a,x'} + \hat{w}'_{a,x'}; \hat{w}_{a,x'} = \hat{w}_{a,x'} + \hat{w}_{a,x'};$ 
9      $\mathcal{Z}_{a,x'} = \mathcal{Z}_{a,x'} \cup \{(v_{a,x}, q)\};$ 
10     $\hat{f}_{a,x} = \max(\hat{f}_{a,x}, \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'} - q \cdot P_a);$ 
11     $x++;$ 
12    if  $(x = \Theta_a^r)$  {
13       $q++;$   $x = 0;$ 
14    }
15  } while  $(x \neq x' \mid \mid \hat{w}'_{a,x'} > q \cdot P_a);$ 
16 }

```

Figure B.1: Algorithm to compute upper bounds on finish times of task phases using period-and-jitter.

Lemma B.3 (Stop Criterion Validity for Unreduced Maximum Busy Periods). *It holds:*

$$\forall_{k \geq 0}: \quad \eta_{b,y}(\hat{w}'_{a,x'} \langle q^* \Theta_a^r + k \rangle) - \eta_{b,y}(\hat{w}'_{a,x'} \langle q^* \Theta_a^r - 1 \rangle) \leq \eta_{b,y}(\hat{w}'_{a,x'} \langle k \rangle) \wedge$$

$$\hat{w}'_{a,x'} \langle q^* \Theta_a^r + k \rangle - \hat{w}'_{a,x'} \langle q^* \Theta_a^r - 1 \rangle \leq \hat{w}'_{a,x'} \langle k \rangle$$

Proof. With the sub-additivity of the ceiling function $\lceil k+l \rceil \leq \lceil k \rceil + \lceil l \rceil$ it follows with $k = m - n$ and $l = n$ that $\lceil m \rceil - \lceil n \rceil \leq \lceil m - n \rceil$ and with $\hat{J}_{b,y} \geq 0$:

$$\eta_{b,y}(\Delta\tau_1) - \eta_{b,y}(\Delta\tau_2) = \left\lceil \frac{\hat{J}_{b,y} + \Delta\tau_1}{P_a} \right\rceil - \left\lceil \frac{\hat{J}_{b,y} + \Delta\tau_2}{P_a} \right\rceil \quad (\text{B.3})$$

$$\leq \left\lceil \frac{\hat{J}_{b,y} + \Delta\tau_1 - \Delta\tau_2}{P_a} \right\rceil = \eta_{b,y}(\Delta\tau_1 - \Delta\tau_2)$$

By adding up extensions of maximum busy periods according to the algorithm in Figure B.1 it follows that $\hat{w}'_{a,x'} \langle q^* \Theta_a^r + k \rangle - \hat{w}'_{a,x'} \langle q^* \Theta_a^r - 1 \rangle$ is the fixed point of a function $f(\Delta\tau)$ and $\hat{w}'_{a,x'} \langle k \rangle$ the fixed point of a function $g(\Delta\tau)$ with:

$$f(\Delta\tau) = \sum_{k'=0}^k \hat{\chi}_{a,x_{k'}} + \sum_{t_{b,y} \in hpa} [\eta_{b,y}(\hat{w}'_{a,x'} \langle q^* \Theta_a^r - 1 \rangle + \Delta\tau) - \eta_{b,y}(\hat{w}'_{a,x'} \langle q^* \Theta_a^r - 1 \rangle)] \cdot \hat{\chi}_{b,y}$$

$$g(\Delta\tau) = \sum_{k'=0}^k \hat{\chi}_{a,x_{k'}} + \sum_{t_{b,y} \in hpa} \eta_{b,y}(\Delta\tau) \cdot \hat{\chi}_{b,y}$$

Note that in the definition of $f(\Delta\tau)$ we have used $\forall_{0 \leq k' \leq k} : x_{q \cdot \Theta_a^r + k'} = x_{k'}$. From Equation B.3 it immediately follows that $\forall_{\Delta\tau} : f(\Delta\tau) \leq g(\Delta\tau)$. With the monotonicity of both $f(\Delta\tau)$ and $g(\Delta\tau)$ and with $f(0) \geq 0$ one can further conclude according to the reasoning in the proof of Lemma 10.3 that also the fixed point of $f(\Delta\tau)$ must be smaller or equal to the fixed point of $g(\Delta\tau)$, i.e.:

$$\hat{w}'_{a.x'} \langle q^* \Theta_a^r + k \rangle - \hat{w}'_{a.x'} \langle q^* \Theta_a^r - 1 \rangle \leq \hat{w}'_{a.x'} \langle k \rangle$$

And with Equation B.3:

$$\eta_{b.y}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r + k \rangle) - \eta_{b.y}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r - 1 \rangle) \leq \eta_{b.y}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r + k \rangle - \hat{w}'_{a.x'} \langle q^* \Theta_a^r - 1 \rangle) \leq \eta_{b.y}(\hat{w}'_{a.x'} \langle k \rangle)$$

□

Lemma B.3 allows us to prove similar inequalities for the combined interference characterizations, with a restriction that we relax in Lemma B.6.

Lemma B.4 (Stop Criterion Validity for Reduced Maximum Busy Periods). *If it holds $\forall_{t_{b.y} \in hp_a} : \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(q^* \Theta_a^r - 1)}) \geq q^*$ it follows with Equations B.1 and B.2:*

$$\forall_{k \geq 0} : \gamma_{b.y \rightarrow a}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r + k \rangle, \mathcal{Z}_{a.x'}^{(q^* \Theta_a^r + k)}) - \gamma_{b.y \rightarrow a}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r - 1 \rangle, \mathcal{Z}_{a.x'}^{(q^* \Theta_a^r - 1)}) \quad (B.4)$$

$$\leq \gamma_{b.y \rightarrow a}(\hat{w}'_{a.x'} \langle k \rangle, \mathcal{Z}_{a.x'}^{(k)}) \wedge$$

$$\hat{w}'_{a.x'} \langle q^* \Theta_a^r + k \rangle - \hat{w}'_{a.x'} \langle q^* \Theta_a^r - 1 \rangle \leq \hat{w}'_{a.x'} \langle k \rangle \quad (B.5)$$

Proof. Recall that $\gamma_{b.y \rightarrow a}(\Delta\tau, \mathcal{Z})$ is defined in Section 12.4.4 as follows:

$$\gamma_{b.y \rightarrow a}(\Delta\tau, \mathcal{Z}) = \min(\eta_{b.y}(\Delta\tau), \zeta_{b.y \rightarrow a}(\mathcal{Z}))$$

If actors v_a and v_b are not connected via a cycle, for instance because the corresponding tasks belong to different task graphs, the interference characterizations considering cyclic data dependencies $\zeta_{b.y \rightarrow a}(\mathcal{Z})$ all result in infinity and Equation B.4 becomes the upper inequality of Lemma B.3.

However, if both actors are connected via a cycle we have to differ between two cases. In the first case we assume that $\eta_{b.y}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r - 1 \rangle) < \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(q^* \Theta_a^r - 1)})$. Then the left-hand side of Equation B.4 resolves to:

$$\begin{aligned} & \gamma_{b.y \rightarrow a}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r + k \rangle, \mathcal{Z}_{a.x'}^{(q^* \Theta_a^r + k)}) - \gamma_{b.y \rightarrow a}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r - 1 \rangle, \mathcal{Z}_{a.x'}^{(q^* \Theta_a^r - 1)}) \\ &= \min(\eta_{b.y}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r + k \rangle), \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(q^* \Theta_a^r + k)})) - \eta_{b.y}(\hat{w}'_{a.x'} \langle q^* \Theta_a^r - 1 \rangle) \\ &\leq \min(\eta_{b.y}(\hat{w}'_{a.x'} \langle k \rangle), \delta(\mathcal{P}_{a.x_k, b.y}) + \delta(\mathcal{P}_{b.y, a.x'}) + q^* + q_k - 1 - q^*) \\ &= \min(\eta_{b.y}(\hat{w}'_{a.x'} \langle k \rangle), \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(k)})) = \gamma_{b.y \rightarrow a}(\hat{w}'_{a.x'} \langle k \rangle, \mathcal{Z}_{a.x'}^{(k)}) \end{aligned}$$

For the first argument of the minimum function we have used Lemma B.3 and for the second argument we have substituted $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(q^* \Theta_a^r + k)})$ using Equation 12.4, which is:

$$\zeta_{b.y \rightarrow a}(\mathcal{Z}) = \delta(\mathcal{P}_{a.\hat{x}, b.y}) + \hat{q} + \delta(\mathcal{P}_{b.y, a.\hat{x}}) - \check{q} - 1 \quad (B.6)$$

To that end we have taken into account that $\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle}$ has the following suprema and infima with respect to the ordering relation defined in Section 12.4.4:

$$\begin{aligned}(v_{a.\hat{x}}, \hat{q}) &= (v_{a.x_{q^* \Theta_a^r + k}}, q_{q^* \Theta_a^r + k}) = (v_{a.x_k}, q^* + q_k) \\ (v_{a.\hat{x}}, \check{q}) &= (v_{a.x'}, 0)\end{aligned}$$

And with Equation B.2 and $P_a = P_b$ (which is implied by actors v_a and v_b being on a cycle, i.e. their corresponding tasks belonging to the same task graph) we have further concluded for the second argument (with $w = \hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle} - \hat{w}_{a.x'}^{\langle (q^* - 1) \Theta_a^r - 1 \rangle} > 0$):

$$\begin{aligned}\eta_{b.y}(\hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) &= \eta_{b.y}(\hat{w}_{a.x'}^{\langle (q^* - 1) \Theta_a^r - 1 \rangle} + w) \geq \eta_{b.y}((q^* - 1) \cdot P_a + w) \\ &= \left\lceil \frac{\hat{J}_{b.y} + (q^* - 1) \cdot P_a + w}{P_b} \right\rceil = \left\lceil \frac{\hat{J}_{b.y} + w}{P_b} \right\rceil + q^* - 1 \geq q^*\end{aligned}$$

In the second case we assume that $\eta_{b.y}(\hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) \geq \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle})$. Then the left-hand side of Equation B.4 resolves to:

$$\begin{aligned}&\gamma_{b.y \rightarrow a}(\hat{w}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle}, \mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle}) - \gamma_{b.y \rightarrow a}(\hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}, \mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) \\ &= \min(\eta_{b.y}(\hat{w}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle}), \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle})) - \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) \\ &\leq \min(q^* + \eta_{b.y}(\hat{w}_{a.x'}^{\langle k \rangle}), \delta(\mathcal{P}_{a.x_k, b.y}) + \delta(\mathcal{P}_{b.y, a.x'}) + q^* + q_k - 1) - q^* \\ &= \min(\eta_{b.y}(\hat{w}_{a.x'}^{\langle k \rangle}), \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle k \rangle})) = \gamma_{b.y \rightarrow a}(\hat{w}_{a.x'}^{\langle k \rangle}, \mathcal{Z}_{a.x'}^{\langle k \rangle})\end{aligned}$$

For the first argument of the minimum function we have used Lemma B.3, the subadditivity of the ceiling function, Equation B.1 and $P_a = P_b$ such that:

$$\begin{aligned}\eta_{b.y}(\hat{w}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle}) &\leq \eta_{b.y}(\hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle} + \hat{w}_{a.x'}^{\langle k \rangle}) \\ &\leq \left\lceil \frac{\hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}}{P_b} \right\rceil + \left\lceil \frac{\hat{J}_{b.y} + \hat{w}_{a.x'}^{\langle k \rangle}}{P_b} \right\rceil \\ &\leq \left\lceil \frac{q^* \cdot P_a}{P_b} \right\rceil + \eta_{b.y}(\hat{w}_{a.x'}^{\langle k \rangle}) = q^* + \eta_{b.y}(\hat{w}_{a.x'}^{\langle k \rangle})\end{aligned} \tag{B.7}$$

For the second argument we have again applied the substitution via Equation B.6. And for the term outside the minimum function we have used the precondition that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) \geq q^*$.

This lets us conclude that Equation B.4 holds for all $t_{b.y} \in hp_a$ if $\forall t_{b.y} \in hp_a : \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) \geq q^*$. In words this means that the differences between the interference characterizations of $\hat{w}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle}$ and $\hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}$ are smaller or equal to the interference characterizations of $\hat{w}_{a.x'}^{\langle k \rangle}$, for all $t_{b.y} \in hp_a$.

Finally you can see from the algorithm in Figure B.1 that if Equation B.4 holds for all $t_{b.y} \in hp_a$ also the difference between the maximum busy periods $\hat{w}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle}$ and $\hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}$ must be smaller than the maximum busy period $\hat{w}_{a.x'}^{\langle k \rangle}$, i.e. Equation B.5 holds as well. $\square$

These lemmas allow us to establish a relation between the maximum finish time computations in algorithm iterations $q^* \Theta_a^r + k$ and k :

Lemma B.5 (Stop Criterion Validity for the Maximum Function). *If it holds $\forall t_{b.y} \in hp_a : \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) \geq q^*$ it follows with Equations B.1 and B.2 for the arguments of the maximum function in line 10 of the algorithm that:*

$$\forall k \geq 0: \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle} - q_{q^* \Theta_a^r + k} \cdot P_a \leq \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^{\langle k \rangle} - q_k \cdot P_a$$

Proof. Using Lemma B.4, $q_{q^* \Theta_a^r + k} = q^* + q_k$ and Equation B.1 it follows:

$$\begin{aligned} \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^{\langle q^* \Theta_a^r + k \rangle} - q_{q^* \Theta_a^r + k} \cdot P_a &\leq \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle} + \hat{w}_{a.x'}^{\langle k \rangle} - (q^* + q_k) \cdot P_a \\ &\leq \hat{s}_{a.x'}^{ext} + q^* \cdot P_a + \hat{w}_{a.x'}^{\langle k \rangle} - (q^* + q_k) \cdot P_a \\ &= \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^{\langle k \rangle} - q_k \cdot P_a \end{aligned}$$

$\square$

Now we show that Lemma B.4 and therewith also Lemma B.5 holds even if the restriction $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) \geq q^*$ is not true for all $t_{b.y} \in hp_a$:

Lemma B.6 (General Stop Criterion Validity for the Maximum Function). *Lemma B.5 applies even if it holds for some $t_{b.y} \in hp_a$ that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) < q^*$.*

Figure B.2: Special case $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) < q^*$.

Proof. First assume that it holds for only one $t_{b.y} \in hp_a$ that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) < q^*$. From Equation B.6 and the ordering relation in Section 12.4.4 it follows that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle n \rangle})$ increases by one if n is increased by Θ_a^r . Thus it holds that:

$$\forall q' \geq 1: \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle (q'+1) \Theta_a^r - 1 \rangle}) = \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q' \Theta_a^r - 1 \rangle}) + 1 = \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle \Theta_a^r - 1 \rangle}) + q'$$

From $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(q^* \Theta_a^{r-1})}) < q^*$ it therefore follows that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}) < 1$ and thus $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}) = 0$ must hold.

If $v_{a.x'}$ is not the first phase of actor v_a (i.e. $0 < x' < \Theta_a^r$) then q must have already become one before iteration $\Theta_a^r - 1$ of the algorithm in Figure B.1. With $v_{a.x}$ the immediate predecessor of $v_{a.x'}$ this implies that the supremum of $\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}$ is $(v_{a.\hat{x}}, \hat{q}) = (v_{a.x}, 1)$, whereas the infimum of $\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}$ is $(v_{a.\hat{x}}, \hat{q}) = (v_{a.x'}, 0)$. According to Equation B.6 it then follows with $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}) = 0$:

$$\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}) = \delta(\mathcal{P}_{a.x,b.y}) + \delta(\mathcal{P}_{b.y,a.x'}) + 1 - 1 = 0$$

This equation can only be true if both $\delta(\mathcal{P}_{a.x,b.y})$ and $\delta(\mathcal{P}_{b.y,a.x'})$ are zero, as depicted in Figure B.2(a).

If $v_{a.x'}$ is the first phase of actor v_a (i.e. $x' = 0$) then q just becomes one at the end of algorithm iteration $\Theta_a^r - 1$. This implies that the supremum of $\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}$ is $(v_{a.\hat{x}}, \hat{q}) = (v_{a.x}, 0)$ and it follows with $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}) = 0$:

$$\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(\Theta_a^{r-1})}) = \delta(\mathcal{P}_{a.x,b.y}) + \delta(\mathcal{P}_{b.y,a.x'}) - 1 = 0$$

In this case the equation can only be true if either $\delta(\mathcal{P}_{a.x,b.y})$ or $\delta(\mathcal{P}_{b.y,a.x'})$ is one and the other zero, as depicted in Figure B.2(b).

In both cases (a) and (b) it holds that $v_{a.x}$ and $v_{a.x'}$ can never be in consecutive execution as $v_{b.y}$ is always executed in between. This intermediate execution of $v_{b.y}$ is however already conservatively considered in the worst-case LP from Section 12.3, which is:

$$\begin{aligned} \text{Minimize: } & \sum_{\hat{v}_{a.x} \in \hat{V}} \hat{s}_{a.x}^{ext} + \hat{s}_{a.x} \\ \text{Subject to: } & \hat{s}_{s,0} = 0, \quad \forall_{e_{a.x,b.y} \in \hat{E}^{ext}}: \quad \hat{s}_{b.y}^{ext} - \hat{s}_{a.x} \geq \hat{\rho}_{a.x} - \delta_{a.x,b.y} \cdot P_b, \\ & \forall_{e_{a.x,b.y} \in \hat{E}}: \quad \hat{s}_{b.y} - \hat{s}_{a.x} \geq \hat{\rho}_{a.x} - \delta_{a.x,b.y} \cdot P_b \end{aligned}$$

Hence it holds for case (a) that $\hat{s}_{b.y} \geq \hat{s}_{a.x} + \hat{\rho}_{a.x}$ and $\hat{s}_{a.x'}^{ext} \geq \hat{s}_{b.y} + \hat{\rho}_{b.y}$ and for case (b) with $P_a = P_b$ that $\hat{s}_{b.y} \geq \hat{s}_{a.x} + \hat{\rho}_{a.x} - P_a$ and $\hat{s}_{a.x'}^{ext} \geq \hat{s}_{b.y} + \hat{\rho}_{b.y}$ or $\hat{s}_{b.y} \geq \hat{s}_{a.x} + \hat{\rho}_{a.x}$ and $\hat{s}_{a.x'}^{ext} \geq \hat{s}_{b.y} + \hat{\rho}_{b.y} - P_a$. From this follows that $\hat{s}_{a.x'}^{ext} \geq \hat{f}_{a.x} + \hat{\chi}_{b.y}$ for case (a) and $\hat{s}_{a.x'}^{ext} \geq \hat{f}_{a.x} + \hat{\chi}_{b.y} - P_a$ for case (b).

With $\hat{f}_{a.x} \geq \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^{(\Theta_a^{r-1})} - P_a$ in case (a) and $\hat{f}_{a.x} \geq \hat{s}_{a.x'}^{ext} + \hat{w}_{a.x'}^{(\Theta_a^{r-1})}$ in case (b) it further holds for both cases that $\hat{w}_{a.x'}^{(\Theta_a^{r-1})} + \hat{\chi}_{b.y} \leq P_a$, which implies that the stop criterion is already met for $q^* = 1$.

Now assume that it holds for multiple $t_{b.y} \in hp_a$ that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{(q^* \Theta_a^{r-1})}) < q^*$. Figure B.3 depicts case (a) for two such phases $t_{b.y}, t_{c.z} \in hp_a$ (case (b) and more than two phases can be constructed analogously).

In case (i) the two phases are always firing one after the other. Thus it follows from the worst-case LP that $\hat{s}_{b.y} \geq \hat{s}_{a.x} + \hat{\rho}_{a.x}$, $\hat{s}_{c.z} \geq \hat{s}_{b.y} + \hat{\rho}_{b.y}$ and $\hat{s}_{a.x'}^{ext} \geq$

Figure B.3: Case (a) for $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) < q^*$ and $\zeta_{c.z \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) < q^*$.

$\hat{s}_{c.z} + \hat{\rho}_{c.z}$ from which we conclude with the same reasoning as above that $\hat{w}_{a.x'}^{\langle \Theta_a^r - 1 \rangle} + \hat{\chi}_{b.y} + \hat{\chi}_{c.z} \leq P_a$.

In case (ii) the two phases are firing in parallel. Thus it follows from the worst-case LP that $\hat{s}_{b.y} \geq \hat{s}_{a.x} + \hat{\rho}_{a.x}$, $\hat{s}_{c.z} \geq \hat{s}_{a.x} + \hat{\rho}_{a.x}$ and $\hat{s}_{a.x}^{ext} \geq \max(\hat{s}_{b.y} + \hat{\rho}_{b.y}, \hat{s}_{c.z} + \hat{\rho}_{c.z})$. From this we derive with above reasoning that $\hat{w}_{a.x'}^{\langle \Theta_a^r - 1 \rangle} + \max(\hat{\rho}_{b.y}, \hat{\rho}_{c.z}) \leq P_a$. Additionally we know that the phases $v_{b.y}$ and $v_{c.z}$ cannot be on a cycle with one token (otherwise we would have case (i) again), which implies that the phases can interfere with each other. As one of the two phases must have a higher priority than the other it follows that either $\hat{\rho}_{b.y} \geq \hat{\chi}_{b.y} + \hat{\chi}_{c.z}$ or $\hat{\rho}_{c.z} \geq \hat{\chi}_{b.y} + \hat{\chi}_{c.z}$. This lets us conclude again that $\hat{w}_{a.x'}^{\langle \Theta_a^r - 1 \rangle} + \hat{\chi}_{b.y} + \hat{\chi}_{c.z} \leq P_a$.

This observation can be generalized as follows. Let hp'_a be the set of all $t_{b.y} \in hp_a$ for which it holds that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle q^* \Theta_a^r - 1 \rangle}) < q^*$. If the set is not empty it holds that the stop criterion is already met for $q^* = 1$ and it holds:

$$\hat{w}_{a.x'}^{\langle \Theta_a^r - 1 \rangle} + \sum_{t_{b.y} \in hp'_a} \hat{\chi}_{b.y} \leq P_a \quad (\text{B.8})$$

Moreover it holds for all $t_{b.y} \in hp'_a$:

$$\begin{aligned} & [\gamma_{b.y \rightarrow a}(\hat{w}_{a.x'}^{\langle \Theta_a^r + k \rangle}, \mathcal{Z}_{a.x'}^{\langle \Theta_a^r + k \rangle}) - \gamma_{b.y \rightarrow a}(\hat{w}_{a.x'}^{\langle \Theta_a^r - 1 \rangle}, \mathcal{Z}_{a.x'}^{\langle \Theta_a^r - 1 \rangle})] \cdot \hat{\chi}_{b.y} \\ &= [\min(\eta_{b.y}(\hat{w}_{a.x'}^{\langle \Theta_a^r + k \rangle}), \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle \Theta_a^r + k \rangle})) - 0] \cdot \hat{\chi}_{b.y} \\ &\leq [\min(1 + \eta_{b.y}(\hat{w}_{a.x'}^{\langle k \rangle}), \zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle k \rangle}) + 1)] \cdot \hat{\chi}_{b.y} \\ &= [\gamma_{b.y \rightarrow a}(\hat{w}_{a.x'}^{\langle k \rangle}, \mathcal{Z}_{a.x'}^{\langle k \rangle}) + 1] \cdot \hat{\chi}_{b.y} \end{aligned}$$

For the first argument of the minimum function we have used Equation B.7 from the proof of Lemma B.4, for the second argument that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle n \rangle})$ increases by one if n is increased by Θ_a^r and for $\gamma_{b.y \rightarrow a}(\hat{w}_{a.x'}^{\langle \Theta_a^r - 1 \rangle}, \mathcal{Z}_{a.x'}^{\langle \Theta_a^r - 1 \rangle})$ that $\zeta_{b.y \rightarrow a}(\mathcal{Z}_{a.x'}^{\langle \Theta_a^r - 1 \rangle})$ equals zero.

From this and Lemma B.4 for the other $t_{b.y} \in hp_a \setminus hp'_a$ it follows:

$$\hat{w}_{a.x'}^{\langle \Theta_a^r + k \rangle} - \hat{w}_{a.x'}^{\langle \Theta_a^r - 1 \rangle} \leq \hat{w}_{a.x'}^{\langle k \rangle} + \sum_{t_{b.y} \in hp'_a} \hat{\chi}_{b.y} \quad (\text{B.9})$$

If $hp'_a \neq \emptyset$ (i.e. $\exists_{t_{b,y} \in hp_a} : \zeta_{b,y \rightarrow a}(\mathcal{Z}_{a,x'}^{(q^* \Theta_a^r - 1)}) < q^*$) we can therefore conclude with Equation B.8 that $q^* = 1$ and with $q_{\Theta_a^r + q_k} = 1 + q_k$ and Equations B.8 and B.9 that for all $k \geq 0$:

$$\begin{aligned} \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{(\Theta_a^r + k)} - q_{\Theta_a^r + q_k} \cdot P_a &\leq \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{(\Theta_a^r - 1)} + \hat{w}_{a,x'}^{(k)} + \sum_{t_{b,y} \in hp'_a} \hat{\chi}_{b,y} - (1 + q_k) \cdot P_a \\ &\leq \hat{s}_{a,x'}^{ext} + P_a + \hat{w}_{a,x'}^{(k)} - (1 + q_k) \cdot P_a \\ &= \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{(k)} - q_k \cdot P_a \end{aligned}$$

□

This allows us to finally prove that when the stop criterion is satisfied no maximum finish times can increase thereafter:

Theorem B.7 (Stop Criterion Validity for Maximum Finish Times). *It holds that no maximum finish time $\hat{f}_{a,x}$ computed in line 10 of the algorithm can increase after the stop criterion from Equation B.1 is met.*

Proof. Let us define any $k \geq 0$ as $k = l \cdot q^* \Theta_a^r + k'$, with $l \geq 0$ and $0 \leq k' \leq q^* \Theta_a^r - 1$. With Lemma B.6 we obtain by repetitively applying Lemma B.5:

$$\begin{aligned} \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{(q^* \Theta_a^r + k)} - q_{q^* \Theta_a^r + k} \cdot P_a &= \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{((l+1) \cdot q^* \Theta_a^r + k')} - q_{(l+1) \cdot q^* \Theta_a^r + k'} \cdot P_a \\ &\leq \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{(l \cdot q^* \Theta_a^r + k')} - q_{l \cdot q^* \Theta_a^r + k'} \cdot P_a \\ &\leq \dots \leq \hat{s}_{a,x'}^{ext} + \hat{w}_{a,x'}^{(k')} - q_{k'} \cdot P_a \end{aligned}$$

This means that for any term in the maximum function in line 10 of the algorithm that is considered in algorithm iteration $q^* \Theta_a^r + k$ there must exist a corresponding larger or equal term in iteration $k' \leq q^* \Theta_a^r - 1$. Furthermore, both such terms are used to determine the same maximum finish time $\hat{f}_{a,x}$ as it holds by definition that $k - k'$ is a multiple of Θ_a^r , i.e. both terms refer to the same phase $x_k = x_{k'}$. From this follows immediately that no maximum finish time $\hat{f}_{a,x}$ can increase after iteration $q^* \Theta_a^r - 1$ of the algorithm, i.e. after the stop criterion is met. □

This allows us to finally conclude that the stop criterion is valid.

List of Acronyms

ABS	Anti-Lock Braking System	4
API	Application Programming Interface	241
BCET	Best-Case Execution Time	105
CCS	Calculus of Communicating Systems	122
CP	Communicating Processes	121
CPS	Cyber-Physical System	28
CSDF	Cyclo-Static Dataflow	60
DMA	Direct Memory Access	165
ECU	Electronic Control Unit	4
FFT	Fast Fourier Transformation	164
FIFO	First-In First-Out	5
HAPI	Hijdra API	116
HSDF	Homogeneous Synchronous Dataflow	60
IoT	Internet of Things	6
KPN	Kahn Process Network	27
LP	Linear Program	109
MAST	Modeling and Analysis Suite for Real-Time Applications	120
MCM	Maximum Cycle Mean	155
MPA	Modular Performance Analysis	119
POOSL	Parallel Object Oriented Specification Language	122
PV	Programmer's View	121
RR	Round-Robin	15

RTC Real-Time Calculus 119

SDF Synchronous Dataflow 47

SDR Software-Defined Radio 15

SoC System-on-Chip 5

SPP Static Priority Preemptive 15

TDM Time-Division Multiplexing 15

TETB The-Earlier-the-Better 29

TLM Transaction Level Modeling 121

TLTB The-Later-the-Better 73

TPLTS Timed Probabilistic Labeled Transition System 122

VRDF Variable-Rate Dataflow 63

WCET Worst-Case Execution Time 105

WLAN Wireless Local Area Network 6

YAPI Y-chart Application Programmer’s Interface 122

List of Symbols

Notation	Description	Page
$ n $	Length of / number of elements in n	
$\lfloor n \rfloor$	Floor function, largest integer m with $m \leq n$	
$\lceil n \rceil$	Ceiling function, smallest integer m with $m \geq n$	
$n \% m$	Modulus operator, remainder of integer division n / m	
$\check{n}$	Lower bound on / best-case / minimum n	
$\hat{n}$	Upper bound on / worst-case / maximum n	
n_a	Property n of actor v_a or task t_a	
$n_{a.x}$	Property n of actor phase $v_{a.x}$ or task phase $t_{a.x}$	
$n_a(i)$	Property n of iteration i of task t_a or actor v_a	
$n_{a.x}(i)$	Property n of iteration i of phase $t_{a.x}$ or $v_{a.x}$	
$\mathbb{N}$	Set of non-negative natural numbers	41
$\mathbb{Z}$	Set of integer numbers	85
$\mathbb{Z}^+$	Set of positive integer numbers	85
$\mathbb{Z}^-$	Set of negative integer numbers	85
$\mathbb{R}_0^+$	Set of positive real numbers, including zero	43
$\tau_x(i)$	Timestamp of location i in stream / signal x	41
$\mathcal{T}$	Time domain	41
$\vartheta_x(i)$	Value of location i in stream / signal x	41
$\mathcal{O}$	Value domain	41
$\models$	Value relation	41
$St(p)$	Set of all allowed streams / signals on port p	42
$Tr(P)$	Set of all allowed traces on interface P	44

Notation	Description	Page
$\leq$	Prefix ordering relation	42
$\rightarrow$	Port connection allowed	42
$\rightarrow_{\Xi}$	Interface connection according to mapping Ξ allowed	44
C^{Ξ}	Interface connection according to mapping Ξ	45
R_A	Relation of timed component A	46
in_A	Input set of timed component A	46
out_A	Output set of timed component A	46
$A B$	Parallel composition of components A and B	47
$A\Xi B$	Serial composition of components A and B according to interface mapping Ξ (feedback if $A\Xi A$)	47
$\doteq$	Partial trace equality	72
$\triangleleft$	Stream / trace bounding	63
$\triangleleft$	Partial trace bounding	64
$\triangleleft$	Best-case component bounding	32
$\triangleleft$	Worst-case component bounding	32
$\propto$	Signal / trace approximation	263
$\propto$	Partial trace approximation	264
$\underline{\propto}$	Best-case component approximation	266
$\overline{\propto}$	Worst-case component approximation	266
G^A	Application model	124
G^{RT}	Response time analysis model	132
G^{BC}	Best-case analysis model	134
G^{WC}	Worst-case analysis model	134
G^{PC}	Precedence constraint analysis model	137
G^T	Task graph	124
T	Set of tasks	124
B	Set of FIFO buffers	124
D^D	Set of data dependencies	124
D^R	Set of resource dependencies	124
t_a	Task with name a	107
$t_{a,x}$	Phase x of task t_a	112
$\beta_{a,b}$	FIFO buffer from task t_a to task t_b	124
P_a	Period of task t_a	89
χ_a	Execution time of task t_a	87
$\mathcal{I}_a$	Interference function of task t_a	89
ε_a	Enabling time of task t_a	89
ε_a^{ext}	External enabling time of task t_a	105
f_a	Finish time of task t_a	89

Notation	Description	Page
J_a	Jitter of task t_a	89
hp_a	Set of higher priority tasks of task t_a	125
w_a	Busy period of task t_a	109
R_a	Response time of task t_a	104
L	End-to-end latency of task graph	84
Υ	Throughput of task graph	153
G^{DF}	Dataflow graph	58
V	Set of actors	58
E	Set of edges	58
v_a	Dataflow actor with name a	58
$v_{a.x}$	Phase x of actor v_a	204
$e_{a,b}$	Edge from actor v_a to actor v_b	58
$\kappa_{a,b}$	Token component between actors v_a and v_b	59
$\delta_{a,b}$	Number of initial tokens on edge $e_{a,b}$	58
ρ_a	Firing duration of actor v_a	58
s_a	Start time of actor v_a	143
s_a^{ext}	External start time of actor v_a	206
Θ_a	Number of phases of CSDF actor v_a	58
r_a	Replication factor of CSDF actor v_a in HSDF expansion	136
Θ_a^r	Number of phases in HSDF expansion of actor v_a	136
$p_{a,b}$	Path of dataflow edges from actor v_a to actor v_b	146
$\mathcal{P}_{a,b}$	Set of paths of edges from actor v_a to actor v_b	146
$\delta(p)$	Number of initial tokens on path p	147
$\delta(\mathcal{P})$	Minimum number of initial tokens on any path $p \in \mathcal{P}$	147
$\delta_{a,b}^\circ$	Number of tokens on cycle between actors v_a and v_b	149
$\mathcal{Z}$	Set containing tuples $(v_{a.x}, q)$, used in multi-phase maximum busy period computation	212
$\mathcal{E}_a(i)$	Execution interval of task t_a in iteration i	122
$P_{v_b \rightarrow v_a(i)}$	Set of iterations of actor v_b that can interfere with $v_a(i)$ according to precedence constraints	148
$E_{t_b \rightarrow t_a(i)}$	Set of iterations of task t_b that can interfere with $t_a(i)$ according to execution intervals	185
$I_{t_b \rightarrow t_a(i)}$	Set of iterations of task t_b that can interfere with $t_a(i)$	187
η_b	Period-and-jitter interference characterization	121
$\zeta_{b \rightarrow a}$	Cyclic data dependency interference characterization	121
$\gamma_{b \rightarrow a}$	Combined period-and-jitter and cyclic data dependency interference characterization	121
$\psi_{b \rightarrow a}$	Execution interval interference characterization	121

Bibliography

- [AD94] R. Alur and D. Dill. A theory of timed automata. *Journal of Theoretical Computer Science*, 126(2):183–235, 1994.
- [AHG07] P. Alexander, D. Haley, and A. Grant. Outdoor mobile broadband access with 802.11. *IEEE Communications Magazine*, 45(11):108–114, 2007.
- [BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996.
- [But11] G. Buttazzo. *Hard real-time computing systems*. Springer US, 2011.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.
- [CKH16] J. Choi, D. Kang, and S. Ha. Conservative modeling of shared resource contention for dependent tasks in partitioned multi-core systems. In *Design, Automation and Test in Europe (DATE)*, pages 181–186, 2016.
- [D⁺10] A. David et al. Timed I/O automata: A complete specification theory for real-time systems. In *ACM Int’l Conf. on Hybrid Systems: Computation and Control (HSCC)*, pages 91–100. ACM, 2010.
- [dAH01a] L. de Alfaro and T. Henzinger. Interface automata. In *European Software Engineering Conf. & ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM, 2001.
- [dAH01b] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *ACM Int’l Conf. on Embedded Software (EMSOFT)*, pages 148–165. Springer, Berlin, Heidelberg, 2001.
- [dAHS02] L. de Alfaro, T. Henzinger, and M. Stoelinga. Timed interfaces. In *ACM Int’l Workshop on Embedded Software (EMSOFT)*, pages 108–122. Springer, Berlin, Heidelberg, 2002.

- [dK⁺00] E. de Kock et al. YAPI: Application modeling for signal processing systems. In *Design Automation Conf. (DAC)*, pages 402–405. ACM, 2000.
- [Don04] A. Donlin. Transaction level modeling: Flows and use models. In *IEEE/ACM/IFIP Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 75–80. ACM, 2004.
- [Dur97] W. Durant. *Our Oriental Heritage*. Fine Communications, 1997.
- [DWBS13] B. Dekens, P. Wilmanns, M. Bekooij, and G. Smit. Low-cost guaranteed-throughput communication ring for real-time streaming MPSoCs. In *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, 2013.
- [F⁺08] J. Falk et al. A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 189–198, 2008.
- [Flo62] R. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [GHGGPGDM01] M. Gonzalez Harbour, J. Gutierrez Garcia, J. Palencia Gutierrez, and J. Drake Moyano. MAST: Modeling and analysis suite for real time applications. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 125–134, 2001.
- [GTW11] M. Geilen, S. Tripakis, and M. Wiggers. The earlier the better: A theory of timed actor interfaces. In *ACM Int'l Conf. on Hybrid Systems: Computation and Control (HSCC)*, pages 23–32, 2011.
- [H⁺05] R. Henia et al. System level performance analysis – the SymTA/S approach. *IEE Proc. of Computers and Digital Techniques*, 152(2):148–166, 2005.
- [Har15] Y. Harari. *Sapiens: A Brief History of Humankind*. Harper, 1st edition edition, 2015.
- [HB16] J. Hausmans and M. Bekooij. A refinement theory for timed dataflow analysis with support for reordering. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, 2016.
- [HBC11] J. Hausmans, M. Bekooij, and H. Corporaal. Resynchronization of cyclo-static dataflow graphs. In *Design, Automation and Test in Europe (DATE)*, pages 1–6, 2011.
- [HE06] R. Henia and R. Ernst. Improved offset-analysis using multiple timing-references. In *Design, Automation and Test in Europe (DATE)*, pages 450–455, 2006.

- [HGWB14] J. Hausmans, S. Geuns, M. Wiggers, and M. 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, 2014.
- [HWGB13] J. Hausmans, M. Wiggers, S. Geuns, and M. Bekooij. Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 13–22, 2013.
- [Jen] E. Jensen. GTKWave - Real-time for the real world.
- [Jer05] M. Jersak. *Compositional performance analysis for complex embedded applications*. PhD thesis, Braunschweig University of Technology, 2005.
- [K⁺13] J. Kim et al. A novel analytical method for worst case response time estimation of distributed embedded systems. In *Design Automation Conf. (DAC)*, pages 129:1–129:10, 2013.
- [Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress*, pages 471–475, 1974.
- [KCH17] D. Kang, J. Choi, and S. Ha. Worst case delay analysis of shared resource access in partitioned multi-core systems. In *IEEE Symp. on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 84–92, 2017.
- [L⁺13] C. Lin et al. Timing analysis of process graphs with finite communication buffers. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 227–236, 2013.
- [Lew51] K. Lewin. *Field theory in social science: Selected theoretical papers*. Harper, 1951.
- [Liu05] X. Liu. *Semantic Foundation of the Tagged Signal Model*. PhD thesis, University of California at Berkeley, 2005.
- [LMC12] A. Lele, O. Moreira, and P. Cuijpers. A new data flow analysis model for TDM. In *Proc. of the Tenth ACM Int'l Conf. on Embedded Software*, pages 237–246. ACM, 2012.
- [LP95] E. Lee and T. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, 1995.
- [LS15] E. Lee and S. Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. <http://leeseshia.org>, 2nd edition, 2015.
- [MBGS10] O. Moreira, T. Basten, M. Geilen, and S. Stuijk. Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Trans. on Computers*, 59(2):188–201, 2010.

- [Mil71] R. Milner. An algebraic definition of simulation between programs. In *Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, pages 481–489, 1971.
- [OSC] OSCI. SystemC.
- [P⁺08] S. Perathoner et al. Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Design Automation for Embedded Systems*, 13(1-2):27–49, 2008.
- [PGH98] J. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 26–37, 1998.
- [PGH99] J. Palencia and M. Gonzalez Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 328–339, 1999.
- [Pla17] Plato. *Apology*. CreateSpace Independent Publishing Platform, 2017.
- [PVB05] H. Posadas, E. Villar, and F. Blasco. Real-time operating system modeling in SystemC for HW/SW cosimulation. In *Proc. Conf. on Design of Circuits and Integrated Systems*. Citeseer, 2005.
- [Red04] O. Redell. Analysis of tree-shaped transactions in distributed real time systems. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 239–248, 2004.
- [RRE03] K. Richter, R. Racu, and R. Ernst. Scheduling analysis integration for heterogeneous multiprocessor SoC. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 236–245, 2003.
- [SB09a] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2nd edition, 2009.
- [SB09b] J. Staschulat and M. Bekooij. Dataflow models for shared memory access latency analysis. In *Proc. of the Seventh ACM Int'l Conf. on Embedded Software*, pages 275–284. ACM, 2009.
- [SBGC07] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conf. (DAC)*, pages 777–782, 2007.
- [SBW09] M. Steine, M. Bekooij, and M. Wiggers. A priority-based budget scheduler with conservative dataflow model. In *2009 12th Euromicro Conf. on Digital System Design, Architectures, Methods and Tools*, pages 37–44, 2009.
- [SE11] S. Schliecker and R. Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):22:1–22:27, 2011.
- [SGB06] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conf. (DAC)*, pages 899–904, 2006.

- [SGB08] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. on Computers*, 57(10):1331–1345, 2008.
- [SMvB14] H. Salunkhe, O. Moreira, and K. van Berkel. Buffer allocation for real-time streaming on a multi-processor without back-pressure. In *IEEE Symp. on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, 2014.
- [Syn] Synopsis. Platform architect.
- [T⁺07] B. Theelen et al. Software/Hardware engineering with the parallel object-oriented specification language. In *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, pages 139–148, 2007.
- [TBW94] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.
- [Tin94] K. Tindell. *Adding time-offsets to schedulability analysis*. University of York, Department of Computer Science, 1994.
- [TLHL09] S. Tripakis, B. Lickly, T. Henzinger, and E. Lee. On relational interfaces. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 67–76, 2009.
- [TS09] L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 127–136, 2009.
- [TWS06] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 34–43. ACM, 2006.
- [WBGB10] M. Wiggers, M. Bekooij, M. Geilen, and T. Basten. Simultaneous budget and buffer size computation for throughput-constrained task graphs. In *Design, Automation and Test in Europe (DATE)*, pages 1669–1672, 2010.
- [WBS07a] M. Wiggers, M. Bekooij, and G. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Design Automation Conf. (DAC)*, pages 658–663, 2007.
- [WBS07b] M. Wiggers, M. Bekooij, and G. 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, 2007.
- [WBS08] M. Wiggers, M. Bekooij, and G. Smit. Computation of buffer capacities for throughput constrained and data dependent inter-task communication. In *Design, Automation and Test in Europe (DATE)*, pages 640–645, 2008.

- [WBS09] M. Wiggers, M. Bekooij, and G. Smit. Monotonicity and run-time scheduling. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 177–186, 2009.
- [WTVL06] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: A case study. *Int'l Journal on Software Tools for Technology Transfer*, 8(6):649–667, 2006.
- [Yu10] K. Yu. *Real-time operating system modelling and simulation using SystemC*. PhD thesis, University of York, 2010.
- [YW98] T. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Trans. on Parallel and Distributed Systems*, 9(11):1125–1136, 1998.

List of Publications

- [PK:1] P. Kurtin. *An abstraction-refinement theory for the analysis and design of concurrent real-time systems*. PhD thesis, University of Twente, Enschede, The Netherlands, 2018.
- [PK:2] G. Kuiper, P. Kurtin, and M. Bekooij. Hybrid latency minimization approach using model checking and dataflow analysis. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 41–50, 2017.
- [PK:3] P. Kurtin and M. Bekooij. An abstraction-refinement theory for the analysis and design of real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):173:1–173:20, 2017.
- [PK:4] P. Kurtin and M. Bekooij. An abstraction-refinement theory for the analysis and design of real-time systems (Extended version). Technical report, Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede, The Netherlands, 2017.
- [PK:5] P. Kurtin and M. Bekooij. Temporal analysis of static priority preemptive scheduled cyclic streaming applications using CSDF models. In *IEEE Symp. on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, 2016.
- [PK:6] P. Kurtin and M. Bekooij. Appendix to temporal analysis of static priority preemptive scheduled cyclic streaming applications using CSDF models. Technical report, Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede, The Netherlands, 2016.
- [PK:7] P. Kurtin, J. Hausmans, and M. Bekooij. HAPI: An event-driven simulator for real-time multiprocessor systems. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 60–66, 2016.
- [PK:8] P. Kurtin, J. Hausmans, and M. 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, 2016.

- [PK:9] P. Wilmanns, S. Geuns, J. Hausmans, and M. Bekooij. Buffer sizing to reduce interference and increase throughput of real-time stream processing applications. In *IEEE Int'l Symp. on Real-Time Computing (ISORC)*, pages 9–18, 2015.
- [PK:10] P. Wilmanns, J. Hausmans, S. Geuns, and M. 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)*, pages 623–630, 2014.
- [PK:11] B. Dekens, P. Wilmanns, G. Smit, and M. Bekooij. Low-cost guaranteed-throughput dual-ring communication infrastructure for heterogeneous MPSoCs. In *Proc. of the 2014 Conf. on Design and Architectures for Signal and Image Processing*, pages 1–8, 2014.
- [PK:12] B. Dekens, P. Wilmanns, M. Bekooij, and G. Smit. Low-cost guaranteed-throughput communication ring for real-time streaming MPSoCs. In *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, 2013.

Propositions

accompanying the thesis

An Abstraction-Refinement-Theory for the Analysis and Design of Concurrent Real-Time Systems

by Philip-Sebastian Kurtin

to be defended on Friday, November 9th, 2018

1. A metamodel must be sufficiently generic to incorporate existing models, sufficiently expressive to capture their relations and sufficiently structured to support efficient analyses. (Chapter 4)
2. While exclusion and inclusion can only simplify the existing, bounding allows for the creation of something new. (Chapter 5)
3. Tighter analyses enable better implementations. Better implementations enable tighter analyses. (Chapters 9-12)
4. Exhaustively simulating the behavior of a single molecule is equally hard as simulating the behavior of all molecules in the universe. But that doesn't stop us from exhaustively simulating the abstraction of a molecule in no time. (Chapter 13)
5. In the past, power was with those who possessed knowledge. Nowadays, it's with those who can process it. Abstraction enables the latter.
6. There is no known reality, but only the realities defined by the abstractions of human perception. Science gives one such abstraction, religion another.
7. Research is always biased by its goal. But without a goal, there wouldn't be research.
8. There is no efficient creativity.
9. One strives for purpose, the other for pleasure, but neither path can lead to more or less than the striver's happiness.
10. Curiosity kills the man. But a lack of it kills mankind.

ISBN 978-90-365-4646-1

9 789036 546461