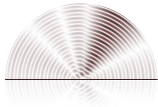
Kenneth C. Rovers

Functional model-based design of embedded systems with UniTi

Kenneth C. Rovers

Members of the dissertation committee:

prof. dr. ir.	G.J.M. Smit	University of Twente (promotor)
dr. ir.	J. Kuper	University of Twente (assistant promotor)
dr. ir.	A.B.J. Kokkeler	University of Twente (assistant promotor)
prof. dr. ir.	M.J.G. Bekooij	University of Twente / NXP Semiconductors N.V.
prof. dr. ir.	F.E. van Vliet	University of Twente / TNO
prof.	W.M. Taha, Eng., PhD	Halmstad University, Sweden
dr. ir.	H. Schurer	Thales Nederland B.V.
prof. dr. ir.	A.J. Mouthaan	University of Twente (chairman and secretary)



This research has been conducted within the Netherlands Streaming (NEST) project (10346), supported by the Dutch Technology Foundation STW, applied science division of NWO and the Technology Program of the Ministry of Economic Affairs.

THALES

This research has been supported by Thales Nederland B.V.

CTIT

CTIT Ph.D. Thesis Series No. 11-213
Centre for Telematics and Information Technology
University of Twente, P.O.Box 217, NL–7500 AE Enschede

Copyright © 2011 by Kenneth C. Rovers, Enschede, the Netherlands.

All rights reserved. No part of this book may be reproduced or transmitted, in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without prior written permission of the author.

Typeset with LaTeX.
This thesis was printed by Gildeprint, the Netherlands.

ISBN 978-90-365-3294-5
ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. 11-213)
DOI 10.3990/1.9789036532945

Functional model-based design of embedded systems with UniTi

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 2 december 2011 om 14.45 uur

door

Kenneth Christian Rovers

geboren op 8 april 1978
te Rotterdam

Dit proefschrift is goedgekeurd door:

prof. dr. ir.	G.J.M. Smit	(promotor)
dr. ir.	J. Kuper	(assistent promotor)
dr. ir.	A.B.J. Kokkeler	(assistent promotor)

Voor jou en mij

Abstract

Advancing the field of embedded systems requires a rigorous approach to their design. This is because embedded systems are complex, diverse and challenging. Yet the design of embedded systems is typically performed in an ad-hoc manner. There is strong evidence that this is reaching its limits and that the design process calls for a unified, integrated, and formal approach. However, we are let down by tool support. Although many tools exist, none support the following four essential features: (i) the modelling of multiple domains, (ii) accurate inclusion of time, (iii) mathematical definitions, and (iv) model transformations. In addition, such a tool must underlie a sound design flow that adequately supports the complexity of designing embedded systems.

In this thesis we propose a design flow and a modelling and simulation framework called UniTi that manages complexity in a top-down fashion; a problem is split up into sub-problems that are solved individually and then combined. This design flow and framework is based on model-based design, i.e. a single reference model is iteratively and incrementally developed and refined during the design process. Our approach is a functional approach, not only because it is practical and useful, but also because it has a mathematical basis supported by a functional language, i.e. computations are considered as evaluations of mathematical functions.

In this work we specialise the design for the application domain of beamforming applications. Beamforming applications use signal processing to achieve directionality for an array of antennas. After a discussion of the basic theory of beamforming, we propose a generic platform for beamforming applications. This platform is hierarchical in the sense that beamforming is performed in multiple stages, and hybrid in the sense that both analogue as well as digital stages are used. A directional antenna can be exploited to search for and track signals-of-interest. Two adaptive algorithms for tracking are developed in the context of this platform.

Next, we investigate suitable architectures for the platform. Supporting multiple applications in an application domain requires scalability to accommodate differently sized applications and flexibility to accommodate different functionality of applications. However, the architecture must also be efficient, because most embedded systems are resource constrained. A tiled reconfigurable architecture is used, as the tiles provide scalability and reconfigurability provides flexibility. Reconfigurability refers to the ability to configure a processor to perform the same computations on a time scale much larger than the processing of individual data elements, improving efficiency by reducing control signal changes and allowing only a limited set of configurations. As not much is known yet about larger high-performance applications (such as beamforming) on tiled reconfigurable embedded systems,

we will explore beamforming on tiled architectures. Different beamforming methods are implemented for a single reconfigurable processor, which can be selected by reconfiguration. Furthermore, reconfigurability supports switching between a computationally intensive searching algorithm and beamforming combined with a much less intensive tracking algorithm. The beamforming applications considered (radar, radio astronomy, satellite reception and wireless communications) are too large to fit on a single tile. Therefore such applications must be partitioned over multiple tiles. This involves making computation and communication explicit, for which we use a dataflow model. Furthermore, this requires the communication infrastructure to be flexible and reconfigurable as well. Finally, we will explore the mapping of a larger beamforming application, a radio astronomy application, on a conceptual architecture consisting of 64 tiles per integrated circuit (IC).

The design of a beamforming platform based on a tiled architecture is supported by a single model that is refined during development. Therefore we need to represent the environment, the architecture and the applications in this model. The environment models the signals that are received at the antennas and requires *exact* modelling of time delays in the continuous-time (CT) domain. In addition, analogue hardware is represented in the CT domain, while digital hardware is represented in the discrete-time (DT) domain and the dataflow (DF) domain is used to represent the software. We propose to use model transformations for the design steps in the design flow, each time breaking down the design into sub-components. We start with an executable specification using mathematical definitions. Then, we perform analogue/digital co-design and hardware/software co-design to divide the functionality over the domains. The next step is division within a domain, consisting of partitioning the application (software). This last step requires parallelisation of the application, after which the partitioned application is mapped and implemented onto the tiles, i.e. assigned to hardware.

There are few tools that support the CT, DT *and* DF domains in a single framework. There are even fewer tools that support model transformations for the presented design steps. Finally, there are no tools (to the best of our knowledge) that are able to *exactly* model time transformations, such as time delays.

UniTi does provide these features. It supports multiple domains: we formally define the CT, DT and DF domains. UniTi also supports exact time delays in the CT domain. This is made possible because signals in the CT domain are represented as functions of time, and model components, represented as signal transformations, are composed using function composition instead of value-passing. To integrate the domains their interaction is defined. UniTi supports unified sequential, parallel and feedback composition of model components. This is achieved by re-defining the dataflow model to match with CT and DT components and signals. As a consequence, mixed-domain models are executable for simulation. State is introduced to improve simulation performance. Visualisations are provided as side-effects during simulation. Finally, UniTi provides support for model transformations; by using (i) automated interaction between domains, (ii) aggregate definitions which specify algorithms at a higher abstraction level, and (iii) by higher-order transformations that exploit mathematical properties of the formally defined models.

We verify UniTi with beamforming on a tiled architecture as a case study. The steps in the design flow are followed from specification to implementation. An executable specification is defined of a simple beamformer, an adaptive beamformer with a tracking algorithm and a hierarchical beamformer. Co-design is performed leading to multi-domain models representing the environment and the system (architecture and applications). These UniTi models are compared to equivalent models in Simulink, and are found to be more efficient (in execution time) while providing exact time delays. Next, the adaptive beamformer is partitioned, mapped and implemented on a small prototype tiled reconfigurable architecture. Finally, the UniTi design flow and framework is evaluated.

The result of this work is a functional model-based design approach for designing, modelling, and simulation of embedded systems. UniTi supports unified composition of multi-domain models and accurate inclusion of time. Using a unified formal transformational design approach improves the interaction between domains and enables smaller iterations, early integration and design space exploration; all sustaining the design of more complex systems. As such, embedded system design is taken to a higher level, allowing the promise of model-based design to become reality.

Samenvatting

Ingebedde of geïntegreerde systemen zijn complex, divers en uitdagend. Toch blijkt er bij het ontwerpen van deze systemen vaak sprake te zijn van een ad-hoc aanpak. Er zijn sterke aanwijzingen dat deze aanpak de ontwikkeling van de systemen begrenst. Om verdere ontwikkelingen mogelijk te maken is het noodzakelijk om bij het ontwerpproces een universele, geïntegreerde en formele aanpak te hanteren. Gangbare programma's ter ondersteuning van het ontwerpproces blijken hiervoor ontoereikend. Hoewel er vele programma's bestaan, biedt geen daarvan ondersteuning voor vier essentiële eigenschappen: (i) het modelleren van meerdere domeinen, (ii) accurate ondersteuning van tijd, (iii) wiskundige definities en (iv) model transformaties. Tevens zou een dergelijk programma de onderbouwing dienen te vormen van een aanpak die de complexiteit van het ontwerpen van geïntegreerde systemen adequaat ondersteunt.

In dit proefschrift presenteren we een ontwerpproces samen met een raamwerk voor modellering en simulatie, genaamd UniTi. Hierin wordt complexiteit beheerst door het probleem op te splitsen in deel-problemen, deze individueel op te lossen, en vervolgens weer te combineren. Het ontwerp proces en raamwerk zijn afgeleid van model-gebaseerd-ontwerp. Hierbij wordt één enkel referentie model iteratief en stapsgewijs ontwikkeld en verfijnd. Onze aanpak is een functionele aanpak, niet alleen omdat het praktisch en bruikbaar is, maar ook omdat het een wiskundige basis heeft welke ondersteund wordt door een functionele taal, dat wil zeggen een berekening wordt beschouwd als de evaluatie van een wiskundige functie.

We specialiseren het ontwerp, in dit werk, voor het applicatie domein van bundelvorming applicaties. Bundelvorming applicaties gebruiken signaalverwerking operaties om richtingsgevoeligheid te bewerkstelligen voor een rooster van antennes. Nadat we de basistheorie van bundelvormen hebben behandeld, wordt een generiek platform gepresenteerd dat geschikt is voor meerdere bundelvorming applicaties. Dit platform is hiërarchisch omdat bundelvorming in meerdere stappen wordt uitgevoerd, en hybride omdat zowel analoge als digitale stappen worden gebruikt. Een richtingsgevoelige antenne kan worden gebruikt om signalen te zoeken of te volgen. In de context van dit platform zijn twee adaptieve algoritmes voor het volgen van signalen ontwikkeld.

Vervolgens evalueren we geschikte architecturen voor dit platform. Om meerdere applicaties binnen een applicatie domein te ondersteunen is schaalbaarheid nodig voor verschillende applicatie groottes, en flexibiliteit voor het ondersteunen van verschillende (bundelvormings-) applicaties. Tevens moet de architectuur efficiënt zijn, aangezien geïntegreerde systemen beperkte middelen tot hun beschikking hebben. We gebruiken een getegelde herconfigureerbare architectuur,

waarbij de tegels voor schaalbaarheid zorgen en de herconfigureerbaarheid voor flexibiliteit zorgt. Herconfiguratie staat voor het configureren van een processor zodat deze dezelfde berekeningen uitvoert op een tijdschaal die veel groter is dan de tijdschaal van de individuele data elementen. Daarbij wordt de efficiëntie verbeterd door het verminderen van controle signalen. We verkennen verschillende implementaties van bundelvormen op een getegelde herconfigureerbare architectuur, aangezien er nog weinig bekend is over het uitvoeren van grotere reken-intensieve applicaties (zoals bundelvorming) op getegelde herconfigureerbare ingebedde systemen. Verschillende bundelvormings methoden zijn geïmplementeerd op een enkele herconfigureerbare processor, waarbij de methode wordt geselecteerd door herconfiguratie. Herconfiguratie ondersteunt ook het wisselen tussen een reken-intensief zoek-algoritme en de combinatie van bundelvormen met een veel minder reken-intensief volg-algoritme. De beoogde bundelvorming applicaties (radar, radio astronomie, satelliet ontvangst en draadloze communicatie) zijn te groot om uitgevoerd te worden op één tegel. De applicatie moet daarom verdeeld worden over meerdere tegels. Hiervoor worden de berekeningen en de communicatie expliciet gemaakt door gebruik te maken van een dataflow model. Verder moet de communicatie infrastructuur evenwel flexibel en herconfigureerbaar zijn. Tenslotte onderzoeken we de verdeling van een grotere bundelvorming applicatie, een radio astronomie applicatie, over een concept architectuur bestaande uit 64 tegels per geïntegreerd schakeling (IC).

Het ontwerp van een platform voor bundelvormen gebaseerd op een getegelde architectuur wordt ondersteund door één model dat tijdens het ontwerp proces wordt verfijnd. In dit model worden daarom de omgeving, de architectuur en de applicatie gerepresenteerd. De omgeving modelleert de signalen die door de antennes worden ontvangen, en het is daarbij noodzakelijk dat tijdvertragingen in het continue tijd (CT) domein *exact* zijn. Tevens is de analoge hardware gerepresenteerd in het CT domein, terwijl de digitale hardware in het discrete tijd (DT) domein wordt gerepresenteerd en het dataflow (DF) domein is gebruikt om de software te representeren. We gebruiken model transformaties voor de ontwerpstappen in het ontwerp proces, waarbij elke keer het ontwerp opgedeeld wordt in sub-componenten. Het startpunt is een uitvoerbare specificatie door gebruik te maken van wiskundige definities. Dan verdelen we de functionaliteit over de domeinen door gebruik te maken van analoog/digitaal co-ontwerp en hardware/software co-ontwerp. De volgende stap is verdeling binnen een domein, bestaande uit opdeling van de applicatie (software). Deze laatste stap vereist parallellisatie van de applicatie, waarna de opgedeelde applicatie wordt toegewezen aan en geïmplementeerd op de tegels van de hardware.

Er zijn weinig programma's die de CT, DT *en* DF domeinen in één raamwerk ondersteunen. Er zijn nog minder programma's die modeltransformaties ondersteunen voor de gepresenteerde ontwerp stappen. Tenslotte zijn er geen programma's (zover ons bekend) die in staat zijn tijdtransformaties, zoals een tijdvertraging, exact te modelleren.

UniTi ondersteunt dit allemaal wel. Het ondersteunt meerdere domeinen: we definiëren de CT, DT en DF domeinen formeel. UniTi ondersteunt ook exacte

tijdvertragingen in het CT domein. Dit is mogelijk omdat signalen in het CT domein als functies van tijd worden gerepresenteerd en omdat componenten van het model, gerepresenteerd als signaal transformaties, worden samengesteld met functie-compositie in plaats van het doorgeven van waardes. Om de domeinen te integreren wordt hun interactie gedefinieerd. Tevens ondersteunt UniTi universele compositie van modelcomponenten met behulp van sequentiële koppeling, parallelle koppeling and terugkoppeling. Om universele compositie mogenlijk te maken is het dataflow model hergedefinieerd om aan te sluiten bij CT en DT componenten en signalen. Als gevolg zijn modellen met meerder domeinen uitvoerbaar voor simulatie. Het bijhouden van de toestand tijdens de simulatie word geïntroduceerd om de efficiëntie te verbeteren. Tenslotte ondersteunt UniTi model transformaties door gebruik te maken van (i) automatische interactie tussen domeinen, (ii) aggregaat definities voor het specificeren van algoritmes op een hoger abstractie niveau en (iii) door hogere-orde transformaties die handig gebruik maken van de wiskundige kenmerken van de formeel gedefinieerde modellen.

We verifiëren UniTi met bundelvormen op een getegelde architectuur als een casestudy. Daarbij worden de stappen in het ontwerpproces gevolgd van specificatie tot implementatie. Een uitvoerbare specificatie van een simpele bundervormer, een adaptieve bundelvormer en een hierarchische bundelvormer zijn gedefinieerd. De co-ontwerp stap leidt tot multi-domein modellen welke de omgeving en het systeem (architectuur en applicatie) representeren. Deze UniTi modellen zijn vergeleken met equivalente modellen in Simulink en het blijkt dat de UniTi modellen efficiënter zijn (in executie-tijd), terwijl ze ook exacte tijdvertragingen ondersteunen. Vervolgens is de adaptieve bundelvormer opgedeeld, toegewezen aan en geïmplementeerd op een kleine getegelde herconfigureerbare prototype architectuur. Tenslotte zijn het UniTi ontwerp proces en raamwerk geëvalueerd.

Het resultaat van dit werk is een functionele model-gebaseerde ontwerpmethode voor het ontwerpen, modelleren en simuleren van ingebedde systemen. UniTi ondersteunt universele compositie van multi-domein modellen en accurate ondersteuning van tijd. Door gebruik te maken van een universele formele ontwerp methode met modeltransformaties wordt de interactie tussen de domeinen verbeterd en worden kleinere iteraties, snellere integratie en exploratie van de ontwerp-ruimte mogelijk; allemaal dragen ze bij aan het ontwerpen van meer complexe systemen. Het ontwerpen van ingebedde systemen wordt als zodanig naar een hoger niveau getild, waarmee de belofte van model-gebaseerd ontwerp realiteit kan worden.

Dankwoord

Voor je ligt het resultaat van vijf jaar hard werken. Dat promoveren een uitdaging zou worden stond van te voren al vast. Dat de grootste uitdaging niet in het werk zou liggen had ik echter niet kunnen weten. Maar het is gelukt; het boekje ligt er. Er zijn echter vele mensen die hier direct of indirect aan hebben bijgedragen, zonder wie het niet gelukt was, en die wil ik hierbij dan ook graag bedanken.

Ten eerste wil ik natuurlijk mijn promotor en co-promotoren, Gerard, Jan en André, bedanken. Jan kwam pas bij de groep toen ik al anderhalf jaar bezig was, maar direct was er de herkenning in de manier van programmeren die Jan meebracht. Door mijn gecombineerde achtergrond in elektrotechniek en informatica paste functioneel programmeren veel beter bij mijn belevingswereld. In de zomer van 2008 gingen Jan en ik naar een “summer-school” over multi-processor systemen in Valkenburg. Ik kan wel zeggen dat de kern van dit proefschrift tot stand is gekomen tijdens deze week. Deze benadering was echter zo vanzelfsprekend voor mij, dat het nog lang heeft geduurd voordat ik herkende dat mijn aanpak wezenlijk vernieuwend was. Dit was nooit gelukt zonder de hulp en het vertrouwen van Jan. Samen hebben we nog wel keihard moeten werken om alles rond te krijgen het afgelopen jaar, en ook daar ben ik Jan heel dankbaar voor.

Ook bij André staat de deur altijd open. André weet feilloos de kern van je werk en wat je wil zeggen bloot te leggen. En bij eventuele problemen weet je zeker dat een paar uur discussie met André, samen met een vol white-board, een oplossing geeft. Als je met werk of vragen van welke aard dan ook bij André komt; je kan erop rekenen dat het secuur bekeken wordt en dat je met goed advies, inclusief een lijst met spelfouten, weer vertrekt. Hier heb ik de afgelopen jaren dan ook veelvuldig gebruik van gemaakt.

Gerard is er voor het grote geheel, en als zodanig perfect op zijn plaats als prof van de groep. Gerard heeft me de mogelijkheid en ruimte gegeven om de inhoud van mijn promotie zelf te bepalen, maar toch weet Gerard altijd de koppeling en relevantie van het werk met de rest van de groep te behouden. Ook kan je altijd bij Gerard terecht als er een stuk tekst, zoals een paper of een hoofdstuk, gereviewed moet worden. Binnen no-time heb je dan commentaar terug dat exact aangeeft waar de sterke en zwakke punten zitten. Hier heb ik vele malen veel profijt van gehad.

Ook de rest van de commissie wil ik graag bedanken; Hans voor de vruchtbare discussies tijdens de vele bezoeken aan Thales, en tijdens mijn stage daar, Frank voor het samen brainstormen over de inhoud en de structuur van het proefschrift, wat de lijn van het verhaal erg heeft geholpen, Marco voor de altijd interessante discussies, welke vaak nuttig bleken om mijn claims scherp te krijgen, and finally Walid Taha

who, as one of the few, is also working in both the areas of functional programming and embedded system design, and who provided encouraging acknowledgement of my work in a larger setting than the Twente region.

Tijdens mijn promotie ben ik betrokken geweest bij het NEST project en bij het CMOS Beamforming project. Iedereen bedankt voor de interessante presentaties, discussies en feedback die voortkwamen uit deze projecten de afgelopen jaren.

Als dubbelstudent lag mijn interesse op het grensvlak van elektrotechniek en informatica, tussen hardware en software, tussen analoog en digitaal, toegepast op de architectuur. Het was dan ook logisch dat ik bij de CAES groep terecht kwam, waar dit ook leefde. Ik was één van de eerste AiO's die Gerard als nieuwe prof aannam en heb CAES onder zijn hand (verder) zien uitgroeien tot een geweldige, dynamische, betrokken en bovenal gezellige groep: bedankt allemaal. Ook onmisbaar zijn natuurlijk de secretaresses, wat dat betreft zitten we bij CAES goed met Marlous, Thelma en Nicole.

Het gros van de tijd heb ik Marcel als kamergenoot en als semi-gedeelde project-genoot gehad. Buiten dat het erg gezellig was, heb ik onze samenwerking altijd als zeer prettig ervaren; vaak lagen wij op één lijn wat betreft onze ideeën, maar toch konden we elkaar ook altijd aanvullen. Meestal samen hebben we een heel blik aan afstudeerders begeleid, wiens werk zeker ook heeft bijgedragen aan dit proefschrift: Rik, Jasper, Mark, Koen, Gerard, Fassil en Rinse bedankt.

Met de voorbereiding en tijdens de verdediging ben ik heel blij dat Koen en Bastiaan mij bijstaan als paranimfen. Bastiaan ken ik al van uit de box en tijdens de middelbare school werden we echt goede vrienden. Samen hebben we ook een flat en onze studietijd gedeeld. Een tijd die ik enorm waardeer en waar ik met veel plezier aan terug denk. Ook daarbuiten ben je er altijd als vriend. Het is daarom ook passend dat je er bij bent, bij deze toch soort van afsluiting van het "eeuwige‘" studeren. Koen ken ik een stuk korter, maar zeker sinds je in Zutphen woont stel ik je vriendschap op prijs, met vele goeie discussies of gezellige gesprekken, in de trein of bij een biertje. Daarbuiten heb je ook een belangrijke inhoudelijke bijdrage geleverd aan dit proefschrift. Fijn dat je me daarom ook bijstaat als paranimf.

Het lijkt soms zo dat er geen wereld is buiten het promoveren, maar toch had ik het zonder vrienden en familie niet gered. Sommige daarvan ken ik al heel lang. Toch heeft soms de frequentie van het contact moeten lijden, maar dat maakt de waardering niet minder. Ik kan de verleiding toch niet weerstaan om een aantal in het bijzonder te noemen; Bas & San, en tegenwoordig ook kleine Hannah, bedankt voor jullie vriendschap. Ermano, ook wij hebben al heel wat meegemaakt, bedankt voor alles. Anneke, ik zie je wat minder, maar elke keer weer sinds die eerste vlucht naar de nieuwe wereld, klikt het. Van mijn UT tijd heb ik ook een aantal goeie vrienden overgehouden. Arno, met jou is het altijd een avontuur, op de scooter in Koh Phangan of in de kajak bij Milford Sound, of gewoon in Nederland met een goed gesprek. Jeroen, jouw drive en enthousiasme heb ik altijd bewonderd, maar bovenal waardeer ik je gezelligheid. Tenslotte wil ik dan nog de familie bedanken: opa en oma natuurlijk, alle ooms en tantes, en alle andere familie.

De basis van wie je bent wordt toch thuis gelegd, en wat dat betreft had ik het niet beter kunnen treffen. Altijd kon en kan ik terugvallen voor steun, warmte en

vertrouwen. De basis van mijn nieuwsgierigheid, motivatie, het doorzettingsvermogen en de rust komt van pap. Het doet dan ook veel pijn dat je er niet in persoon bij kan zijn. In mijn hart draag ik je bij me, ik weet dat je trots zou zijn. Mam, van jou komt het enthousiasme, de kracht, en alle steun en zorg die ik nodig heb. Dit proefschrift is ook voor jou. Natuurlijk is thuis niet compleet zonder mijn broertje. Dan, bedankt voor je levendigheid en plezier. Samen met Marieke en lieve kleine Mirthe is het nog steeds een geweldig thuis.

Al aardig wat jaren nu heb ik ook een nieuw thuis, met Linda, mijn allerliefste schat. Het was een pittige tijd, maar met jou heb ik het samen gedaan, altijd ben je er voor me, weet je wat er moet gebeuren en geef je me net het beetje extra dat ik nodig heb.

Kenneth Rovers
Warnsveld, November 2011

Table of Contents

1 Introduction 1
1.1 *Trends in embedded systems* 5
1.2 *Beamforming as an example* 7
1.3 *Problem statement* 8
1.4 *Contributions* 9
1.5 *Outline* 10

2 Application domain: beamforming 13
2.1 *Characteristics* 14
2.2 *Phased array beamforming theory* 17
2.3 *Generic beamforming platform* 32
2.4 *Beamcontrol* 45
2.5 *Conclusion* 56

3 Tiled reconfigurable architectures for beamforming 59
3.1 *Requirements from the application domain* 60
3.2 *Architecture* 63
3.3 *Experiments with tiled reconfigurable architectures* 65
3.4 *Conclusion* 73

4 Model-based design of multi-domain systems 75
4.1 *Motivation* 76
4.2 *Time, signals, components and systems* 83
4.3 *The problem with time* 85
4.4 *Survey of existing tools* 89
4.5 *Unified modelling based on time* 94
4.6 *Design flow* 96
4.7 *Conclusion* 100

5 UniTi 103
5.1 *Formalisation of the domains* 104
5.2 *Composition* 117
5.3 *Integration of the domains* 126
5.4 *Simulation* 131
5.5 *Model transformations* 140
5.6 *Conclusion* 145

6 Case study **149**

6.1 Specification 151
6.2 Co-design 157
6.3 Partitioning 170
6.4 Mapping 174
6.5 Implementation 176
6.6 Results 178
6.7 Conclusion 184

7 Conclusions **187**

7.1 Research questions 190
7.2 Discussion 191
7.3 Outlook 191

A Dataflow **193**

A.1 Terminology 193
A.2 Dataflow model 194
A.3 Dataflow analysis 194
A.4 Dataflow execution 195
A.5 Properties 195

B The Montium **197**

B.1 Processor landscape 197
B.2 The Montium processor 198
B.3 Kernels implemented on the Montium 201

Acronyms **205**

Bibliography **207**

List of Publications **215**

Refereed 215
Non-refereed 216

CHAPTER 1

Introduction

Embedded systems are everywhere; in your car, your television, your mobile phone, printer, router, pacemaker, dish washer etc. As such the embedded systems market is huge and fast growing [41], but as they are embedded into larger systems their presence remains relatively unnoticed to the general public. Yet embedded systems are clearly relevant because of their ubiquity, and they are challenging because of their variety and complexity.

Although existent in a wide variety, there are some common characteristics. All embedded systems are computer systems interacting with their environment, i.e. they contain some form of information processing and interaction via sensors and actuators. The system must continuously monitor and process the signals coming from the environment and act accordingly, i.e. most embedded systems are control systems and contain a lot of signal processing. For example, the embedded system in a refrigerator measures the temperature and based on that controls the cooling system. In addition, an embedded system comprises multiple domains: there are elements with continuous dynamic behaviour such as sensors or analogue filters as well as elements with discrete dynamic behaviour such as digital processors or specialised digital hardware for e.g. encryption. A system with both continuous and discrete dynamic behaviour is called a *hybrid system*. Typically embedded systems are resource constrained, e.g. they must perform their job with limited processing power, limited memory, limited area, and with a low energy consumption. At the same time, embedded systems must be highly reliable and stay within timing constraints. Therefore, it is important that an embedded system is not only correct, but also on time. This means (physical) time is an important aspect for embedded systems.

Overall, most embedded systems are complex and constrained by the interaction with the environment and a limited amount of resources. To deal with this, an embedded system is often optimised for a specific *application domain*, i.e. a range of applications that have similar characteristics. The complexity of the system is

reduced by limiting the required functionality to that required by the application domain, and processing elements can be specialised to and optimised for their common characteristics. On the other hand, as a range of applications must be supported, the requirements of which can also change over the lifetime of the embedded system, the system must also have enough flexibility to cope with this. In other words, an embedded system consists of (specialised) hardware and software. A second strategy to deal with the complexity of embedded systems is to divide the system into sub-components, each with well-defined responsibilities and interfaces. Such components can then be designed independently, re-used for multiple systems and optimised for their task. The resource requirements, both in integrated circuit (IC) area and energy consumption, are further reduced by integrating these sub-components on a system-on-chip (SoC). When such components are connected by a network-on-chip (NoC), we will call them *tiles* and the architecture of the SoC is called a tiled architecture. The architecture is also heterogeneous; as the tiles are optimised for their task to deal with a limited amount of resources they are not all the same but differ in functionality, size, efficiency, etc.

In this thesis we will limit the class of applications to *streaming applications*. Streaming applications operate on streams of data such as audio or video, i.e. new data is not available at once but is made available over time. Typically, streaming applications consist of signal processing operations such as filtering or compression. For embedded systems, the data-rate of the streaming data is often high requiring a relatively large amount of communication per computation. As such, the performance must be sufficient to keep up with the (high) data-rates. Therefore, the architecture of the system is mainly concerned with the flow of data, in contrast to the flow of control as is more common in general purpose architectures [41]. Relevant characteristics of streaming applications are latency (how long it takes for data to go through the system), throughput (how much data is processed per time unit), and real-time constraints that determine the maximum latency and minimum throughput for a defined data rate such that correct operation of the system can be guaranteed.

Designing, modelling and verifying embedded systems is a big challenge; the systems are complex, they comprise multiple domains, they are resource constrained and they need to provide guarantees. As a consequence, the designer needs knowledge about hardware, software, analogue design, digital design, computer architecture, control theory, signal processing and their interaction. Especially their interaction is important: each respective field is well developed but synergy between the fields is lacking and not well understood in our opinion[1]. However, this is essential for current complex multidisciplinary embedded systems. We believe a unified approach will strengthen the collaboration between the fields, a view shared by [13, 48, 59, 68]. However, no satisfactory tool or framework supporting this exists. In this thesis we will propose a design flow and supporting framework, called UniTi, which does offer a unified approach to designing, modelling and simulating embedded systems.

[1] Which is understandable considering the amount of knowledge required.

The hardware and the software of embedded system can not be considered in isolation as they influence each other; they need to be considered simultaneously, i.e. embedded system design requires a holistic approach. There are efforts to support both hardware and software in one computation model such as synchronous languages (e.g. Esterel [10], Lustre [37]) or dataflow (e.g. [57, 65]) or recent efforts such as SystemC [35]. We will use the dataflow (DF) domain for representing an application on a tiled architecture.

The hardware/software co-design approach, however, focuses on the digital side of a design and does not include the analogue side. In this thesis we will take a further step in integrating design methodologies by including analogue components, i.e. analogue/digital co-design or mixed signal design. The integration of analogue and digital components requires support for the continuous-time (CT) domain and the discrete-time (DT) domain. In addition, if the interaction of an embedded system with the environment has a significant effect on its operation, the environment must be included in the model. The need for this is often the case, illustrated by the recent interest in "cyber-physical" systems, which emphasises this interaction with the environment [53, 59]. The environment is typically modelled in the CT domain. So for a unified approach, support is required for the CT, DT and DF domain, expressing the environment and analogue hardware, the digital hardware and the software respectively. A single unified domain will not suffice and is not desirable as we will find that these domains have significantly different interpretations of model components and interaction. For integration of the domains it is therefore important to have a precise definition of their interaction (also see [24]). A formal representation of these domains and their interaction is presented for UniTi in the present work, enabling such integration.

In each of the relevant fields for embedded system design, models aid the design process and are used for functional verification. Such models provide an abstraction of relevant properties of a design. Of course, what is considered an adequate abstraction changes during the design process; at first a basic model is sufficient but as the design develops more detailed properties and second order effects of the system under design are needed for verification. In other words, the design is continuously refined during the design process. Thus, modelling and simulation of the design in its various stages of development is of great importance. UniTi is a design flow and framework based on *model-based design* that supports design refinements. Model-based design makes the model the centre of the design process. Using model-based design, a single reference model is iteratively and incrementally developed and refined, aiming at shortening the design cycles and making integration part of the design process early on. However, in order to make this a viable approach, we need to ensure such a refinement is correctness preserving. Hence, this is supported in UniTi with model transformations.

Model transformations are used to provide structured and well-defined refinements. We use formally defined models so that mathematical properties of such transformation can be proven, ensuring that they *are* well-defined. Others too plead for a mathematical basis for system modelling and analysis [42, 48, 59]. Ideally, automated model transformations could be employed to evaluate many different

designs, so-called *design space exploration*. In practice, model transformations are guided by and even performed manually by the designer. By defining models formally and at a higher abstraction level, we will find that the designer effort for applying model transformations is reduced.

The time at which events from the environment occur is out of the control of the embedded system. Yet the response time or reaction time of the embedded system is typically constrained. In other words, accurate inclusion of time in modelling is vital, when interfacing with the environment, to verify the behaviour of an embedded system over time. In computing, physical time is mostly abstracted away and the time a computation takes is mainly a matter of performance [54]. In *real-time systems* the reaction time is important for correctness rather than performance; typically such systems are control systems that react to an event from the environment and must do so within time constraints [16]. However, time in real-time systems is the execution time of the processing as response from events from the environment; it does not include physical time such as the delay of signals over the network or the CT response of an analogue filter at the input of the system. Mixed domain simulation tools such as Simulink do support modelling physical time, but do not support the modelling of execution time with real-time analysis. In addition, current mixed domain tools model the CT domain by discretising a global simulation time into small time steps and representing signals as a sequence of values at these time steps. As a result, such time steps must be small enough for an accurate representation of the signals. Furthermore, when the time reference of a value is changed, such as for a time delay, and a value at a time between the time steps is needed, interpolation is used between available values. The interpolation approximates the actual value, thereby introducing inaccuracies. Such inaccuracies are not part of the modelled system but are modelling artefacts introduced by the modelling tool. These modelling artefacts are not easily distinguishable from modelling of signal distortions that *are* present in the physical system such as noise or non-linearities. In UniTi we support *exact* modelling of the CT domain as well as supporting execution time in the DF domain for real-time analysis.

In summary, the design of embedded systems consists of managing complexity by specialising to dedicated functionality, by applying a divide-and-conquer approach and by employing model-based design. Using a model-based design flow for designing embedded systems requires a modelling and simulation tool or framework. Unfortunately, support for multiple-domains, time, mathematical definitions and model transformations is not well represented in current design and modelling tools. This thesis on functional model-based design of embedded systems proposes exactly such an approach: a functional approach, because it is practical and useful, but also because it has a mathematical basis supported by a functional language. This approach is called UniTi to emphasise the unification of the design flow, the unification (and integration) of the CT, DT and DF domains, and the unification and accurate inclusion of time.

1.1 Trends in embedded systems

We will identify current trends for the design of embedded systems, so as to guide our work in the coming chapters and evaluate its applicability.

Complexity The complexity of designing embedded systems is increasing as applications are becoming more demanding. Besides common design criteria of embedded systems such as price, energy efficiency, real-time requirements and application specific performance [41], embedded systems must guarantee service with correct functionality while interaction with the environment is increasing and becoming more important, making the inputs and outputs of the system less predictable [59]. This makes complexity the major challenge for embedded system design.

Model-based design To deal with increasing complexity the use of models and model-based design is becoming essential [42, 68]. One step in raising the abstraction level is for example SystemC [35], for which a system is a hardware architecture plus software. This is a large step forward compared to hardware description languages such as VHDL which do not support hardware/software co-design. Often model transformations in model-based design are about code generation [42, 71]. To raise the abstraction level further, multiple domains in a single model supported by "higher level" model transformations are necessary. In this thesis model transformations are considered for the whole design process from specification to implementation.

Multi-domain integration Cost and size reductions for embedded systems are achieved by integrating digital components on an SoC. A natural next step is integrating analogue hardware and digital hardware on a single chip, so-called mixed-signal ICs. For example in the CMOS Beamforming Techniques project[2] we research the feasibility of mixed analogue and digital beamforming and a suitable integrated architecture on a single (CMOS) chip, which could enable beamforming for consumer applications because of the higher integration and lower cost. As technology and integration continues, mixed-signal ICs are expected to become more common for embedded systems. As explained above, a second trend is the increasing emphasis on including the environment in the modelling and design process [53, 59].

Modelling time We have already motivated that accurate inclusion of (physical) time in modelling is vital. As embedded systems are expected to increasingly interact with the environment, support for modelling time is increasingly important [54].

Adaptivity Applications are becoming more adaptive and dynamic. For example, in the latest digital video broadcast for satellite (DVB-S) standard for satellite broadcasting, adaptive coding and modulation is used on a frame by frame basis

---

[2] STW project CMOS Beamforming Techniques (07620) [49]

depending on the signal conditions [29]. Another example is a user running applications on a mobile phone; at any time a new application can be started, possibly together with other time critical applications. Embedded systems must be flexible enough to support adaptivity such as switching functionality or adding functionality as a result of changing conditions.

Many-cores IC manufacturing is still following Moore's law, meaning the number of transistors on a single chip doubles approximately every two years [5]. However, the extra transistors are used differently. Single core processor performance has stopped increasing because we hit a limit in the power usage and thereby the operating frequency, the relative memory latency has become larger, and it has become difficult to find more parallelism in sequential programs [5]. To still improve performance and make use of the extra transistors, more processors are combined on a chip. The number of cores is expected to increase further leading to many-core architectures (hundreds of cores). For example, Intel already has 12-core processors and a 80-core research chip [107].

This work has been performed in the NEST project[3]. In this project, we are researching high performance streaming applications on tiled many-core architectures. In the NEST project research ranges from a system-level design flow, modelling and analysis to the implementation of tiled architectures and the applications running on the architecture. Tiled architectures are used to design scalable systems; by adding tiles the system can perform more processing and/or achieve a higher performance. In addition tiled architectures provide dependability; if a tile breaks down during the lifetime of the system it can be replaced by one of the additional redundant backup tiles, or the performance of the system is reduced enabling graceful degradation.

Larger applications Until recently, tiled architectures have been mainly used for multimedia applications [11, 39, 115]. Those applications themselves are becoming more complex requiring more processing power. However as tiled architectures are becoming more powerful, also larger applications can be supported. Many high-performance applications also operate on streams of data and could benefit from the energy efficiency, scalability and dependability of an embedded system based on a tiled architecture. For example, in the NEST project we are using medical imaging and radar processing as case studies. The radar processing case study is performed in cooperation with Thales Nederland B.V., which specialise in radar equipment, and will be used as a case study in this thesis.

Flexibility and efficiency There is a trade-off between flexibility and efficiency, as a more flexible system requires more hardware [43] making it less (energy) efficient. Flexibility is required to support adaptivity, but also to be able to adapt to future requirements that are not yet known during the design of the system. In addition, a more flexible system can be used for a broader class of applications. For example,

[3] STW project NEST: Netherlands Streaming (10346) [67]

a (hardware and software) platform that supports multiple applications from an application domain. This way the development costs can be shared among the applications. On the other hand embedded systems are resource constrained and efficiency is an important factor of the design.

Hardware costs are becoming less important as the number of transistors on a chip increases. Therefore embedded systems are becoming more flexible. Yet, energy efficiency is becoming more important. In our view, reconfigurable systems nicely balance flexibility and (energy) efficiency and are a good choice for systems that require a modest amount of flexibility, i.e. functionality that changes every few hundred clock cycles or less.

1.2 Beamforming as an example

Throughout this thesis beamforming applications are used as an example of large high-performance streaming applications. Beamforming applications use the signals of multiple antennas to make a directional receiver. This direction can be electronically controlled by processing the streaming data from the antenna signals.

The design of an embedded system as platform for beamforming applications will form a good case study as it covers all of the above trends in embedded systems.

A signal from a direction of arrival (DoA) at an angle with an array of antenna elements will arrive at a slightly different time at each element. The beamforming application will correct for these time differences so that the antenna signals all add up coherently. To model and simulate a beamforming system, accurate representations of the antenna signals need to be generated. In other words, the environment needs to be included in the model where the source signal experiences a time delay to each antenna, and these time delays must be modelled accurately.

Beamforming can be performed in multiple stages, a so-called *hierarchical beamformer*. One or more of these stages can also be performed in the analogue domain, giving a *hybrid beamformer*. For including these analogue beamforming front-ends in the model, as well as for the environment, we need CT domain support. Furthermore, beamforming reduces the data rate by combining signals, so analogue beamforming reduces the amount of digital processing, but digital beamforming is more flexible. So there is a trade-off, requiring an analogue/digital co-design step during the design process.

A beamforming application consists of straightforward processing of the antenna signals, yet requires complex control for determining the steering direction. As the number of antennas can be large, the processing must be performed efficiently. The steering direction is determined by an adaptive algorithm. In the general case, the initial DoA of a signal-of-interest is unknown, requiring a search algorithm. Such an algorithm is computationally complex. When the initial DoA is found, a tracking algorithm can be used which is less computationally complex. We will develop a novel tracking algorithm for modulated signals with a constant modulus and phase, i.e. for phase-shift keying (PSK) modulated signals. This algorithm determines a steering correction per antenna to track the signal-of-interest.

However, such steering corrections are not useable for an hierarchical beamformer. Therefore, we will develop a second tracking algorithm providing a steering angle, which is useable for an hierarchical beamformer at a slightly higher computational complexity but still much less complex than a search algorithm.

Traditionally, for radar and radio astronomy applications, the design of (high-performance) beamforming systems is driven by functional requirements (e.g., resolution, sensitivity, response time) where non-functional requirements (e.g., costs, power consumption) are of secondary concern [109]. For that reason, no low-cost, low-power systems for more than a few antennas are available yet. However, in areas like wireless communications and satellite receivers, phased array antennas show great promise but their large scale introduction has been obstructed by the high costs involved. In this work we present a generic platform for beamforming applications. The goal is to develop a low-cost, low-power beamforming platform by using a scalable architecture that is flexible enough to support multiple applications, such that the same architecture can be reused. In addition, flexibility is used to switch between an initial searching algorithms and a tracking algorithm. Conventional beamforming architectures typically use a large amount of dedicated central processing hardware, making the system neither scalable nor power efficient [90].

We postulate a tiled reconfigurable architecture will provide such a scalable and flexible platform, as they offer high performance (by enabling parallel processing through multiple processors) and flexibility within a certain application domain (reconfiguration enables efficient reuse of hardware by reconfiguring parts of an application). In other words: scalability is achieved by adding tiles, while flexibility, with a limited reduction in efficiency, is achieved by reconfiguration. Hierarchical beamforming is used to achieve scalability in the application. To verify the suitability and to explore the consequences of a tiled reconfigurable architecture for larger applications, the beamforming application is mapped on a small existing tiled architecture and a larger concept architecture. As expected, the beamforming application is too large to run on a single tile and must be partitioned over multiple tiles. Therefore, communication between the parts of the application needs to be explicit. The DF domain provides a fitting representation for partitioned applications, as also used in [39, 115]. The DF domain is therefore also required for modelling and simulation.

1.3 Problem statement

The topic addressed in this thesis is managing complexity in the design of embedded systems. Whenever complexity is encountered (e.g. during design, defining an architecture, implementing the application) the same approach is applied: divide-and-conquer.

For the design process, we choose for a model-based design approach. Such an approach needs a modelling and simulation framework that supports a single model containing multiple domains and model transformations. For the architecture, we choose for a tiled reconfigurable design. Such an architecture supports the

scalability and flexibility needed for a generic platform for the application domain under consideration. As application case study, a larger application is considered requiring modelling of the environment and containing analogue and digital components. This larger application must be partitioned over and implemented on the proposed tiled architecture. This requires explicitly separating computation and communication, and parallelisation of the application.

This leads us to the following research questions and propositions:

- What is a suitable design flow for embedded systems based on a divide-and-conquer approach? A division based on model transformations is needed that requires transformations that are generic, well-defined and correctness preserving.
- What is required from a modelling and simulation framework to support this design flow? The environment, the architecture and the application are to be modelled, requiring accurate and efficient support for multiple domains and their interaction.
- Are tiled reconfigurable architectures suitable for large high-performance applications? Such applications must be defined in such a way that they can be parallelised and partitioned for a tiled architecture.

1.4 Contributions

The main contribution of this thesis is a functional model-based design approach for designing, modelling and simulating embedded systems based on a sound mathematical foundation. We will limit our scope to an application domain requiring high-performance streaming processing, yet propose a hierarchical scalable and flexible platform to support multiple applications in this domain. We will then explore the consequences of mapping such an application onto a SoC with a tiled architecture. As a result from the requirements of the application and architecture, we propose a design flow and framework which uses model-transformations for co-design and partitioning.

Specifically the contributions of this thesis are:

- **A design, modelling and simulation framework** called UniTi is developed supporting multi-domain models, model transformations and exact modelling of time [KCR:10] (chapter 5). UniTi provides a formal, unified, integrated and transformational environment for the design of embedded systems. It is based on function composition of components, where components represent signal transformations. We provide a formalisation of the CT, DT and DF domains in UniTi, and unified composition of mixed-domain models [KCR:9]. To achieve this, dataflow models are re-defined as DF components and signals. DF components still adhere to dataflow execution semantics, but in addition can now be composed with CT and DT components [KCR:10]. As a consequence, time in the CT or DT domain determines the time in the DF domain and gives the execution time of dataflow processes meaning during simulation instead of only during analysis.

- **A design flow that raises the abstraction level** to include the environment, analogue/digital co-design, and an executable specification of the hardware and software [KCR:3, KCR: 10] (chapter 4). The design flow proposes a co-design step as model transformation from the specification to a model including the environment, the architecture and the applications. Furthermore, it proposes a partitioning step as model transformation for parallelising the application, and a mapping and implementation step.
- **An analysis of modelling time transformations in hybrid systems** [KCR:8] (chapter 4). We identify different notions of time in modelling. Current tools coalesce these notions of time into a single global notion of time. As a result, the time of continuous signals is discretised during simulation, causing approximation errors when for example time delays or multi-rate systems are simulated. This is because for such systems the exact time at which the value of a continuous signal is needed may not match with the global discretised simulation time.
- **The design of an hierarchical beamforming platform** suitable for multiple beamforming applications [KCR:12] (chapter 2). A tiled reconfigurable architecture is explored as architecture for the platform as it provides scalability and flexibility [KCR:6, KCR: 11] (chapter 3). A larger application such as beamforming requires the application to be divided over the tiles. Different implementations of beamforming applications are evaluated with respect to their required computation and communication. The beamforming application on a tiled reconfigurable architecture is used as a case study for UniTi [KCR:6, KCR: 9, KCR: 10, KCR: 11] (chapter 6).
- **The application and analysis of two novel beamcontrol algorithms for this platform**, for tracking signals-of-interest with low computational complexity [KCR:4, KCR: 7] (chapter 2). The first algorithm allows low-cost tracking of M-PSK modulated signal when the initial DoA of the signal is known (by first running a search algorithm and reconfiguring), but it is not suitable for hierarchical systems. Therefore an alternative algorithm is developed that is suitable, but has a higher computational cost.

1.5 Outline

In this thesis the first step in managing complexity for the design of embedded systems is specialisation. The application domain of beamforming applications is presented in chapter 2. As result we will find that a generic platform for beamforming applications must be scalable and flexible.

In chapter 3, reconfigurable tiled architectures are explored for beamforming applications on the premise that scalability is provided by tiles and flexibility by reconfigurability.

Following from the discussion of the application domain and architecture we find that functional components of a beamforming application are divided over a representation of the environment, the hardware (mainly architecture) and the

software (mainly application), and that the application is divided over tiles of the architecture using DF models. This leads to a design flow that supports multiple domains, time and model transformations in chapter 4. A survey of existing tools shows that such a tool is not available. Therefore we propose UniTi in chapter 5; a multi-domain transformational design flow and modelling and simulation framework.

In chapter 6 the results of the previous chapters are combined in a case study. An embedded system for beamforming is designed from specification to implementation and the UniTi design flow and framework is evaluated.

Finally, we will present the conclusions, and we will discuss future work, in chapter 7.

Chapter 2

Application domain: beamforming

Abstract – *Many embedded systems perform signal processing on streaming data from the environment. One such application is a beamforming application, which will form a good case study for the design of embedded systems. In this chapter we will present the application domain characteristics and basic theory of beamforming applications. We will then develop a generic beamforming platform. Such a platform must be (energy) efficient, scalable and flexible to be cost-effective. This is achieved with a hierarchical and hybrid design. In addition, we present two new beamcontrol algorithms for tracking signals with a phased array beamforming system at a low computational cost.*

Embedded systems are specialised for a specific application domain in order to reduce their complexity and improve their efficiency by requiring embedded systems to do less, but do it well. A range of applications that have similar characteristics are together called an *application domain*. Throughout this thesis we will use the application domain of phased array beamforming applications as an example.

Phased array beamforming systems use multiple antennas in an array to make a directional receiver. In essence, a phased array is performing a *spatial* filter that selects the signal from the direction of interest. This direction can be electronically controlled, thereby making a phased array system very suitable for situations in which the direction of the signal is continuously changing or where signals from multiple directions need to be selected simultaneously.

A beamforming application is a high-performance streaming signal processing application; as an array of antennas is used, each continuously transmitting or receiving a signal, phased array systems involve a lot of signal processing on streaming data. Yet, phased array beamforming is typically part of a larger system,

Parts of this chapter have been published in [KCR:4], [KCR:7] and [KCR:12].

such as a radar system, that poses resource constraints, e.g. in area, processing capacity and power. In addition, there are timing constraints resulting from the continuous stream of data: typically no data may be lost or the reaction time is bounded. Beamforming systems also interact with the environment by sending and/or receiving signals. All these characteristics together make a phased array system a good case study for the design of embedded systems.

Traditionally, phased arrays have been used for radar systems to detect and track moving targets. Another common use is for radio astronomy, to correct for the movement of the earth but also because very selective filtering can be performed in multiple directions simultaneously. Their requirements normally dictate a dedicated design. Costs have withheld the use of phased arrays for other applications, but one can imagine the usefulness in consumer applications, such as a flexible satellite receiver or for mobile and wireless communication.

In this chapter we will propose a generic platform for beamforming applications. By providing a flexible, scalable and efficient design, the same platform can be reused. This lowers the cost of the platform because the design costs are shared and the production volume is higher, thereby possibly enabling phased array systems for consumer applications. To achieve a scalable and modular design, the beamformer is hierarchical: beamforming is performed in multiple stages. To save further costs, part of the beamforming stages are performed in the analogue domain with dedicated hardware, resulting in a hybrid beamformer.

A beamforming platform requires an (adaptive) algorithm to search or track a signal-of-interest. We will present an overview of beamcontrol algorithms and find that search algorithms are computationally expensive. Yet, we require a beamcontrol algorithm with low computational cost so that limited additional hardware is required. Therefore we present an equalisation algorithm for PSK modulated signals which we apply as an adaptive beam-control algorithm with low computational cost. Furthermore, based on this adaptive beamcontrol algorithm, an algorithm is presented that, unlike existing algorithms, is also useful for hierarchical beamforming systems.

This chapter is organised as follows. First we present an overview of the characteristics of the beamforming application domain in section 2.1. Thereafter, in section 2.2, we will present relevant beamforming theory. Next, we will discuss the system design of a hierarchical hybrid beamforming system, proposed as generic beamforming platform. Section 2.4 gives an overview of adaptive algorithm classes for beamcontrol and presents two novel tracking algorithms, followed by a conclusion.

2.1 Characteristics

In this section a short overview of the areas relevant to phased array beamforming systems is given. This will also be useful for later chapters on architectures for and design of embedded systems.

2.1.1 *Signal processing*

A signal, in the sense of signal processing, is a representation of a physical quantity that varies with time or space, i.e. signals are functions of the independent variable time and/or space. Signals encode and transfer information. On a single channel, information can be encoded in the amplitude, frequency and/or phase of a signal. For example, a speech signal encodes phonetic symbols as well as emphasis etc. as sections of varying frequency and amplitude.

Multiplexing information into one signal is used to send more information over a shared medium or channel. Information can be multiplexed over time, frequency and space or a combination of these.

Signals are generated by sources and consumed by sinks. A system, subsystem or component responds to or transforms signals, i.e. it performs processing on the signal. Signal processing can be performed in both the analogue and the digital domain. Digital signal processing is often preferred because it is more flexible and/or has better accuracy. By continuous signals, continuous-*time* signals are meant [93]. Likewise, discrete signals are defined for discrete-*time* (and may well have continuous values). A *digital* signal is a discrete-valued discrete-time signal.

2.1.2 *Streaming data*

A stream is an infinite sequence of data. Signal processing systems often operate on streaming data, because an input signal (as function of time) that is digitised can be represented as an infinite stream of data; the samples of the signal. Thus, a digital signal can be represented as streaming data.

The advantage of a stream representation for signal processing applications is that it becomes easier to formally analyse and verify the applications [53, 57]. It can be guaranteed that the application is functionally correct and what kind of throughput and latency it achieves. Throughput for streams and signal processing is defined as the (average) rate that elements from the stream are processed. Latency is defined as the time delay of an element when being processed.

There are two common representations of streaming data [57]. The first is as an infinite list; the first element of the list is the current data and the remainder of the list are the future values. Signal processing operations are performed on the list, i.e the inputs and outputs of an operation are lists. Laziness, i.e. values are only calculated or retrieved when used, ensures that the whole stream does not need to be available when used by the operation. A stream as a list can also be represented as a pair of the current value and a function to get future values, i.e. a linked list.

The second representation of a stream is a representation as a channel. A channel is an unbounded first-in first-out (FIFO) buffer. New elements are added to the channel, thereby modifying the channel, and signal processing operations consume elements from the channel. An advantage of the channel model is that signal processing operates on single elements at a time, which fits the conceptual execution of the signal processing operation, as it progresses over time. A disadvantage is that a channel does not match the semantics of a discrete time signal, i.e. a sequence of

samples over time. The channel represents a container where samples are put in or taken out. As such, it is a lower abstraction concerned with memory management.

We propose a different representation: a stream is data that changes over time. It is therefore close to a DT signal: instead of consisting of samples, the signal consists of data. In fact, we consider data (elements) to be equivalent to samples, i.e. data is represented as a (large) number. In other words, a stream is a value that changes over time. A signal processing operation is just an operation on a value at a certain time. As a consequence its output is also a value that changes over time or a stream. An operation on a stream has no notion of time, only of the ordering of data and the next element to process. It can have state, i.e. its output depends on its history of inputs. The state and output of an operation change with a new input value, irrelevant at which time this is. As such, time is defined at a higher level, and is irrelevant for (the correctness of) the signal processing operation. Summarising, the same representation of values that change over time is used for DT signals and streams. We will come back to this in chapter 4.

In synchronous languages such as Lustre [37], Lucid, [18], Esterel [10] or Signal [52], each stream is associated with a (global) "clock". The next element of the stream models a "clock tick". With the proposed representation, the value of streams at a certain time does not necessarily represent the same (global) "current" time or clock, i.e. time is locally defined and relative.

2.1.3 *Hybrid systems*

The dynamic behaviour of a system is the time-varying evolution of the system. If the behaviour of such a dynamic system has both continuously changing elements as well as discrete changes, it is called a *hybrid system*. As mentioned, embedded systems interact with their environment, which is typically a continuous system, while most of the processing is typically in the discrete domain. Therefore, an embedded system is often a hybrid system.

A hybrid system thus includes the CT domain and the DT domain. A domain is a meta-model that formalises what entities or components in the domain represent and how they interact.

A system that responds to dynamic behaviour from the environment is called a *reactive system*. Signal changes usually indicate events which the system must react to. A system that must react within a certain time (before its deadline), while it has no control over when events occur, is called a *real-time system*. For real-time (hybrid) systems it is especially important that throughput and latency constraints are met, as missing a deadline can have severe consequences for the system's operation. Models of the system are used to analyse the system's behaviour and provide guarantees that constraints can be met. As such, a model, based on streams, is often used [57].

When modelling hybrid systems, the dynamic behaviour of the environment must be included in the model to verify the dynamic behaviour of the system's interaction with the environment. The signal generation part models the aspects of the environment that are relevant to generate the input signal of the system. When

actuators are controlled by the system, they might influence the input signal, and thus the signal generation model, thereby forming a closed loop system.

2.1.4 *Adaptive algorithms*

Adaptive algorithms (automatically) adapt or adjust to the characteristics of an input signal [3]. As such they are similar to closed-loop control systems; the algorithm compares the input and the output of an operation with an expected result and changes the operation on the input signals accordingly. A feedback loop is thus created by the adaptive changes, or corrections, to the operation on the input signals.

Typical aspects of control engineering and control theory are also relevant here, such as:

- *stability* ensures the controlled response converges to the intended effect and stays within bounds,
- *responsiveness* is the time it takes to reach the intended effect,
- *overshoot* refers to a controlled signal exceeding its intended value.

Often the feedback control does not need to run at the same speed as the rest of the signal processing because the dynamic behaviour is slower than the rate of information. In that case the signals for the control and feedback can be decimated, i.e. their rate is reduced.

2.2 Phased array beamforming theory

Beamforming, as the name implies, is about forming an electromagnetic or acoustic beam into a certain direction, i.e. it makes a transceiver directional. An (mechanical) example is a light-beam from a spotlight. Exploiting the directivity of a transceiver is an obvious way of improving the performance of a radio frequency (RF) system. This is because less energy is wasted compared with sending the signal to or receiving from all directions. Sending only to the direction of the receiver also reduces distortion to other receivers. Receiving only from the direction of the transmitter increases the signal-to-noise ratio (SNR) of the receivers. This larger SNR can be exploited for energy savings, higher throughput, or less sensitive (simpler) systems, among others. In this thesis we will mainly focus on receiving systems.

Directivity can be achieved by using a directional antenna, such as a dish antenna (figure 2.1), or by using multiple antennas in an array as in phased array systems (figure 2.2). Some options for directional antennas are illustrated in table 2.1. Omni-directional isotropic antennas are hypothetical antennas with an equal directivity in all directions. Dish and aperture antennas achieve directivity by their shape, while array antennas achieve directivity by combining signals from the array which is discussed in more detail below.

Beamsteering (BS) refers to changing the direction of the formed beam. Beam-steering can be achieved mechanically by moving the antenna, or by changing the path length from each antenna to the location where the signals are combined in case of a phased array. Practical reasons allow only a discrete number of different

Figure 2.1: Directivity by dish antennas

Figure 2.2: Phased array

Table 2.1: Beamsteering for different antenna options

Antenna	Steering
Directional antenna	
Omni-directional isotropic	-
Parabolic reflector (dish)	Mechanical
Aperture	Mechanical
Multi-antenna transceivers	
Array	
Fixed plane	Mechanical
Phased array	
Selectable path length	Mechanical/Electrical
Delay or phase correction	Electrical
Smart antenna	
Switched beam	Electrical
Adaptive array	Electrical

path lengths for the mechanical beamsteering option (for each path length a cable of a different length is needed) [109]. With electrical beamsteering, this restriction can be relaxed, allowing faster and more flexible beamsteering. Controlling the direction of maximum sensitivity of the beam is called beamcontrol (BC).

Following [33], smart antennas refer to systems which determine the DoA of a signal by using signal processing. Switched beam systems choose between a number of pre-determined beams, while adaptive arrays allow for complete flexibility in steering the beam (using adaptive algorithms).

2.2.1 *Beamforming*

In this section we will provide a basic outline of the principle of beamforming and relevant terminology.

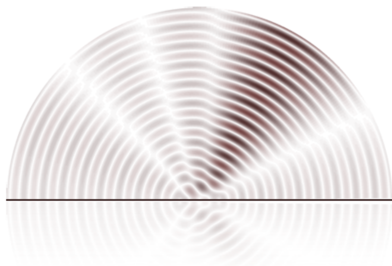


Figure 2.3: Interference pattern

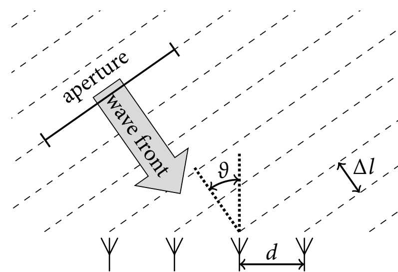


Figure 2.4: Wavefront received by multiple antennas in a phased array

Interference Beamforming is based on the principle of interference. Interference is the pattern resulting from the addition of two or more (partly) correlated waves. A famous example is the double-slit experiment in which a light beam is blocked except for two small slits. The light after the slit is scattered over all directions, i.e. it is comparable to an omnidirectional antenna. As a result the two scattered light waves interfere and the resulting pattern after the slit has a varying intensity as illustrated in figure 2.3. At a location where the light waves from each slit arrive in-phase, the intensity is at its maximum. This is called constructive interference. At a location where the light waves are exactly out-of-phase, the intensity is at its lowest. This is called destructive interference. The same can be accomplished electronically with any two kind of signals by varying phase delays between the signals.

Array A phased array combines the signals from two or more antennas; the array. Whether the waves add up depends on the location of the antennas. For a uniform linear array (ULA), the antennas are located on a straight line with uniform spacing (see figure 2.5). It is therefore a 1-dimensional (1D) array. The uniform spacing simplifies a lot of the mathematics [90, 109].

A ULA is only directional in one dimension, the azimuth direction for a ULA on the x-axis as in figure 2.5. In the dimension orthogonal to this, i.e. in the elevation directions, the array has no directivity, i.e. it is omni-directional. To achieve directivity in two dimensions, a 2-dimensional (2D) array is needed. A common 2D array is a rectangular planar array, for which the antennas are located in a uniform grid (as in figure 2.6).

In general, the antennas can be placed anywhere in a 3-dimensional (3D) space. For such arrays, there is no regularity that can be exploited to simplify the mathematics and the signal from each antenna must be calculated individually.

Phased array systems often use a coordinate system (r, α, γ) slightly different from a spherical coordinate system (r, θ, ϕ). The array is located at the origin of the coordinate system. The distance from the origin is still the range, but *azimuth* (α) is a clock-wise "horizontal" angle (instead of counter-clockwise) and instead of an inclination angle from the zenith direction, an *elevation* (γ) from the horizon is used. The azimuth (α) and elevation (γ) are shown in figure 2.5

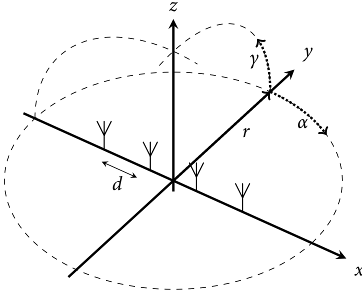


FIGURE 2.5: Uniform linear array

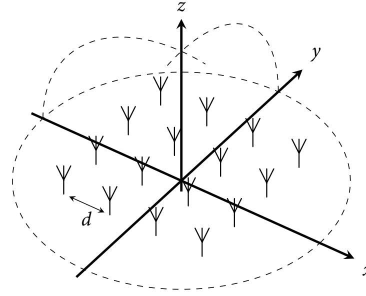


FIGURE 2.6: Rectangular planar array

Planar wavefront Assume a single omni-directional wave source, emitting a spherical waveform s in time and space:

$$s(t,l) = A \cdot \cos\left(\omega t - \frac{2\pi}{\lambda} l\right)$$

with A the amplitude, ω the frequency, λ the wave length, t time and l the path length (distance) from the source. At a large distance, in the far field region, the wavefront of this source arrives almost at the same time at two relatively closely placed receivers (antennas) with their plane perpendicular to the direction of the source, i.e. the path length from the source to each antenna is almost the same. Thus, if we neglect this small error, the wavefront arriving at the receivers can be seen as planar and the two signals add up constructively. Note that we do not consider the polarisation of the wave in this thesis.

Directivity The directivity of an array is dependent on the incident angle (DoA) of the wavefront. If the wavefront arrives at an angle incident to the array (ϑ in figure 2.4) the wavefront arrives at different times at distinct antennas, because of the path length differences between the antennas. This is illustrated in figure 2.4. If the antennas are placed a distance d apart, and if the DoA of the wavefront is at an angle ϑ, the wavefront travels a distance

$$\Delta l = d \cdot \sin(\vartheta)$$

further to the next antenna. This translates in a time delay

$$\Delta t = \frac{\Delta l}{c} = \frac{d \cdot \sin(\vartheta)}{c}$$

between the received signals at the antennas (where c is the propagation speed of radio waves). Depending on the frequency of the wave, this time delay results in a phase shift ($\Delta\phi = \omega \cdot \Delta t$), giving rise to the term "phased array". In the general case the path length between a chosen origin and an antenna element at $\vec{p} = (x, y, z)$ for a

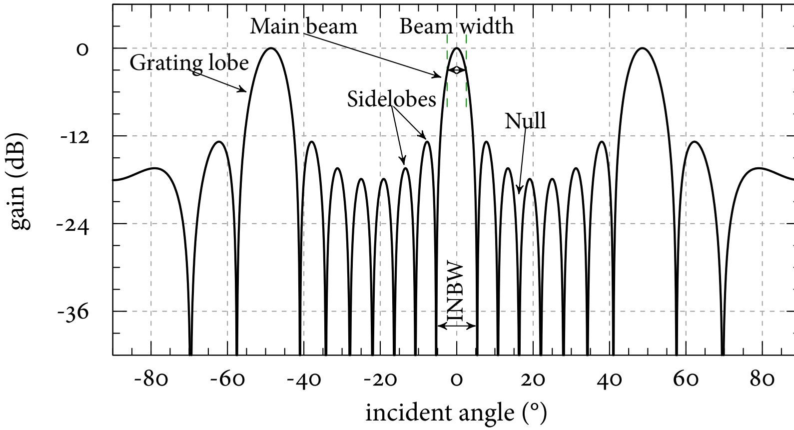


FIGURE 2.7: 2D radiation pattern of an 8-element 1D ULA

source from $\vec{d} = (r, \alpha, \gamma)$ is (calculated with the help of a coordinate transformation of $\vec{d}$):

$$
\begin{aligned}
l &= \left\|\vec{d} - \vec{p}\right\| = \sqrt{\left(r \cdot x_d - x_p\right)^2 + \left(r \cdot y_d - y_p\right)^2 + \left(r \cdot z_d - z_p\right)^2} \\
\Delta l &= \vec{p} \bullet \text{-}\vec{u_d} = x_p \cdot \text{-}x_d + y_p \cdot \text{-}y_d + z_p \cdot \text{-}z_d \\
x_d &= \sin(\alpha) \cdot \cos(\gamma) \\
y_d &= \cos(\alpha) \cdot \cos(\gamma) \\
z_d &= \sin(\gamma)
\end{aligned}
$$

where l is the total path length and Δl is the path length difference with respect to the path length to the origin, determined by the projection of $\vec{p}$ on the unit vector in the direction of $\vec{d}$.

By correcting the path length differences between the antennas, we can influence the directivity of the array.

Radiation pattern Phased array beamforming systems use multiple antennas in an array to make a directional receiver. The directional sensitivity of the array, i.e. the gain of the array versus the incident angle, is called a *radiation pattern* or *beam pattern*. A radiation pattern for an 8-element ULA is shown in figure 2.7.

A direction of maximum sensitivity is called a *beam* because of its shape. The (half-power) *beamwidth* (HPBW) is the angular range between both sides of the beam where the gain is half (−3 dB) of its maximum. For a ULA it can be estimated by [38]:

$$
\text{HPBW}(\vartheta_0) \approx \arcsin\left(\sin(\vartheta_0) + 0.4429\frac{\lambda}{Nd}\right)
$$

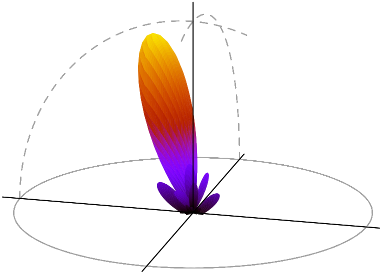


Figure 2.8: 3D radiation pattern

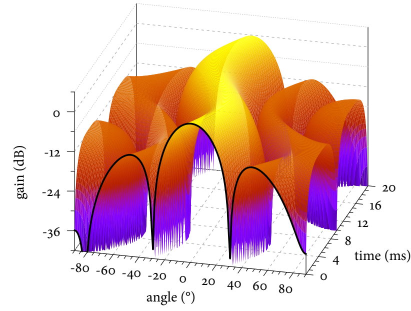


Figure 2.9: 2D radiation pattern over time

with ϑ_0 the steering angle of the beam. The largest beam is called the *main beam*. If there is more than one beam with a maximum gain, the one we are interested in is the main beam and the others are called *grating lobes*. The remaining beams are called *side lobes*. There are also parts where the gain is very small, i.e. a signal from that direction is almost completely attenuated. Those directions are called *nulls* in the radiation pattern. The inter-null beamwidth (INBW) for a ULA is given by [3]:

$$\mathrm{INBW}\left(\vartheta_0\right) = \arcsin\left(\sin\left(\vartheta_0\right) + \frac{\lambda}{dN}\right) - \arcsin\left(\sin\left(\vartheta_0\right) - \frac{\lambda}{dN}\right)$$

Typically, the intent is to steer the main beam in the direction of the signal of interest and to place nulls in the direction of interferers. Note that the HPBW and INBW decrease with an increasing number of antennas N and increase with the incident angle. As the incident angle increases, the area of the wavefront that reaches an antenna (the aperture) becomes less (also see figure 2.4). A smaller (effective) area results in a larger HPBW and INBW, i.e. a wider main beam.

Figure 2.7 shows a 2D radiation pattern in a Cartesian coordinate system. Alternatively we can show a 3D radiation pattern in a spherical coordinate system as shown in figure 2.8. Note that there are no grating lobes in this radiation pattern. If we are steering the beam, it is useful to show how the 2D radiation pattern changes over time. This is achieved by making time the third dimension of the plot as shown in figure 2.9.

Element factor In the previous section we assumed that the antennas radiate or receive the signal from a point source with equal sensitivity, no matter which direction the signal comes from. Such an antenna is called an isotropic radiator and is illustrated in figure 2.10. The gain of an isotropic radiator is defined as:

$$G_E\left(\alpha, \gamma\right) = 1$$

A coherent isotropic radiator is a hypothetical device and is not physically realisable [109]. However, it is useful as an ideal antenna that does not influence the directivity of the array.

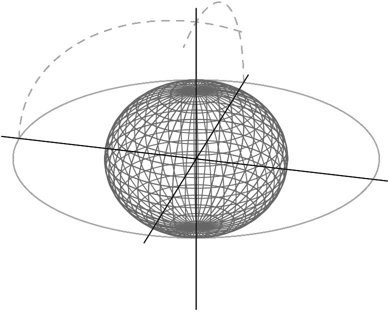


Figure 2.10: sphere radiation pattern

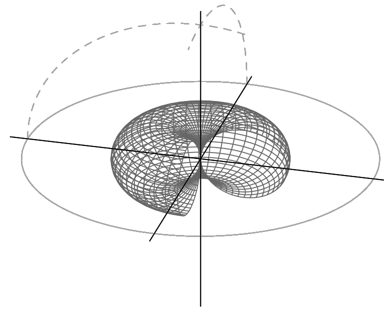


Figure 2.11: torus radiation pattern

A physically realisable and often used antenna is a half-wave dipole antenna. The dipole antenna has a uniform gain in azimuth and a half cosine wave cross-section in elevation, resulting in a torus shaped element factor (shown in figure 2.11):

$$G_E(\alpha, \gamma) = |\cos(\gamma)|$$

Array factor The array factor is the directivity resulting from the phased array including the applied path length corrections (and assuming isotropic elements). The radiation pattern of figure 2.7 shows an array factor without any corrections. The maximum signal amplitude is received for a wavefront perpendicular to the array, thus if we correct the path length difference between the antennas for the angle we are interested in, it is as if the wavefront is perpendicular to the array and the maximum signal amplitude is received when a signal is incident from that angle. We can determine the sensitivity of an array with equidistant elements into each direction α by calculating the array factor [90, 105, 109]:

$$G_A(\alpha) = \sum_{i=1}^{N} e^{j\left(\frac{2\pi}{\lambda_0}(N-i)d\sin(\alpha)+\phi_i(\vartheta_0)\right)} \tag{2.1}$$

with N the number of elements, d the distance between elements, λ_0 the wavelength in free space and ϕ_i the applied correction. In general the array factor is:

$$G_A(\alpha) = \sum_{i=1}^{N} e^{j\frac{2\pi}{\lambda_0}(\Delta l_i+\Delta c_i)}$$

with Δl_i the path length difference and Δc_i the correction for antenna i.

Typically, the antennas are placed a distance $d = \lambda/2$ apart, where λ is the wavelength of the received signal, as for larger distances grating lobes appear as shown in figure 2.7 with $d = 3\lambda/4$. The distance (1D array) or area (2D array) between the outermost elements is the *aperture* of the array. For a larger aperture the beamwidth becomes smaller, so a smaller distance than $d = \lambda/2$ is not preferred.

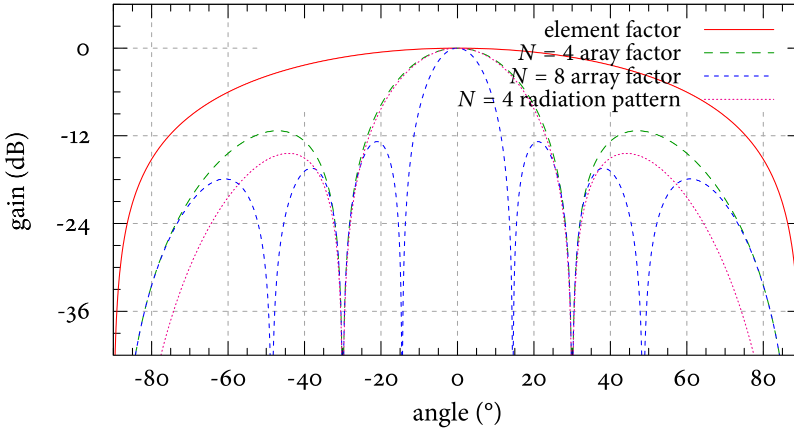


Figure 2.12: 2D radiation pattern

The consequence is that for a larger array, i.e. an array with more elements, the beamwidth becomes smaller and the antenna array becomes more selective. This is illustrated in figure 2.12 for $N = 4$ antennas and $N = 8$ antennas.

The radiation pattern consists of an element factor and an array factor. As the element factor is only a directional gain, if all elements are equal, we can simply multiply its gain with the array factor gain to get the radiation pattern (see figure 2.12):

$$G(\alpha, \gamma) = G_E(\alpha, \gamma) \cdot G_A(\alpha, \gamma)$$

with G_E the element factor and G_A the array factor. For simplicity we will assume an isotropic antenna for the rest of the thesis.

Amplitude and phase taper An *amplitude taper* is a vector of amplitude corrections (one for each antenna) and is used for controlling the *beamshape*. It is comparable to windows for finite impulse response (FIR) filters or fast Fourier transforms (FFTs). For example, we can lower the sidelobes or position a null at the cost of a larger beamwidth. As an example, a triangular amplitude taper is applied for the dashed radiation pattern in figure 2.13, indeed resulting in lowered sidelobes and a widened beam.

From equation (2.1) we find that the phase difference can be corrected by setting $\phi_i(\vartheta_0)$ to $-\frac{2\pi}{\lambda_0}(N-i)d\sin(\vartheta_0)$ for *steering angle* ϑ_0. Thus, the vector with phase correction for a ULA, also called a *steering vector*, is linearly increasing with i. Therefore such a steering vector is called a linear phase taper (LPT). A phase taper is used for beamsteering. Figure 2.13 shows a main beam steered to $\vartheta_0 = -30°$ and to $\vartheta_0 = 10°$ for the dashed radiation pattern.

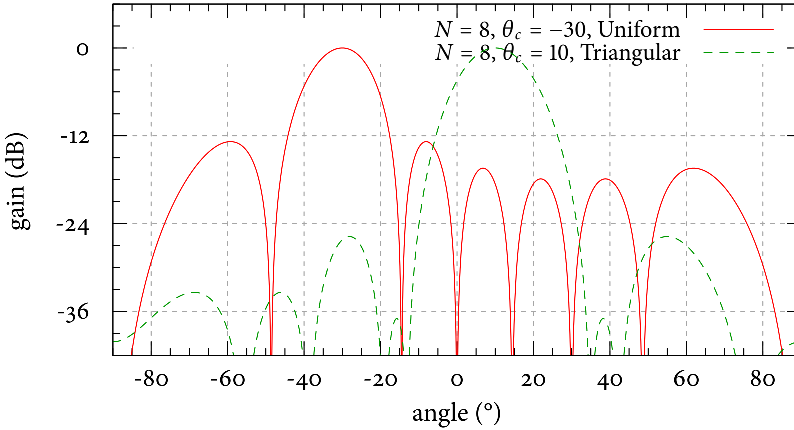


FIGURE 2.13: 2D radiation pattern

Friis transmission equation There are a number of components in a phased array system that have directional gain. For example we have the transmitting antenna, the receiving antenna, the channel between them and the array factor, all contributing a directional gain and possible a phase transfer function. Representing the gain and phase factor as a complex number we can combine their effects by simply multiplying them.

The ratio between the transmitted power and the received power, in which these directional gain factors are included, is given by the *radio equation* or *Friis equation* [109]:

$$\frac{P_R}{P_T} = G_T \cdot G_C \cdot G_R$$
$$G_C = \left(\frac{\lambda}{4\pi r}\right)^2$$

with G_T the transmitter element factor, G_R the receiver element factor, G_C the channel factor and r the range. The radio equation is valid under idealised conditions (aligned antennas, small bandwidth, no multi-path effects) in free space.

We extend the radio equation to include the path length delay ($2\pi\,\delta l/\lambda_0$) of the channel as a function of the DoA:

$$G(r,\alpha,\gamma) = G_T(\alpha,\gamma) \cdot G_C(r,\alpha,\gamma) \cdot G_R(\alpha,\gamma)$$
$$G_C(r,\alpha,\gamma) = \left(\frac{\lambda}{4\pi r}\right)^2 e^{j\frac{2\pi}{\lambda_0}\Delta l}$$

This represents the transfer function from a source to a single antenna. Later we will use this to combine the signals at a receiver antenna from multiple sources and to apply a delay correction for the path length difference between each source and antenna element of the receiver.

2.2.2 *Beamsteering*

In phased arrays there is a path length difference in the path from a source to the different antennas of the array. This path length difference results in time delay between the antenna signals:

$$\Delta t(r, \alpha, \gamma) = \frac{\Delta l(r, \alpha, \gamma)}{c}$$

For beamsteering we correct the time delay in a certain direction (α_0, γ_0) so that signals from that direction add up coherently:

$$\Delta t_c = -\frac{\Delta l(\alpha_0, \gamma_0)}{c}$$

Consider a delay function $D_\delta(s)$, that delays a signal s by δ, with $\delta_i = \Delta t_{c,i}$. The beamformer is then defined as:

$$y = \sum_i^N D_{\delta_i}(s_i)$$

If the signal is a narrowband signal, this time delay can be approximated with a phase shift:

$$\Delta\phi = \omega_0 \cdot \Delta t_c$$

hence by applying the inverse phase shift, the time delay is corrected. A gain and phase shift is typically represented as a weight w_i. A phase-shift based beamformer applies a correction weight w_i^* to each antenna signal s_i and sums the results:

$$y = \sum_i^N w_i^* \cdot s_i = \sum_i^N a_i \cdot e^{j\omega\Delta\phi_i} \cdot s_i$$

with w_i^* the complex conjugate of w_i. This can also be represented using as a matrix multiplication of a weight vector or *steering vector* $\vec{w}$ and a vector of the antenna signals $\vec{s}$:

$$y = \vec{w}^H \cdot \vec{s}$$

with $\vec{w}^H$ the Hermitian or conjugate transpose of $\vec{w}$.

For frequencies slightly different than ω the error ϵ is:

$$\epsilon_\phi = (\omega_0 + \Delta\omega)\Delta t - \omega_0\Delta t = \Delta\omega\Delta t$$

A better approximation can be achieved if we apply different phase shifts for different frequencies.

Multiple beams can be formed by using the same antenna signals with different delays or different steering vectors, i.e. we use multiple beamformers on the same input signal. Each beamformer has its own radiation pattern, which are in the general case independent (see below). Therefore, a phased array is very flexible:

with the same array additional beams can be formed and steered, only at the cost of additional hardware or processing.

There are different options for achieving a time delay or phase shift, which we will discuss next. In practice all options approximate a time delay (or path length) to some degree, resulting in a large number of beamformer structures which have different characteristics. They differ in support for narrowband or wideband signals, accuracy and complexity [3, 33, 50].

2.2.2.1 Time delay

As the time delay exactly corrects the path length difference experienced by the wavefront, it is not dependent on the (frequency of the) transmitted signal. Therefore, any source signal, including wideband signals, can be used. The disadvantage is that implementation is difficult. Note that the time delay can be arbitrary small as the incident angle becomes closer to 0°. The largest time delay is caused by the two antennas which are the furthest away and therefore scales with the array size. Next, we will discuss three options for implementation.

Physical delay The time delay between antennas can be corrected for by using different path lengths between the antennas and the location where the signals are added. For beamsteering, these path lengths must be changeable. Therefore, many different routes with different path length must be implemented and switched between. This makes implementation difficult and costly.

Time shifted sampling Instead of physically changing the path length, one can also adjust the sample moment of each analogue-to-digital converter (ADC), so-called *time shifted sampling*. However, as part of the beamforming is performed by the ADCs of which there is only one per antenna signal, only a single beam is supported. Furthermore, time-shifted sampling ADCs are not standard and require fine-grained control of the delay.

Buffering and interpolation A different option is to use buffers to temporarily store samples. Such an implementation, as with changeable path lengths, can only implement a discrete set of time delays. We can approximate an arbitrary time delay by using interpolation between available samples or by using sampling rate conversion techniques [77]. However, for small delays, a time approximation by interpolation is complex and difficult [51, 77, 103]. In general, the higher the sample rate the better the delay approximation. One choice for interpolation is to use an all-pass filter with a linear group delay [83] (which must be tuneable for steering). This filter can be in the analog domain or the digital domain. In the digital domain, a FIR filter can be used, which consists of complex multiplications followed by summing the results. A FIR filter corresponds to performing a truncated convolution.

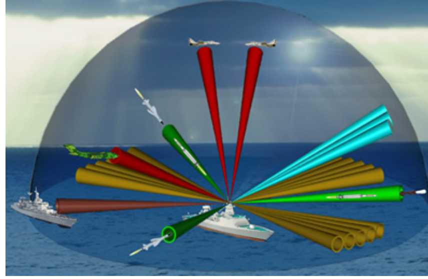


Figure 2.14: FFT based beamforming in elevation

2.2.2.2 Phase shift

A phase shift can be used to approximate a time delay. A time delay is independent of the frequency, but the phase shift resulting from a time delay does change over frequency. As found before, a fixed phase-shift equals a time delay only at a single frequency and the time delay approximation error becomes larger for frequencies further away. A phase-shift approximation is therefore only suitable for narrowband signals. However, its implementation is simpler; we will discuss two options, both use a complex representation of signals requiring a Hilbert transform.

Complex multiplication When using a complex representation of signals, a phase shift is simply a complex multiplication:

$$a \cdot e^{j\omega t+\phi} \cdot e^{j\Delta\phi}$$

Phase shift based beamforming for N antennas consist of N complex multiplications with the steering vector and summing the results, giving a single beam (pattern). For multiple beams, we just repeat the process with different steering vectors.

Spatial FFT An FFT can be seen as performing N parallel convolutions or filters with the same filter shape. As such, an FFT over the antenna samples performs a spatial filter, resulting in N beams. These beams have a fixed relative position and equal shape, i.e. the main beams are non-overlapping as with FFT bins in the frequency domain. However, they can be calculated in $\mathcal{O}(Nlog(N))$ instead of $\mathcal{O}(N^2)$ for N "normal" beams.

As the main beams are non-overlapping, with each top of a main beam exactly at the position of a null for all other beams, the resulting radiation pattern resembles a number of beams stacked next to each other, a so-called *fan-of-beams*. This is illustrated by the lighter beams in figure 2.14. A fan-of-beams is typically used for searching or scanning. The darker beams are single beams used for tracking.

The window (amplitude taper) used for the FFT determines the shape that the signal is convolved with. Without a window (meaning a rectangular window), this filter shape is a sinc function.

In order to differentiate between performing an FFT on a sequence of signals in time, we will refer to this method as a *spatial* FFT.

Note that the complex multiplication and spatial FFT can operate on real input signals, but in that case the result of positive angles overlap with the mirrored negative angles [83]. To differentiate between those, a complex signal must be used.

2.2.2.3 Hilbert transform

To differentiate between positive and negative angles, the phase-shift and FFT based beamformers use complex antenna signals. A complex number is an ordered pair, which can be seen as a coordinate pair in a complex plane. Euler's formula relates a complex number to an orthogonal sine and a cosine pair. A complex signal can thus be represented by an in-phase version and a 90° phase shifted quadrature version (see [80]).

However, the antenna signals are real. To get a complex representation, a Hilbert transform, which corresponds to a 90° phase shift for all frequencies, is performed on the antenna signals to get the quadrature signal. Together with the original (in-phase) signal, this gives a complex antenna signal.

A Hilbert transform can be performed with a quadrature mixer (typically in the analogue domain as we often need analogue frequency conversion anyway) or with a filter (typically in the digital domain for flexibility, stability and an equal power in both paths). For an accurate phase shift over all frequencies the filter order must be high, making a Hilbert filter computationally expensive. Therefore a quadrature mixer is to be preferred, at the cost of twice as many ADCs, albeit at half the sample rate [83]. At the same time, it is difficult to make accurate wide-band quadrature mixers; resulting phase errors are often a reason to employ digital implementations.

2.2.3 *Delay at baseband*

A typical beamforming system contains (frequency) down-conversion of the RF signal from the channel to a more manageable intermediate frequency (IF), i.e. baseband, through mixing with a local oscillator (LO). We will discuss how the time delay (and phase shift) between the signals from the channel translates to an equivalent delay at baseband.

If the antennas are at a distance $d = \lambda/2$, the phase difference between two adjacent antennas is between 0 and π for a 0° to 90° DoA. For a large array the time delay between the two outer antennas can thus become quite large relative to the RF frequency, i.e. the time delay is equal to a number of a periods.

However, for beamforming at the IF, this delay translates to a much lower frequency. At IF we have:

$$s(t) = A\cos(\omega t + \varphi)$$

$$s_{RF}(t)s_{LO}(t) = \frac{A_{RF}A_{LO}}{2}\Big[\cos\big((\omega_{RF}+\omega_{LO})t + (\varphi_{RF}+\varphi_{LO})\big) + \cos\big((\omega_{RF}-\omega_{LO})t + (\varphi_{RF}-\varphi_{LO})\big)\Big]$$

For a time delay Δt and a frequency around RF ($\omega = \omega_{RF} + \Delta\omega$), it follows:

$$\begin{aligned}
s_{RF}(t) &= \cos\big((\omega_{RF} + \Delta\omega)(t + \Delta t) + \varphi_{RF}\big) \\
&= \cos(\omega_{RF} t + \Delta\omega t + \omega_{RF}\Delta t + \Delta\omega\Delta t + \varphi_{RF}) \\
s_{IF}(t) &= s_{RF}(t) s_{LO}(t) \\
&= A_{IF}\Big[\cos\big((\omega_{RF} + \Delta\omega - \omega_{LO})t \\
&\qquad\qquad + (\omega_{RF} + \Delta\omega)\Delta t + (\varphi_{RF} - \varphi_{LO})\big) + \dots\Big] \\
&= A_{IF}\Big[\cos\big((\omega_{RF} - \omega_{LO})t + (\varphi_{RF} - \varphi_{LO}) + \Delta\omega t \\
&\qquad\qquad + \omega_{RF}\Delta t + \Delta\omega\Delta t\big) + \dots\Big]
\end{aligned}$$

where $(\omega_{RF} - \omega_{LO})\, t + (\varphi_{RF} - \varphi_{LO}) + \Delta\omega t$ is the desired signal, while $\omega_{RF}\Delta t + \Delta\omega\Delta t$ is negated by a time delay correction, but only $\omega_{RF}\Delta t$ is negated for a phase delay correction.

From the above we can make two observations. Firstly, the time delay is the same at IF as at RF, which means the time delay is small relative to the IF. So a time delay, as a phase shift, is transparent to mixing. Note, however, that a time delay correction at IF, i.e.:

$$s_{IF}(-\Delta t) = A_{IF}\Big[\cos\big((\omega_{RF} - \omega_{LO})(-\Delta t) + \Delta\omega(-\Delta t)\dots$$

indeed corrects the relevant terms, but also includes the term $-\omega_{LO}(-\Delta t)$. Therefore, a time delay correction also comprises an additional phase shift of the signal. As the applied time delay correction differs per antenna for beamforming, this phase shift is also different per antenna signal and should be corrected.

Secondly, a phase shift correction leaves an error term of $\Delta\omega\Delta t$, which becomes larger for larger $\Delta\omega$, i.e. for a larger bandwidth around the carrier. Thus phase shift based beamforming is only suitable for signals with a small bandwidth (as we found in section 2.2.2) compared to the RF. Also, there is a constant phase shift resulting from the initial phase of the LO (φ_{LO}), which should be synchronised among the antennas, or included in the phase shift correction.

2.2.4 *Narrowband and wideband*

Narrowband signals have a small fractional bandwidth; the bandwidth of the signal is small compared to the carrier or median frequency. What is considered to be "small" depends on the application and allowable errors. We will follow [3] and call signals with a fractional bandwidth of less than 1 % *narrowband*:

$$\frac{f_h - f_l}{(f_h + f_l)/2} \times 100\% < 1\%$$

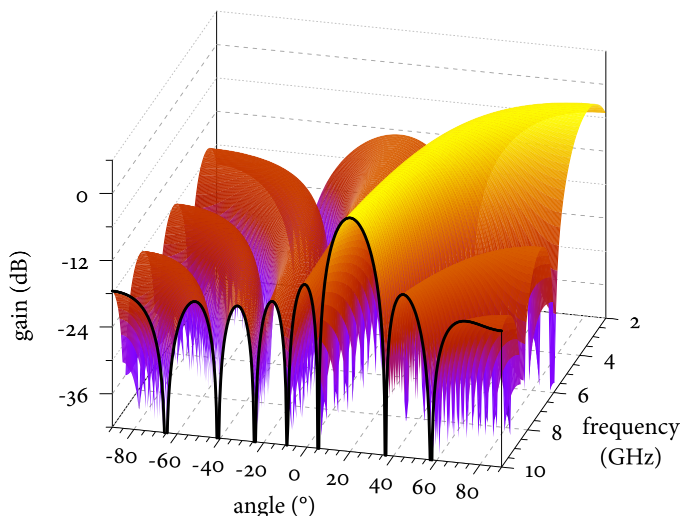


Figure 2.15: Beam squint

with f_h the highest and f_l the lowest frequency used. Signals with a larger fractional bandwidth are called *wideband*. Bandwidth is relevant to beamforming in at least two ways.

First, the distance d between antennas is fixed, but the phase shift experienced for this fixed distance is dependent on frequency. The distance is set to $\lambda/2$ to avoid grating lobes. Furthermore, $\lambda = c/f$, so the smallest distance is set for the highest frequency. As such, lower frequencies have a smaller effective area or aperture than the highest frequency. This in turn results in a broadening beamwidth and nulls moving outward for lower frequencies as can be seen in figure 2.15.

Second, if we apply a phase shift correction for steering the beam, we have seen that the phase shift corrects the time delay and thus path length difference *exactly* for only one frequency. The larger the difference in frequency, i.e. the larger the bandwidth, the larger the error. This error can also be interpreted as mispointing and is called *beam-squint*; the actual beam direction is different from the intended direction and changes over frequency as clearly visible in figure 2.15. Beam-squint does not happen when a time delay correction is used, as a time delay exactly corrects the path length difference independent of the frequency.

As a result, a different definition for a narrowband signal can be used. A signal is narrowband up to the largest bandwidth for which the error with phase-shift based beamforming is negligible. What is considered to be “negligible” again depends on the applications. We will choose an amplitude error of 1 ‰ or −60 dB. Otherwise, the signal is wideband and needs time delay (approximation) based beamforming.

2.2.5 *Phased array system characteristics*

From the discussion of the previous section, we can deduce a number of characteristics of phased array systems. We found that an array is most sensitive in the direction for which the delays at the different antennas are corrected.

Furthermore:

- If we increase the array size (by increasing N or d), the beam-width of the main beam decreases (HPBW), which results in a higher angular resolution.
- A larger incident angle results in a larger HPBW, i.e. a wider main beam.
- If we achieve a more precise time delay or phase shift correction for each antenna, we achieve a higher angular (α,γ) precision, i.e. deeper nulls and more accurate beam-directions.
- For a higher sampling (measurement) rate, we achieve a better time resolution and a more accurate steering and therefore signal.
- The time delay between the antenna signals is transparent to frequency conversion.

As mentioned, the delay of each antenna must be corrected for. The two basic options are:

- a time delay approximation, which is suitable for wideband signals but complex,
- a phase shift by a complex multiplication, which requires complex signals but once we have those, the number of generated beams is easily increased at low computational cost,

For the phase shift option, we can also compute many beams at once efficiently with:

- a spatial FFT, which also needs complex signals but efficiently computes a fan-of-beams.

As each beamforming method has advantages and disadvantages it is useful to support all three of them if enough computational resources are available.

2.3 Generic beamforming platform

In section 2.2 an overview was presented of the advantages of using phased array beamforming. Beamforming can be beneficial for any radio system, such as satellite reception, radar, (radio) astronomy or mobile (3G/4G) and wireless (WLAN/WiMax) communications. However, radar and radio astronomy applications use large arrays with high cost and low production volume, while mobile and wireless communication for consumer applications needs to be low-cost. If we can design a platform that is flexible and modular and therefore generic and scalable, we can support all these applications with a single generic platform. Such a generic platform could lower the cost by sharing development and production costs and by enabling higher production volumes.

Table 2.2: Overview of applications where beamforming can be beneficial

	Satellite reception	Radar	Radio astronomy	Mobile and wireless communications
Number of antennas	256	4096	7392	64
Number of beams	3	20	24	32
Frequency (GHz)	10–13	7–13	0.01–0.24	2–6
Bandwidth (MHz)	50	100	100	1-30
SNR dynamic range (dB)	16	100	70	30
ADC (bits)	4	16	12	10

Figures for radar, radio astronomy and wireless base stations are based on current requirements and extrapolated to the near future.

In this section we will propose a design for such a generic beamforming platform. This design will also form the basis of the beamforming systems used in chapters 3 and 6. We will first analyse the above mentioned applications and derive the requirements of the platform. Next, we will present a system design of a typical beamforming system, that will form the basis of the generic platform. We aim for an IC beamformer for cost reasons, which is typically a monolithic component and therefore not scalable. To achieve a scalable system, we propose a hierarchical beamformer that splits up beamforming into multiple stages. Finally, we discuss a hybrid beamformer, for which the beamformer includes analogue stages.

2.3.1 *Applications*

A comparison of the requirements of a beamforming system for satellite reception, radar, radio astronomy and mobile and wireless communication is given in table 2.2 and further discussed below.

Satellite reception Digital television broadcasts are transmitted by many different satellites, orbiting in a fixed position with respect to the earth. Satellites are used to reflect an uplink signal to a large region on earth. Satellite positions in Europe range from about 20° to 50° elevation and 5° to 30° azimuth. Since a satellite has to operate for many years in space, it cannot be equipped with batteries and hence uses solar energy for the transmission. Therefore, the transmitted power is limited.

Satellite systems require line-of-sight between transmitter and receiver. Therefore, multi-path effects are assumed to be negligible. A satellite transmits multiple unique data streams. To maximise the usage of such data streams, multiple TV programs are compressed and multiplexed in the stream.

The DVB-S standard [28] specifies a frequency of 10.7 GHz to 12.75 GHz, a maximum SNR of 16 dB (−2 dB minimum), a channel bandwidth up to 36 MHz (effective bandwidth used is 50 MHz due to pulse shaping filter roll-off) and satellites that are at least 5° apart. The modulation technique used for individual DVB-S channels is quadrature phase-shift keying (QPSK). QPSK uses four different phases to represent transmitted information, equally distributed on the unit circle of the

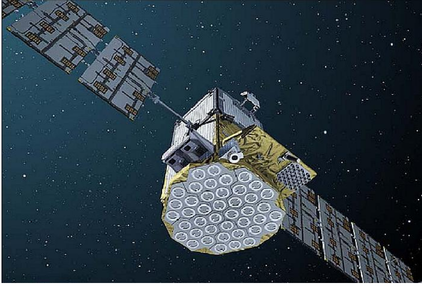


Figure 2.16: Phased array in satellite reception

Figure 2.17: Phased array in radar

complex plane [40]. Each of these four phases represents a symbol, which represents two data bits. Since the transmission of two subsequent symbols requires instantaneous phase shifts in the transmitted output signal, high frequency components are introduced. A pulse shaping filter is used to decrease the effects of these phase shifts by spreading the signal into a slightly larger frequency band such that the high frequency components are attenuated.

Conventionally, DVB-S receivers use a parabolic dish antenna, which can be constructed easily and have a high efficiency. A dish antenna focuses a wavefront incoming from a single direction to one focal point. The disadvantages of satellite dishes are that they must be aimed mechanically and the dish must be at a fixed position (stationary), as continuous steering is problematic because of the size of the dish and wear and tear of the mechanics. This makes the dish unsuitable for moving (or often relocating) vehicles, such as a car or yacht.

A phased array system can therefore be beneficial. A mobile environment is not a problem as it is fully steered electronically. A 16×16=256 element array is expected to be a reasonable size for cost reasons; a smaller array is not selective enough, while a larger array requires additional antennas and processing making the satellite receiver more expensive and therefore less competitive as a consumer product. As satellites are at least 5° apart and we need 16 dB SNR, a beamwidth of 10° at −16 dB is needed. This is not possible as the beamwidth with a 16×16 array is already 11° at 90° elevation and increasing at lower angles. If we allow one grating lobe (or use 4 times as many antennas), the beamwidth is small enough down to 35° elevation. The phased array size is then about 0.5 m by 0.5 m. Broadcasts from multiple satellites can be received simultaneously by enabling for two or three independent beams. This is useful, when multiple users want to receive signals from different satellites [104].

Radar The main purpose of radar systems is to detect, locate and follow reflecting objects or targets. It is for example used for scanning, tracking or guiding objects. A typical radar system uses short periods of pulses to scan its environment. Since the actual moment of transmission is known, the receiver is only used during a certain time frame shortly after the transmission of the pulse. By measuring the

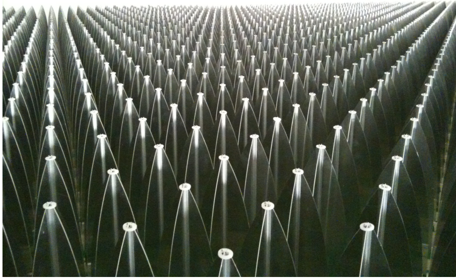


Figure 2.18: Phased array in radio astronomy (EMBRACE)

Figure 2.19: Phased array in communications

time between transmission and reception of the reflection, the distance to the target can be calculated.

Phased arrays for radar have been used since the 1950s [109]. Current systems use separate antennas and front-ends produced in specialised processes, making the antenna front-ends costly because of the relatively low production volumes. Furthermore, traditionally a large amount of specifically designed central processing is used. This makes the system neither scalable nor energy efficient [90].

Future phased array radar systems require a large array size (upto thousands of antennas), a high SNR (100 dB) and a high sample rate (100 MHz). We will consider radar systems using 7 GHz to 13 GHz signals. At each moment in time, there might be multiple interesting objects in sight. Therefore, tens of objects are scanned or tracked simultaneously, requiring at least that many independent beams.

Radio astronomy The aim of radio astronomy is to construct images of celestial objects. Similar to satellite reception, radio astronomy is a receiver-only application. The energy radiated by celestial bodies is picked up and analysed, among others by doing long-term correlation and integration of the signal. Therefore, the received signal and steered direction have to be very stable. As the objects of interest are very distant, the beamwidth must be as narrow as possible and as the received signals are very weak the antennas must be as sensitive as possible.

Traditionally, very large dish antennas are used. However, structural/mechanical limitations constrain their size. A phased array is not limited by this constraint. Furthermore, phased arrays can track multiple objects at once and can be dynamically steered much easier and faster than dish antennas.

The narrow beamwidth and high stability and sensitivity requires a very large array with accurately calibrated antenna processing. The figures in table 2.2 are based on the low frequency array (LOFAR) [21, 22, 36], a phased array for radio astronomy that is currently being made operational. LOFAR consists of 77 stations with 96 antennas each and can create 24 simultaneous beams. Note that a larger SNR allows differentiation of strong and weak objects, while more sensitivity and longer correlation allows for detection of fainter objects.

Mobile and wireless communications During the last few years, multi-antenna techniques have been introduced in the latest wireless standards (e.g. IEEE 802.11n). Such multiple-input multiple-output (MIMO) systems allow for higher channel utilisation, as their transmission techniques are designed for both spatial and spectral optimisation. Beamforming is one of the techniques that can be used to optimise the spatial use of the spectrum. Mobile and wireless communication systems heavily suffer from multi-path effects, which can be reduced with spatial filtering.

Nowadays, receivers typically use two or three antenna elements at most. Adding more antenna elements implies that multiple additional front-ends are to be included, which makes a portable receiver less compact and efficient. For the base station, however, beamforming is a useful technique as the transmitted power can be spatially controlled such that the beam is focused at receivers. Additionally, independent beams can track individual users, limiting interference and energy.

Base station are commercial products for a large volume market, which also implies that cost is important. Therefore, we assume the number of antennas to be limited, but with a maximum number of beams for such limited arrays. Most wireless and mobile communications operate from around 2 GHz to 5 GHz, using up to 5 MHz bandwidth per channel and up to 60 dB SNR [9].

2.3.2 *Requirements*

From the short survey of applications it is evident that there are large differences in array size. Therefore, a generic beamforming platform must be scalable and modular. Beamforming requires a substantial amount of signal processing on streaming data, but many applications also need to track objects requiring control (e.g. tracking) algorithms with low computational cost. A generic beamforming platform must thus be flexible enough to support control algorithms and efficient enough for processing the antenna signals. Such a large amount of processing also requires an energy-efficient design. The specifications of radar front-ends are sufficient for all applications if cost can be low enough. This can be achieved, for example, with a software-defined radio design. In a hierarchical system, beamforming is performed in multiple stages, reducing the requirements of later stages because interferers can be reduced in the first stages and signals are combined, in each stage, reducing the number of signals. Analogue beamforming may be used in the first stage reducing the number of needed ADCs and processing, resulting in a hybrid beamforming system. The disadvantages of a hierarchical design are distributed control and extra calibration. Hierarchical and hybrid beamforming and beamcontrol algorithms are further discussed in this chapter, while a scalable, flexible and efficient processing architecture is further discussed in chapter 3.

2.3.3 *System design*

In this section we will discuss a basic phased array beamforming system design. For most applications, the received (RF) signal is at a high carrier frequency up to 13 GHz, which must be down-converted to an IF for further processing. This is done

by mixing the RF signal with an LO signal [80]. Beamforming can be performed at several stages in this design, which we will discuss first. Then we will present a block diagram of a typical system and its environment, and a short discussion on the components of the block diagram.

2.3.3.1 Beamforming location

A time delay or phase shift correction can be performed at all the signal-paths of the down-conversion step, i.e. at RF, at the LO or at IF. Each location has advantages and disadvantages [106].

RF beamforming As beamforming combines signals together, RF beamforming requires fewer down-conversion stages, saving hardware and cost. The disadvantage is that design at RF is difficult and must be fully analogue [80]. Furthermore, the RF front-ends have to be duplicated for additional beams. For these reasons it is not considered further in this thesis.

LO beamforming A phase shift of the signal of interest can also be implemented by setting the initial phase of the LO for each antenna to its phase correction, i.e. the initial phases are the steering vector. The combining of signals is performed at IF. The advantage is that the phase shift operation is out of the main signal path, thereby not introducing extra noise and distortions. The disadvantage is that the timing and distribution of each LO is critical; for correct beamforming, the timing of all LOs must be synchronous. Also, only a phase shift correction can be applied, not a time delay. Therefore, LO beamforming is also not considered further.

IF beamforming Beamforming can also be performed at IF, after down-conversion. Timing of the LO is still critical, but now the beamforming is performed at a lower frequency. However, a disadvantige is that the time delay between the signals is small relative to the IF, as it is transparent to mixing (see section 2.2.3). The major advantage of IF beamforming, and also the reason why we choose for IF beamforming, is that the signal can first be digitised before beamforming. This allows for the flexibility to use the same processing hardware for multiple applications. Digital beamforming is only feasible at IF, because of the ADC requirements. ADCs with both a high sampling rate and a high dynamic range (in bits) are either not feasible or very costly and power hungry, requiring down-conversion before the ADCs [83].

2.3.3.2 Block diagram

The block diagram of a basic beamformer system, based on digital beamforming, is shown in figure 2.20. The system consists of two major components: analogue front-ends and digital processing. The (relevant) environment of the system also is shown, illustrating the signal characteristics of the signals received by the antennas. Signal generation, modelling the environment, is used for verifying the design. The components in figure 2.20 are discussed in more detail in the following sections.

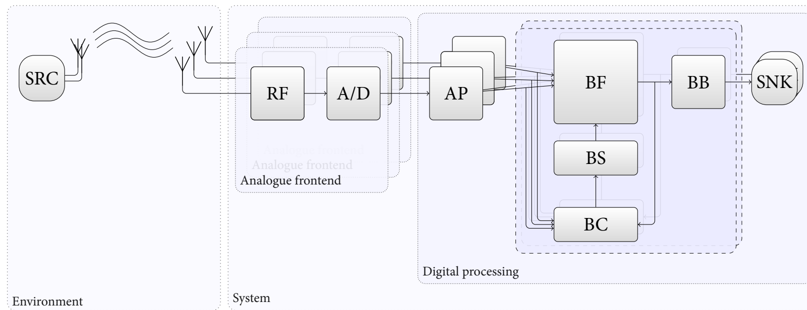


Figure 2.20: Basic phased array system

2.3.3.3 Environment

A phased array receives signals from multiple sources. Some can be considered signals-of-interest, others are interferers or noise. Combined they form the signals that are received at the phased array antennas. Signal generation modelling or emulation is needed during the system design, to test the phased array receiver.

We will consider one or more signal sources. Each transmitted signal travels over a channel (indicated by the waves in figure 2.20) to each of the receiver antennas, thereby experiencing a delay and attenuation and the addition of noise. Thus, a separate channel for each transmitter-receiver combination is used. The signals from all channels to a single antenna are combined at that receiver and form the input for the analogue front-end.

2.3.3.4 Analogue front-end

After reception at the antenna elements, the RF front-end performs down-conversion and possibly amplification and image rejection [80]. Next, sampling and quantisation is performed by an ADC. The (digital) signals from multiple analogue front-ends form the input for the digital processing.

2.3.3.5 Digital processing

The digital processing consists of antenna processing (AP) on individual antenna signals, followed by beamforming processing and further (application dependent) baseband (BB) processing.

Antenna processing For accurate beamforming, it is important that the gain and delay of the signal from each antenna is equal (except for the path length difference), i.e. the distortions from the antennas and front-ends should be corrected. To realise this, antenna processing, e.g. calibration and/or equalisation, is applied.

A non-ideal antenna and front-end needs to be fine-tuned to compensate for errors occurring due to non-idealities. For example:

- the antenna position and shape might not be as specified,
- the RF front-ends are not perfectly matched and introduce non-linearities,
- LO and ADC clocks may be slightly out of phase.

Calibration consists of a static gain and phase correction, which is determined by comparing the measured results from a known reference signal to the expected results. Equalisation is more advanced as it applies a gain and phase correction over a frequency range, by using a filter. For distortions that vary over time, we need periodic calibration or equalisation or we can use adaptive feedback.

As explained in section 2.2.2, in some systems a Hilbert transform is needed to reconstruct the phase information of the sampled signal. An ideal Hilbert transform is not possible as it is non-causal and of infinite length [77]. It is approximated with a FIR filer, where the minimum order can be determined based on the required SNR, filter roll-off and bandwidth [77].

So, typically for antenna processing a FIR filter is used. As a filter is needed for each antenna, the amount of processing can easily become as large as the beamforming itself. However, it is independent of the number of beams that are formed.

Beamforming The beamforming (BF) processing (beamformer) applies a time delay or phase shift correction and sums the signals, or uses an FFT. Note that the same antenna signals can be used to form multiple beams in different directions simultaneously. Therefore, for each beam, the antenna signals are combined with different correction parameters. This requires a dedicated instance of the beamformer and beam control parts for each beam formed simultaneously.

Beamsteering The BS processing calculates the time delay or phase shift correction (as well as the gain) to be applied by each antenna. The beamsteerer thereby defines the direction and shape of the formed beam.

Often the beamsteerer is combined with either the beamformer, which has the advantage that only the shape and direction needs to be communicated while the correction parameters calculation is local to the beamformer, or with the beamcontroller as part of the control algorithm.

Beamcontrol To calculate the correction parameters, the beamsteerer needs to know in which angle (direction) to point the beam (and optionally which beamshape to use). This information is provided by the BC processing.

The beamcontroller processing can simply scan an area, or determine the direction by searching for a source, for example based on an estimation of the angles of the strongest sources available, or by tracking a source, for example using adaptive feedback. Beamcontrol is further discussed in section 2.4.

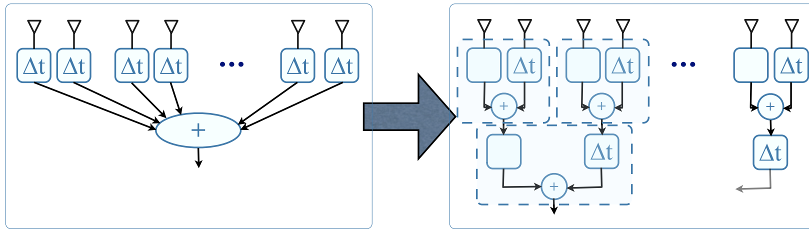


Figure 2.21: Dividing the beamforming operation into multiple stages

Baseband processing The beamformer, which has spatially filtered out the signal-of-interest, is typically followed by application dependent BB. Typical baseband processing operations are detection, demodulation, error correction and decoding.

For example, radar baseband processing consists of (matched and Doppler) filtering and detection to determine the presence, location and speed of an object. Another example is the DVB-S applications. A DVB-S satellite signal is QPSK modulated, and is filtered by a matched filter, followed by demodulation and error correction at baseband [28].

2.3.4 *Hierarchical beamforming*

A beamformer can be split up into multiple stages, so-called hierarchical beamforming. When beamforming is performed in multiple stages, the array is divided into sub-arrays which are independently beamformed, while the next stage(s) combines the signals of the subarrays. Such a scheme is illustrated in figure 2.21 for time delay based beamforming.

Hierarchical beamforming is used to make the beamformer scalable, as a single monolithic beamforming operation is split up into several smaller beamforming operations. By normalising the time delays (or phase shifts) of the sub-array elements to their first element, this element has a zero time delay (also illustrated in figure 2.21). Therefore, the total number of time delays remains the same for both the monolithic beamformer and the hierarchical beamformer.

Hierarchical beamforming has a number of advantages. The beamforming operation is modular and scalable. Further, the beamforming operation combines signals, thereby reducing the number of signals and amount of processing for later stages (assuming the number of beams is less than the number of elements of a subarray). For a monolithic beamformer, all antenna signals are communicated to a central location, making it a bottleneck. For a hierarchical beamformer, the first stages can be distributed and moved closer to the antennas, thereby reducing the communication bottleneck. However, the distributed processing and communication makes the design more complex. For example, a disadvantage is that the steering vector from the beamsteerer or the steering angle from the beamcontroller must be distributed to each stage and sub-array. Furthermore, the time delays or phase shifts must be normalised for each stage. Finally, as the first stages combine

antenna signals, not all antenna signals are available at later stages. This is an advantage but also a disadvantage if a beamcontrol algorithm computes a steering vector based on a later stage, as we will then only have weights for the elements of that stage. We will come back to this in section 2.4.

We will assume the subarrays consist of adjacent antenna elements and are all steered equally. The sub-arrays are therefore smaller than the original array, so the main beam and sidelobes are wider. Assume a 4×4 sub-array with antenna elements $\lambda/2$ apart. For the next stage, the signal from the sub-array could as well be from a single antenna element, as it is already combined into a single signal. However, these "virtual" antenna elements are four times farther apart in each direction (at 2λ distance) then the original array. This results in grating lobes for the second and later stages.

As we will rely on hierarchical beamforming for partitioning the beamforming application in chapters 3 and 6, it is defined formally as follows. Again consider a delay function D_δ, that delays a signal s by δ. The delay, like addition and multiplication, can be applied in multiple successive steps, i.e.:

$$D_\delta(s) = D_{\delta_1}(D_{\delta_2}(s)), \text{ where } \delta = \delta_1 + \delta_2$$

Let N be the number of "real" or "virtual" antenna elements and M be the number of elements in a sub-array or part in the partitioning. Hence, for a beamformer:

$$\ni(\vec{s},\vec{\delta}) = \begin{cases} \sum_{i=0}^{N-1} D_{\delta_i}(s_i) & \text{, if } N < M \\ \sum_{i=0}^{M-1} D_{\Delta_j}(S_j) & \text{, otherwise} \end{cases}$$

where

$$\begin{aligned} \vec{S}_j &= \widehat{\ni}\ \overrightarrow{(\vec{u}_j, \vec{\eta}_j)} \\ \vec{u}_j &= [s_{(j\cdot M)} \dots s_{((j+1)\cdot M-1)}] \\ \vec{\Delta}_j &= \delta_{(j\cdot M)} \\ \vec{\eta}_j &= [0 \dots \delta_{((j+1)\cdot M-1)} - \Delta_j] \end{aligned} \quad \text{, for } j = \left[0 \dots \lceil \frac{N}{M} \rceil\right]$$

where $\ni$ is the beamforming operation and $\widehat{\ni}$ applies the beamforming operation to each element of a vector.

The list of signals $\vec{s}$ is split into M element sub-vectors if it is larger than M, forming the sub-arrays of the previous stage. For each sub-array only the time delays between the elements in the sub-array with respect to a reference element are corrected, while the time delays between the reference elements are corrected by the next stage. Therefore $\vec{\eta}$ contains the time delays for the sub-array normalised to the first element. The results of the sub-arrays (S) are then beamformed with the delays used for normalisation (Δ). Each stage the delays become larger, because the reference elements are further apart. Also, the number of sub-arrays reduces each stage as the beamforming combines their signals. Therefore, multi-stage beam-

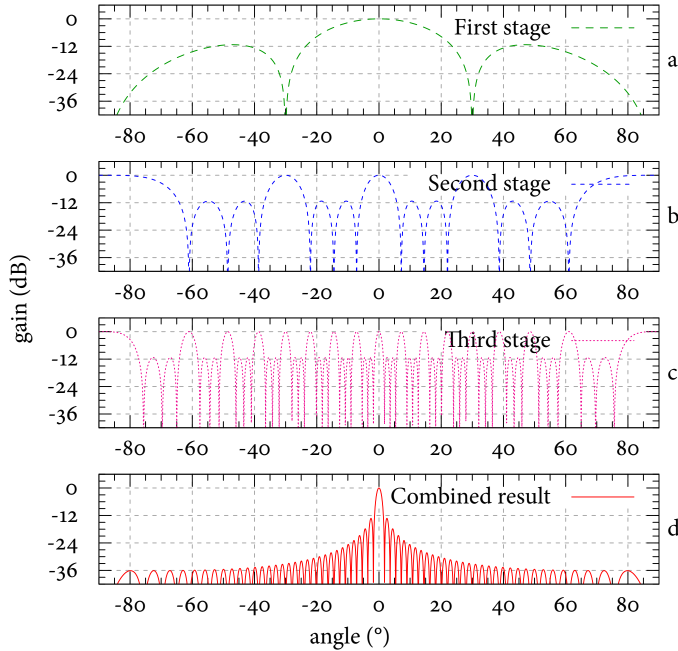


Figure 2.22: Beam pattern of multiple stages

forming has a tree-like structure. Note that phase shift corrections can similarly be distributed over the stages.

As an example, consider a three stage beamformer for a 64 element ULA, i.e. $N = 64$. Each stage combines 4 signals ($4^3 = 64$), i.e. $M = 4$. The beam patterns of the three stages and their combined result are shown in figure 2.22. The beam pattern of the first stage is shown in figure 2.22a. As expected, the beam-width of the first stage is large because the aperture of the 4 elements is small. The beam pattern of the second stage (figure 2.22b) shows a smaller beam-width, as the aperture is increased to a distance of 16 elements, and grating lobes, as only 4 signals are beamformed. Note that these grating lobes are exactly cancelled by the nulls of the first stage. The third stage (figure 2.22c) has the same beam-width of the original 64-element array with 15 grating lobes, of which 12 are "nulled" by the second stage and 3 by the first stage. The final radiation pattern of all stages combined is shown in figure 2.22d.

Because the grating lobes of later stages are "nulled" by previous stages, it is important that the delay corrections are very accurate in the first stages so that the nulls are deep and at the correct position.

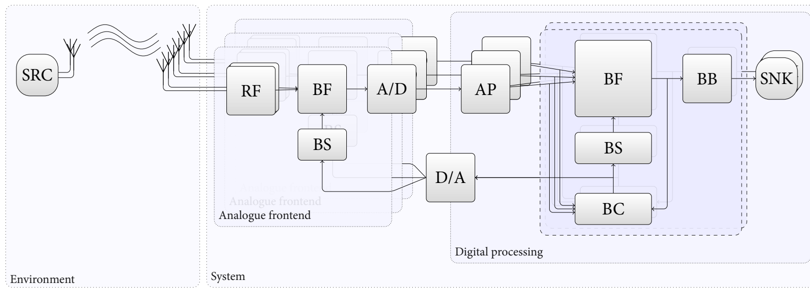


Figure 2.23: Hybrid phased array system

2.3.5 *Hybrid beamforming*

As a consequence of hierarchical beamforming, the first stage(s) can be performed in the analogue domain, while later stages are performed in the digital domain. In that case, the steering vector or steering angle must also be communicated to the analogue beamforming stage(s). For the latter option an analogue beamsteerer is needed. Such a system is shown in figure 2.23. Note that this results in a feedback loop that crosses multiple domains.

Analogue beamforming The advantage of analogue beamforming is that antenna signals are combined before they are digitised, thereby reducing the number of required ADCs and the processing requirements. On the other hand, analogue beamforming is less accurate and less flexible than digital beamforming. When the signals are combined, the signal-of-interest is coherently added, while the noise is incoherently added, so the SNR increases. The analogue signals have a limited SNR, while the digital signals effectively have an unlimitedly SNR (by increasing the word size). Furthermore, analogue beamforming typically allows only a single beam, as each additional beam requires the same amount of additional hardware, i.e. supporting a second beam duplicates the analogue beamforming hardware.

Digital beamforming With digital beamforming, beamforming is performed after the ADCs. Digital beamforming at earlier stages increases the number of required ADCs (although with a lower number of bits per ADC, assuming limited interferers and for the same SNR at the output), but the system is more flexible. For example, such a system can support time delay, phase shift, and FFT based beamforming (see section 2.2.2) as well as different search and track algorithms, using programmable hardware. Again, each additional beam requires the same amount of additional processing (except for FFT beamforming). However, processing capacity can be traded between computing more beams, more sophisticated beamcontrol algorithms, doing nothing to lower energy consumption, or running other applications, for example.

Nonetheless, the increased cost (in terms of e.g. money or power consumption) of an ADC per antenna can be prohibitive. Thus, there is a trade-off between analogue and digital beamforming and with a generic beamforming platform, consisting of a hybrid hierarchical beamformer, we can accommodate all these trade-offs.

Mixed analogue and digital beamforming The time delay between antennas can be very small compared to the sample rate, because the RF is typically much higher than the sample rate at IF. For example, a 5° DoA at 10 GHz with $d = \lambda/2$ results in a time delay of $\frac{sin(5°)}{2 \times 10^9}$ =4.36 ps, while a 200 MS/s ADC has a 5 ns sample period. Therefore, for a digital time delay based beamformer, interpolation is needed to approximate time delays smaller than the sample period. Note that because the DoA can become arbitrary close to 0° the time delays can also become arbitrary small. On the other hand, the maximum delay between the outermost antennas is much larger; for a 90° DoA at 10 GHz and a 256-element ULA, the time delay between the two outer-most antennas is $\frac{sin(90°)}{2 \times 10^9} \cdot 255$ =12.75 ns.

Large delays that are a multiple of the sample rate are easy to implement for a digital beamformer, as they simply require memory elements to buffer samples. However, such larger delays are difficult to implement for an analogue beamformer (for example with delay lines or time-shifted sampling). Hence, the analogue beamformer and the digital beamformer can nicely complement each other in a hierarchical design by implementing small delays in the analogue domain and (the remaining) larger delays in the digital domain.

A digital beamformer can implement both time delay based beamforming and phase shift based beamforming, while an analogue beamformer typically only supports one. For a phase shift based beamformer, the beam direction shifts to larger angles for lower frequencies, i.e. mispointing or beam squint (see section 2.2.4). For a time delay based beamformer mispointing does not occur. Thus, if the first stage is phase shift based and the second is time delay based, the grating lobes of the second stage are not accurately cancelled at lower frequencies (and similarly when the first stage is time delay based and the second phase shift based). If the first stage is time delay based, we can still choose between a time delay solution for wideband signals or a phase shift solution at the second stage. If the first stage is phase shift based, we can no longer (straightforwardly) support wideband signals with a time delay based second stage. One possibility, which we have not further explored, could be to use a frequency dependent null placement at the second stage to match with the frequency dependent grating lobes of the first stage. This is because for the wider beams at the first stages, mispointing is less of an issue (the error is smaller). If the beamwidth of the first stage is wide enough to include the (wideband) signal-of-interest at its lower frequencies, we can use a beam direction that is constant over frequency for the later stages (when the beamwidth narrows) to still support wideband signals.

2.4 Beamcontrol

We have proposed a hybrid hierarchical beamforming platform suitable for multiple applications. All applications require a control algorithm to determine the steering direction, so that we can scan or search for signals-of-interest or track signals (e.g. to search or track satellites, targets, celestial bodies or mobile terminals). In this section we will first provide an overview of three classes of beamcontrol algorithms, so that we can position and evaluate the tracking algorithms of the following sections. Next we will present an algorithm with a low computational complexity which we apply for tracking M-PSK modulated signals. M-PSK modulated signals are used in satellite communications, for example. This algorithm is not suitable for hierarchical systems. Therefore an alternative algorithm is developed that can be used for tracking signals with a hierarchical beamforming system.

2.4.1 *Beamcontrol algorithm classes*

For many applications it is necessary to track signals-of-interest in a mobile environment; the source or receiver or both can be moving (see section 2.3.1). Furthermore, the initial DoA is often not known, thus we must first search for signals-of-interest. Searching and tracking signals is achieved by beamcontrol algorithms.

Beamcontrol algorithms determine the steering direction and optionally the beamshape. The algorithm can use a pre-determined steering direction or determine the direction from the received signals, so-called *adaptive beamcontrol* algorithms. We will focus on the latter as the first is (relatively) straightforward. An adaptive beamcontrol algorithm measures the received signal, analyses it and applies a control feedback signal in the form of a steering vector or steering angle (and beamshape).

There are 3 classes of adaptive beamcontrol algorithms [3].

- *Temporal reference beamforming* algorithms rely on correlation in time between the received signals and a known reference signal. The reference signal is known beforehand and embedded in the signal, such as a training sequence or pilot signal.
- *Spatial reference beamforming* algorithms use correlation in space between the signals received by individual antennas.
- *Blind beamforming* algorithms rely on structural and statistical properties of the received signal.

2.4.1.1 **Temporal reference**

A temporal reference algorithm compares the received signals with expected reference signals and based on that calculates a correction. As such, it can be seen as a form of calibration or equalisation. A temporal reference algorithm is for example used in radar to search for hits, as the sent signal (radar pulse) is known. It is also used in mobile and wireless communication where pilot symbols are used to synchronise in case of multi-path interference.

A temporal reference algorithm calculates a correction weight vector $\vec{w}$ based on the cross-correlation $\vec{\rho}$ between the antenna signals $s_i(t)$ and reference signals $r_i(t)$ (with i enumerating the antennas) and the covariance matrix $\mathbf{\Gamma}$ of the antenna signals [3]:

$$\vec{w} = \mathbf{\Gamma}^{-1}\vec{\rho}$$

Note that because a weight vector is calculated, beamsteering is included in the algorithm. Computing the cross-correlation $\vec{\rho}$ and covariance $\mathbf{\Gamma}$ mainly involves multiply-accumulate (MAC) operations. The computational complexity in MAC operations for cross-correlation is $\mathcal{O}(NK)$, with N the number of antenna elements and K the reference signal length. The covariance matrix and matrix-vector multiplication have an $\mathcal{O}(N^2)$ complexity, but the algorithm is dominated by the matrix inversion with complexity $\mathcal{O}(N^3)$.

2.4.1.2 Spatial reference

A spatial reference algorithm estimates DoAs by comparing the received antenna signals with each other, followed by a selection of the signal-of-interest (the remaining signals being interferers). The major feature of spatial reference algorithms is DoA estimation, which is useful for searching for the initial location of sources, but also for tracking if the DoA estimation is performed continuously. The DoA is estimated by (spatially) correlating the antenna signals with delayed version corresponding to a DoA. This assumes the frequencies of the signals are known and that different sources are uncorrelated.

A basic DoA estimation algorithm is to scan over all angles and perform peak detection. Some more sophisticated algorithms are multiple signal classification (MUSIC), estimation of signal parameters by rotational invariance techniques (ESPRIT) or maximum likelihood (ML) based techniques [74, 84, 88, 120].

Typically, spatial reference algorithms have a high computational complexity. For example, the MUSIC algorithm exploits the eigenstructure of the covariance matrix $\mathbf{\Gamma}$ to calculate the MUSIC spectrum (the power P as a function of the DoA ϑ) [3]:

$$P(\vartheta) = \frac{\vec{w}^H(\vartheta)\vec{w}(\vartheta)}{\vec{w}^H(\vartheta)\boldsymbol{V_n}\boldsymbol{V_n^H}\vec{w}(\vartheta)}$$

with V_n the noise vector, i.e. the vector of the eigenvectors after eigendecomposition of $\vec{\Gamma}$ less the D largest eigenvectors which are considered of the signals-of-interest.

A block diagram of MUSIC is shown in figure 2.24. The covariance matrix calculation correlates signals from all the antenna elements and has complexity $\mathcal{O}(N^2)$. The eigendecomposition is the most complex part and requires an iterative numerical approximation, such as QR-decomposition [31], requiring $\mathcal{O}(N^3)$ MAC operations [85]. The spectrum calculation is shown above and consists of matrix and vector products with pre-calculated steering vectors ($\mathcal{O}(N^2)$). The pre-calculated steering vectors consist of a steering vectors for each DoAs considered, and are

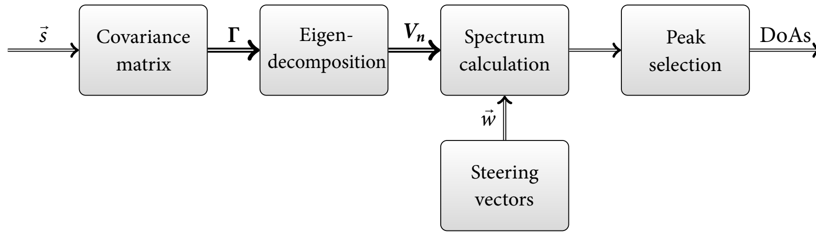


Figure 2.24: MUSIC

expected to be static. Peak selection is a simple search over N angles ($\mathcal{O}(N)$). Thus, complexity is dominated by eigendecomposition [111], and is comparable to the temporal reference algorithms.

2.4.1.3 Blind

Both temporal as well as spatial reference beamforming algorithms require $\mathcal{O}(N^3)$ of MAC operations. For applications with a 100 MS/s sample rate per antenna for (N =) 100 to 1000 antennas, this is not feasible for tracking sources. Of course, the dynamics of the DoAs is likely much lower than the sample rate, but the number of antennas and thus the number of operations remains high. Blind beamforming algorithms provide a way to track signals with much lower computational complexity.

Blind beamforming algorithms, also known as blind deconvolution algorithms, use known signal characteristics of the received signal after beamforming such as a constant modulus or fixed constellation points in the complex plane. Signal changes are translated to a weight vector or steering angle, however, the initial angle of the signal-of-interest must be known. Temporal or spatial reference algorithms can, for example, be used to search for objects, after which they can be tracked with a blind beamforming algorithm.

To find the difference between the properties of the received beamformed signal and the expected properties, a cost function is defined. Properties of the received signal are compared with expected properties using this cost function ($\mathcal{O}(1)$) and the beamformer is steered into the direction of the lowest cost ($\mathcal{O}(1)$), thereby iteratively updating the steering vector ($\mathcal{O}(N)$) (hence including the beamsteerer).

As blind beamforming algorithms rely on structural or statistical signal properties, different algorithms are needed for applications with different signal characteristics (such as the modulation scheme used). The next section presents a blind beamforming algorithm that we have developed for tracking signals with constant modulation points, such as QPSK signals used in DVB-S. Furthermore, in the following section we will modify this algorithm to provide steering angle updates instead of a weight vector.

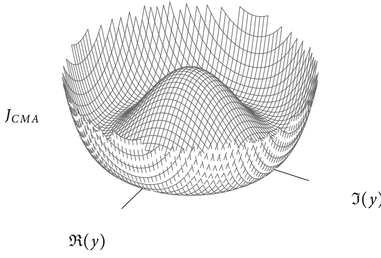


Figure 2.25: Surface plot of the J_{CMA} cost function

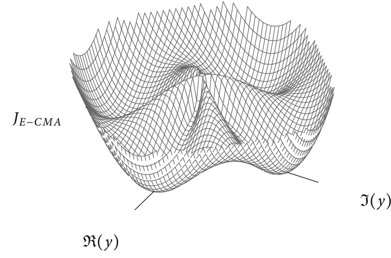


Figure 2.26: Surface plot of the J_{E-CMA} cost function

2.4.2 *Extended CMA*

An algorithm for equalisation of signals with a constant modulus, developed by Godard [34] and independently by Treichler and Agee [100], can be used for blind beamforming [100]. This algorithm is called the constant modulus algorithm (CMA). Xu [118] proposed a phase extension for equalisation of PSK modulated signals, which we call extended CMA (E-CMA). We apply E-CMA to beamforming in order to correct modulus and phase deviations caused by the movement of the source [14]. This is useful for the DVB-S application (see section 2.3.1) which uses QPSK (4-PSK) signals or for example for radar with PSK modulated pulses.

For example, for the DVB-S application, consider a phased array on a moving vehicle. With respect to the array, the source signal from a satellite is continuously moving. Furthermore, as a consumer product, the computational resources required for continuously tracking the source with a spatial reference algorithm are too large. Consequently, we propose to use a spatial reference algorithm only to determine the initial angle, which therefore can take some time, followed by the E-CMA algorithm to track the source in real-time.

2.4.2.1 Constant modulus algorithm

As said, CMA equalises signals with a constant modulus. Therefore, a cost function is defined as the expected deviation of the squared modulus of the signal-of-interest y from the constant modulus R, and which has minimal cost if $|y| = R$:

$$J_{CMA} = E\left\{\left(|y|^2 - R\right)^2\right\}$$

Herein, E represents the expected value. The cost function is illustrated in figure 2.25. As can be seen, the lowest cost is at a circle around the origin of the complex plane which has a constant range or modulus R.

The beamformer output $y = \vec{w}^H \vec{s}$ is the result of the product of antenna signals after antenna processing ($\vec{s}$) with a correction weight vector ($\vec{w}^H$). The aim of the CMA algorithm is to update the weights of the steering vector in such a way that the costs J_{CMA} are minimised. When the source is moving, the beam is slightly

mispointing as the DoA changes, resulting in a gain or modulus decrease and therefore a cost increase. Furthermore, interferers vary the modulus of the received signal. These effects are compensated by the CMA algorithm.

To minimise costs, a stochastic gradient descent technique with respect to $\vec{w}$ is used in CMA for blind beamforming [100]. The gradient follows as:

$$\begin{aligned} \nabla_{\vec{w}} J_{CMA} &= E\left\{2 \cdot \left(|y|^2 - R\right) \cdot \nabla_{\vec{w}} \left(|y|^2 - R\right)\right\} \\ |y|^2 &= yy^* = \vec{w}^H \vec{s}\, \vec{s}^H \vec{w} \end{aligned} \tag{2.2}$$

where $\nabla_{\vec{w}}$ represents the gradient with respect to $\vec{w}$. Using $\nabla_{\vec{w}} \left\{\vec{w}^H \vec{s}\, \vec{s}^H \vec{w}\right\} = 2\vec{s}\, \vec{s}^H \vec{w}$ [66] gives:

$$\nabla_{\vec{w}} J_{CMA} = 4 \cdot E\left\{\left(|y|^2 - R\right) \cdot \vec{s}\, \vec{s}^H \vec{w}\right\} = 4 \cdot E\left\{\left(|y|^2 - R\right) \cdot \vec{s}\, y^*\right\}$$

The steering vector $\vec{w}$ is updated in the direction of the negative gradient to minimise J:

$$\vec{w}\,[t+1] = \vec{w}\,[t] - \mu \nabla_{\vec{w}} J_{CMA}$$

Herein, μ determines the convergence rate of the gradient descent.

This is rewritten as:

$$\vec{w}\,[t+1] = \vec{w}\,[t] - \mu \cdot \left(|y\,[t]|^2 - R\right) \cdot \vec{s}\,[t]\; y\,[t]^*$$

where instantaneous values are used as an approximation of the expected value and μ absorbs the factor 4.

With the resulting weight vector, the phased array is steered by the CMA algorithm, tracking the source. Furthermore, interferers are rejected because they increase the cost, causing the gradient descent algorithm to adjust the beamshape to minimum cost.

2.4.2.2 Phase extension

An M-PSK modulated signal has a constant amplitude and a uniformly distributed phase $\phi = \frac{2\pi m}{M}$, with M the number of constellation points. CMA can be improved by including the phase in the cost function. To include the phase we observe that the phase constraint $\frac{M}{2}\phi = m\pi$ is equivalent to $sin(\frac{M}{2}\phi) = 0$ [118]. The cost function then follows as:

$$J_{E-CMA} = E\left\{\left(|y|^2 - R\right)^2\right\} + E\left\{\left(\sin^2\left(\frac{M}{2} \angle y\right)\right)\right\}$$

with $\angle y$ giving the (polar) angle of y. The cost function is illustrated in figure 2.26 for QPSK, and shows minima that are not only at a constant modulus but also at one of four constant angles or phases, corresponding to the constellation points of QPSK. Thus, minimum costs are reached whenever y simultaneously has a modulus $|y|$

and a phase $\angle y$ equal to one of the M-PSK symbol phases with modulus R. E-CMA thus equalises both the modulus and the phase of the signal-of-interest.

Similar to CMA the cost J_{E-CMA} is iteratively minimised using a stochastic gradient-descent [118], resulting in:

$$\vec{w}[t+1] = \vec{w}[t] - \mu \cdot \varepsilon \cdot \vec{s}[t]$$

where

$$\varepsilon = \frac{8j\left(|y|^4 - |y|^2\right) + M\sin\left(M \angle y\right)}{4j \cdot y}$$

Herein μ again determines the convergence rate, and ε is a scalar correction factor that is multiplied with the antenna signals $\vec{s}$ to compute steering vector updates.

Concerning the computational complexity of E-CMA; N multiplications are required for the multiplication with $\vec{s}$, and N subtractions for updating $\vec{w}$. The remaining operation are scalar operations. E-CMA thus has complexity $\mathcal{O}(N)$.

2.4.2.3 E-CMA for beamforming

A moving phased array experiences both translational as rotational movement with respect to the source.

For a translational movement the antenna elements all experience the same change in path length and thus time delay from the source. The beamformed signal therefore also experiences this time delay. If the array position changes over time (i.e. it moves), the time delay of the beamformed signal also changes over time. For a M-PSK signal, this effect corresponds to a phase shift that varies over time. As such it causes a rotation of the constellation points. Typically, this rotation is corrected with a de-rotator. However, for E-CMA the rotation causes an increase in cost which is automatically corrected by the algorithm, eliminating the need for a de-rotator.

For a rotational movement of the array, the DoA of the source changes (from the perspective of the array). A rotational movement causes a gain or modulus decrease because of mispointing.

For a phase reference at the centre of a ULA the array factor is:

$$G_a(\alpha) = \sum_{i=1}^{N} e^{j\left(2\pi\left(\frac{d\cdot\sin(\vartheta)}{\lambda}\right)\cdot\left(i-\frac{N+1}{2}\right)+\phi_i\right)}$$

With the centre of the array at the origin, translational and rotational movements are orthogonal effects. A translation of the array only causes a rotation of the QPSK constellation points and no modulus change, while a rotation of the array only causes a modulus change and no rotation of the QPSK constellation points.

Thus, movement of a phased array introduces modulus and phase deviations in the M-PSK modulated output of the beamformer due to angular mispointing and a changing path length from the array to the source, respectively. These need to be corrected before the demodulator. E-CMA compensates for those deviations by altering the steering vector weights of the beamformer.

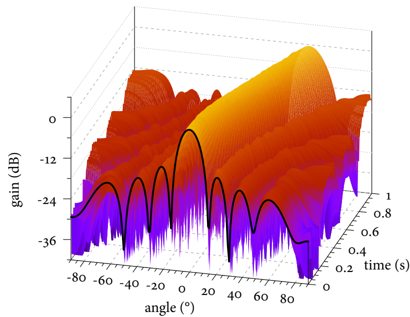


Figure 2.27: E-CMA radiation pattern for the vehicle dynamics scenario

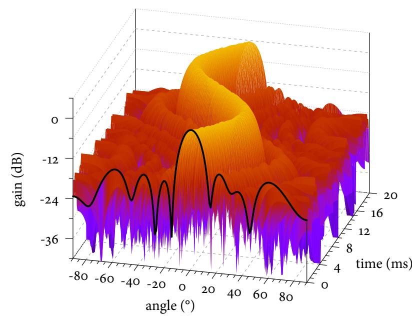


Figure 2.28: E-CMA radiation pattern for the synthetic scenario

2.4.2.4 Results

To verify E-CMA, we have modelled the scenario of a ULA on a moving vehicle for the DVB-S application. A source is QPSK modulated, from which antenna signals are generated. Phase-shift based beamforming is used with E-CMA for beamcontrol and beamsteering. The beamformer output is demodulated and verified against symbol errors.

The translational and rotational movements of the array are based on the vehicle dynamics of a Renault Clio RL 1.1 (the vehicle dynamics are discussed in more detail in [14]) and a synthetic scenario. In both scenarios an 8-element ULA is used, a channel with a SNR of 16 dB, and a convergence rate μ of 0.05.

For the vehicle dynamics, the car's velocity is 72 km/h (20 m/s). In the initial situation the car is driving towards the source ($\alpha = 0°$, i.e. the steering angle is 0°). At $t = 0.1$ second the car is instantaneously steered to 11.5° causing the car to start turning and the DoA of the source to change. Furthermore, the velocity towards the source decreases. The resulting radiation pattern over time is shown in figure 2.27. The main beam is following the scenario as described, it is kept at 0° until 0.1 s, after which the car start to turn and the main beam follows. There is a slight gain increase and some irregularities in the beam pattern due to the E-CMA algorithm, but overall the source is tracked well. The constellation diagram of the output symbols is shown in figure 2.29. The figure shows a clear separation between the constellation points in the four quadrants. We have also compared the output symbols with the input symbols and no symbol errors have occurred for the 40×10^6 symbols simulated from 0 s to 1 s. We therefore conclude that E-CMA is correctly following the trajectory of the car and de-rotating the QPSK symbols.

The car dynamics are relatively slow compared to the antenna data rate of 50 MS/s (complex). For the synthetic scenario a more extreme situation is used; the velocity is 216 km/h (60 m/s) and the phased array moves in such a way that an ideal steering angle would describe a sine wave with a frequency of 50 Hz and an amplitude of 30° azimuth. The resulting radiation pattern over time is shown in

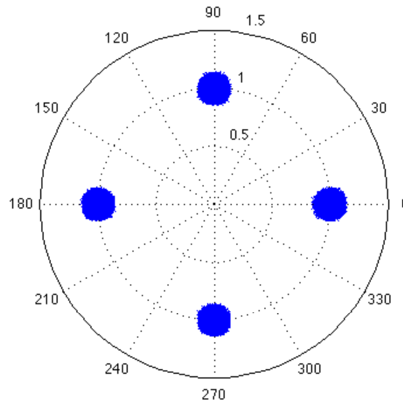


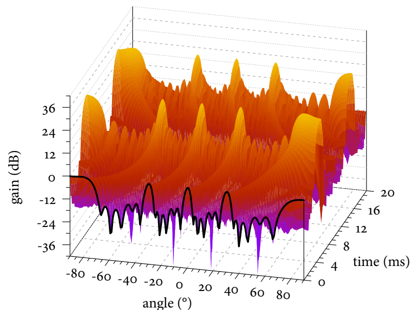
FIGURE 2.29: Constellation diagram of the output symbols

figure 2.28. For this scenario, E-CMA can also successfully track the source without any symbol errors. Note that the radiation pattern contains somewhat distorted sidelobes and has less deep nulls than an ideal pattern. This is because there are no interferers for this scenario, giving the gradient descent algorithm no incentive to increase the null depth to reject those.

The next scenario concerns a hierarchical hybrid array. The array consists of a 4-element analogue beamformer first stage, followed by a 8-element digital beamformer second stage, for a total of 32 antenna elements. In general, CMA and E-CMA use antenna signals $\vec{s}$ to calculate a steering vector $\vec{w}$ which consists of a gain and phase correction for each antenna signal. Assuming the blind beamforming algorithm is implemented in the digital domain, only the digital antenna signals are available. These "digital" antenna signals consist of the beamformed results of the analogue stage. For the radiation pattern of the digital stage in case of a hierarchical beamformer, we expect grating lobes which are cancelled by the radiation pattern of the analogue stage. However, this means the analogue stage must also be steered, while the $\vec{w}$ of E-CMA contains weights for the "digital" antenna signals. Figure 2.30 shows the radiation pattern of the digital stage resulting with E-CMA when the analogue stage is not steered (i.e. has a constant 0° azimuth angle). As we can see, E-CMA can not follow the source when the azimuth angle becomes larger than the beam-width of the analogue stage, causing the gain to increase significantly and causing symbol errors. Figure 2.31 shows the same situation with an ideally steered analogue stage. Now, E-CMA can correctly track the source without symbol errors.

In the next section, we will discuss a version of CMA that calculates a steering angle instead of a weight vector. As such, we can use this steering angle to steer both the analogue stage as well as the digital stage.

E-CMA is defined for M-PSK modulated signals. Further work in this direction could define cost functions for other modulation schemes such as quadrature amplitude modulation (QAM) and derive a gradient descent minimiser for those kind of signals.



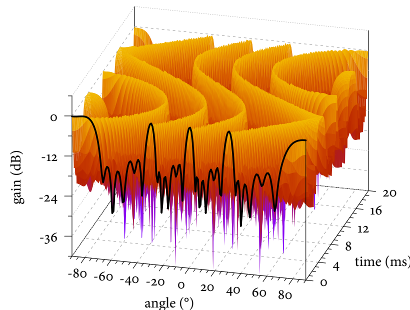


Figure 2.30: E-CMA radiation pattern of the digital stage without steering the analogue stage

Figure 2.31: E-CMA radiation pattern of the digital stage with ideal steering of the analogue stage

2.4.3 *Angular CMA*

Angular CMA (A-CMA) is an adaptation of CMA to provide a steering *angle* instead of a weight vector. As we found from the previous section, CMA and E-CMA take a vector of antenna signals as input to calculate a correction for each antenna signal. However, not all antenna signals are available in digitised form in a hybrid hierarchical beamforming system (see section 2.3.4). Therefore, only corrections are provided for the input signal of the digital stage after analogue beamforming is already performed. With a steering angle, a separate beamsteerer can be used to calculate correction parameters for both the analogue as well as the digital beamforming stage. More importantly, the steering angle is used to calculate just the phase taper. This means the amplitude taper can still be defined independently, unlike the weight vector from CMA. CMA also has the characteristic that sometimes just the gain is increased to achieve the expected modulus instead of steering the beam in the direction of the source, especially when there are no interferers. This is problematic because it reduces the SNR while beamsteering with a phase taper does not.

The derivation of A-CMA was performed by Blom in [KCR:7]. In this thesis we will therefore only provide the major steps in the derivation, needed to understand the general idea. Furthermore, we will apply A-CMA for a hybrid hierarchical beamformer as an additional contribution.

2.4.3.1 Derivation

The derivation of A-CMA is based on the derivation of CMA: first a cost function is defined, then the gradient of the cost function is determined and finally the gradient is used with the gradient descent algorithm.

To adapt CMA to calculate a steering angle, the cost function is made dependent on the angle by using a phase taper as weight vector. Therefore we need the array

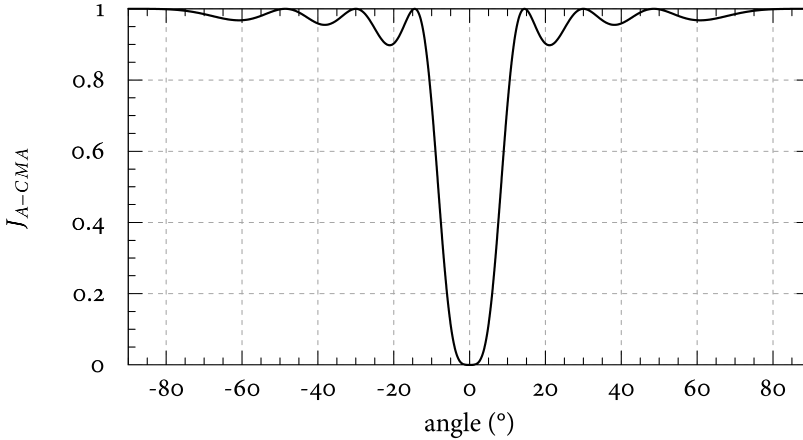


Figure 2.32: Surface plot of the J_{A-CMA} cost function

structure (i.e. the antenna positions) as the phase taper is dependent on it. In this work, a ULA is assumed with a LPT. The resulting weight vector follows as:

$$\vec{w}(\theta) = e^{\phi(\theta)\cdot\vec{n}} = e^{j\frac{2\pi\cdot d\sin(\theta)}{\lambda}\cdot\vec{n}}, \quad \vec{n} = [0\ldots(N-1)]^T$$

Next we will define the cost function and apply a gradient descent to derive A-CMA.

Cost function Using the cost function of CMA with the above weight vector we find:

$$\begin{aligned} J_{A-CMA}(\theta) &= E\left\{\left(\left|\vec{w}(\theta)^H\vec{s}\right|^2 - R\right)^2\right\} \\ &= E\left\{\left(\left|\left(e^{j\frac{2\pi\cdot d\sin(\theta)}{\lambda}\cdot\vec{n}}\right)^H\vec{s}\right|^2 - R\right)^2\right\} \end{aligned}$$

This cost function is shown in figure 2.32 for $d = \lambda/2$ and a 0° DoA. As can be seen, the cost function has a global minimum at 0° DoA, but also has local minima. Therefore, the steering angle must stay within the convergence region for the gradient descent to converge to the global minimum, i.e. the angle must stay between the maxima surrounding the global minimum.

Because of the $\vec{w}(\theta)^H\vec{x}$ term, the cost function is comparable to an (inverted) radiation pattern. The convergence region is thus likewise determined by the physical distance between the two outer-antennas of an array, i.e. it is comparable to the beamwidth. The beamwidth of a ULA, and thus the convergence region for A-CMA with an LPT, is given by the INBW [3]:

$$INBW_{\theta_0=0} = 2\arcsin\left(\frac{\lambda}{dN}\right)$$

Gradient descent The gradient descent algorithm uses the derivative with respect to θ of the cost function (using equation (2.2)):

$$\nabla_\theta J_{A-CMA} = E\left\{2 \cdot (|y|^2 - R) \cdot \nabla_\theta \left(\left(e^{\phi(\theta)\cdot\vec{n}}\right)^H \vec{s}\,\vec{s}^H e^{\phi(\theta)\cdot\vec{n}} - R\right)\right\}$$

Reordering terms and using $B(\theta) = e^{\phi(\theta)\cdot\vec{n}}(e^{\phi(\theta)\cdot\vec{n}})^H$ gives:

$$\nabla_\theta J_{A-CMA} = 2 \cdot E\left\{\left(|y|^2 - R\right) \cdot \left(\vec{s}^H B'(\theta)\vec{s}\right)\right\}$$

where $B'(\theta) = \frac{\partial}{\partial\theta}B(\theta)$ can be written as:

$$B'(\theta) = \begin{bmatrix} 0 & \cdots & (n_0 - n_{N-1})\phi'(\theta)e^{(n_0-n_{N-1})\phi(\theta)} \\ \vdots & 0 & \vdots \\ (n_{N-1}-n_0)\phi'(\theta)e^{(n_{N-1}-n_0)\phi(\theta)} & \cdots & 0 \end{bmatrix}$$

$$\phi'(\theta) = j\frac{2\pi \cdot d\cos(\theta)}{\lambda}$$

As with CMA and E-CMA the angle θ is updated in the direction of the negative gradient descent using instantaneous values as an approximation of the expected value:

$$\theta[t+1] = \theta[t] - \mu\left(|y[t]|^2 - R\right) \cdot \left(\vec{s[t]}^H B'(\theta[t])\,\vec{s[t]}\right)$$

where μ absorbs the factor 2. More details can be found in [KCR:7].

The computational complexity of A-CMA is then dominated by the computation of and matrix-vector multiplication with $N \times N$ matrix $B'(\theta)$ resulting in a $\mathcal{O}(N^2)$ complexity, i.e. more than E-CMA but far less than temporal and spatial beamforming algorithms.

2.4.3.2 Results

We repeat the hierarchical hybrid array scenario of the previous section, but instead of using an ideal steering vector for the analogue part, we use a LPT based on the steering angle determined by A-CMA. The radiation pattern of the analogue beamformer is shown in figure 2.33 and the radiation pattern of the digital beamformer is shown in figure 2.34. The analogue stage consists of 4 element beamformers and digital stage is an 8 element beamformer. The steering angle is used for both the analogue and digital stage, resulting in radiation patterns that closely follow the DoA of the source.

A-CMA operates on only the digital antenna data. The antenna positions of the digital "virtual" antennas have a distance d which is larger than $\lambda/2$. Therefore the cost function has multiple global minima, comparable to grating lobes. In this case $d = 2\lambda$ and $N = 8$ resulting in a convergence region given by the INBW of:

$$\text{INBW}_{\theta_0=0} = 2\arcsin\left(\frac{\lambda}{dN}\right) = 2\arcsin\left(\frac{1}{2\cdot 8}\right) \approx 7^\circ$$

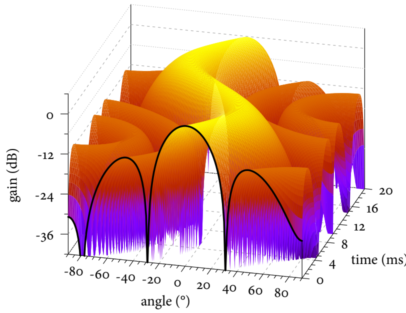


Figure 2.33: A-CMA radiation pattern of the analogue stage

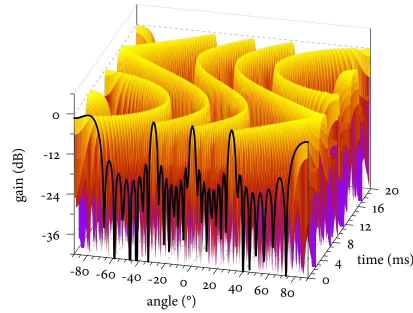


Figure 2.34: A-CMA radiation pattern of the digital stage

A LPT is used for derivation by A-CMA, requiring the antenna elements to be at a fixed distance. A LPT is also used for beamsteering and no gain taper is used. Therefore, the nulls are well-defined but at fixed position. By using a gain taper, the beam-shape and null positions can be defined. It is therefore interesting to further explore A-CMA based blind beamforming in this direction.

Note that A-CMA does not correct the rotation of the constellation points as E-CMA does, because a steering angle is calculated based on the CMA cost function. Therefore, the steering angle is only adjusted on the basis of the modulus of $\vec{w}(\theta)^H\vec{s}$ and not the phase. This steering angle is used to calculate a LPT; de-rotation could possibly be included as a constant phase offset for the LPT.

2.5 Conclusion

In this chapter we have presented the application domain of phased array beamforming applications. A phased array consists of an array of antennas, the signals of which are continuously being processed to perform spatial filtering. As such, a beamforming application consists of a lot of signal processing on streaming data. As a larger high-performance streaming signal processing application that is resource constrained, beamforming applications form a good case study for the design of future demanding embedded systems.

A generic beamforming platform is proposed to enable beamforming for consumer applications. This is achieved by lowering the cost of the platform by sharing development cost and economies of scale. The generic beamforming platform consists of an analogue front-end including an ADC for each antenna, and digital processing consisting of antenna processing (filtering) for each antenna, followed by beamforming and baseband processing. The beamformer applies a time delay or phase shift to each antenna signal before summing the signals. The time delays or phase shifts for a certain steering direction are computed by the beamsteerer, while the steering direction is determined by the beamcontrol processing. The

beamcontroller introduces adaptivity and feedback, complicating the design. Furthermore, the verification of applications on the platform requires inclusion of the environment when modelling and simulating the platform.

An analysis of beamforming applications shows large differences in array size and therefore processing requirements. As a consequence, a generic platform that supports all applications must be modular and scalable to be cost-effective. In addition, the platform must be flexible enough to support multiple applications and multiple beamforming methods. This also involves partitioning the beamformer into multiple stages to support modular and scalable processing, so-called hierarchical beamforming. Furthermore, hybrid beamforming is proposed to further reduce cost by lowering the number of required analogue front-ends and antenna processing using analogue beamformers for the first stage.

With a phased array system, signals-of-interest can be searched or tracked in a dynamic scenario, e.g. a satellite is tracked with a phased array on a moving vehicle, or searching for targets with a radar system. Searching or tracking is performed by the beamcontrol algorithm. Analysis of beamcontrol algorithms show that search algorithms are computationally complex. A tracking algorithm based on structural or statistical properties is less complex, but requires such signals from the application. The generic platform should be flexible enough to support switching between a search algorithm for finding the initial DoA of a signal-of-interest and tracking and beamforming after that for normal operation.

E-CMA is a tracking algorithm we have developed for M-PSK modulated signals, such as used for DVB-S signals in satellite reception. It is an adaptive algorithm that updates the steering vector of a phase shift based beamformer such that the cost of a cost function is minimised. The cost function has lowest cost if the constellation points of the PSK modulated signal have a constant modulus and phase. Modulus changes are caused by rotational movement of the array, while phase changes are caused by translational movement. As such, both movements are corrected by the E-CMA algorithm. E-CMA has low complexity that increases linearly with the number of antennas. The algorithm was verified with a scenario of a phased array on a vehicle that is moving towards the source while turning, and a more extreme synthetic scenario where the DoA changes over 60°. In both cases E-CMA is able to successfully track the QPSK modulated source without causing symbol errors.

However, E-CMA is not suitable for a hybrid beamforming system, because it only computes a steering vector for the digital beamforming stage, which we can not use for steering the analogue stage. Therefore, the A-CMA tracking algorithm is developed. A-CMA is an adaptive algorithm that iteratively updates a steering angle instead of a steering vector. This is achieved by defining a cost function based on a LPT in the direction of the steering angle. This steering angle is used to steer both the analogue stage and the digital stage of the hybrid beamformer. A-CMA is more complex than E-CMA with a quadratic dependence on the number of antennas, but it is less complex than search algorithms. A-CMA is verified using the above synthetic scenario and a hybrid hierarchical array with 4-element analogue beamformers and an 8-element digital beamformer, and is successfully able to track the source.

Chapter 3

Tiled reconfigurable architectures for beamforming

Abstract – *The main requirements from the application domain for a generic beamforming platform are (energy) efficiency, scalability and flexibility. In this chapter we explore tiled reconfigurable architectures for beamforming applications. Scalability is achieved by using a tiled architecture, and efficiency and flexibility are achieved using a reconfigurable architecture. The consequences of mapping a large application such as beamforming onto an embedded system consisting of a (low-cost) tiled reconfigurable architecture will be analysed. Therefore, we show the results of three example implementations; an audio beamformer on a single reconfigurable tile, a small tiled reconfigurable architecture for a DVB-S beamformer, and a large conceptual tiled architecture for radio astronomy.*

A tiled reconfigurable architecture could provide an efficient, scalable and flexible platform for embedded systems. In this chapter we will explore tiled reconfigurable architectures to investigate whether they are suitable for the application domain described in chapter 2. There we proposed a generic phased array beamforming platform and we concluded that such a platform must be scalable and flexible to support multiple beamforming applications.

Phased array beamforming techniques have been applied in radar and radio astronomy systems for many years already. The design of these systems is mainly driven by functional requirements (e.g., resolution, sensitivity, response time) where non-functional requirements (e.g., costs, power consumption) are of secondary concern [109]. For that reason, no low-cost, low-power systems for more than a few antennas are available yet. Conventional phased array systems also typically

Parts of this chapter have been published in [KCR:6], [KCR:11] and [KCR:12].

use a large amount of dedicated central processing hardware, making the system neither scalable nor flexible [90].

In consumer applications such as wireless and mobile communications and satellite receivers, phased array antennas show great promise but their large scale introduction has been obstructed by the high costs involved. A generic low-cost beamforming platform could enable phased array beamforming for consumer applications. This can be realised by using a scalable architecture that is flexible enough to support multiple applications, such that the same architecture can be reused for more applications. A tiled and reconfigurable architecture seems to be a promising candidate for such an architecture.

Tiled architectures have not been widely used for phased array beamforming applications yet. The usage of tiled platforms is until now limited to small and medium size applications, and it is not clear whether they are usable for large scale applications as well. We will discuss various aspects of tiled architectures such as flexibility and scalability, and we present example implementations for beamforming on three different architectures: the Montium, the LEON SoC, and a concept tiled architecture. We will find that a tiled reconfigurable architecture indeed is suitable for large scale applications, provided such an application can be partitioned into components which each fit on one tile. To be able to do so communication has to be made explicit. We use the dataflow model to express both the partitioning of the application and the communication between components.

Note that typically multiple applications run simultaneously on a tiled architecture [39]. As such, applications are distributed over the tiles and care is taken that adding an application does not interfere with the applications that are already running, i.e. composability with respect to applications. For the beamforming applications, a single application runs on the tiled architecture and tasks of the application are distributed. Resource management such as mapping the application on the platform is less critical and less dynamic than with multiple applications and we will perform a manual mapping of the beamforming application on a tiled architecture.

Concerning reconfigurability we will discuss the possibilities to run various scenarios of the same application on the same platform. In particular, regarding the beamforming application we observed that the DoA estimation algorithm is too expensive to run continuously, so that it is necessary to reconfigure between this algorithm and less expensive tracking algorithms. As such, with reconfiguration of a tiled architecture, it is well possible to switch between an expensive search algorithm and a cheaper tracking algorithm.

3.1 Requirements from the application domain

In this section we will discuss the requirements from the application domain that are relevant to evaluate tiled reconfigurable architectures.

Table 3.1: Complexity per basic operation per sample

	Operation	Complexity
Antenna processing	Filter	$\mathcal{O}(NF)$
Beamforming	Time delay / Phase shift	$\mathcal{O}(NFB)$ / $\mathcal{O}(NB)$
Beamsteering	Compute correction vector	$\mathcal{O}(NB)$
Beamcontrol	E-CMA	$\mathcal{O}(NB)$
	A-CMA	$\mathcal{O}(N^2B)$

N: the number of antennas; F: the number of filter taps; B: the number of beams to be formed.

3.1.1 *Distributed processing*

Typical operations that occur in streaming signal processing applications, such as beamforming, are first of all multiply-accumulates (MACs), often occurring in matrix and vector multiplications, or in FFTs. In addition, especially in control algorithms, such as beamcontrol algorithms, operations such as division and trigonometric functions are required, but these are much less frequent.

The number of times that these MAC operations have to be executed is huge, due to the very high data rates. For example, for radar applications a 200 MS/s sample rate with 16 bit samples (≈ 100 dB) is used. For 1024 antennas, FFT based beamforming would need $200 \cdot 10^6 \cdot 5120 \cdot 10 \approx 10$ T operations per second (ops). For satellite applications with 16×16 antennas, three beams and 100 MS/s sample rate, we would need $100 \cdot 10^6 \cdot 8 \cdot 256 \cdot 3 \approx 400$ G ops with 4 bit samples (≈ 20 dB).

The most demanding application is probably radio astronomy. LOFAR, for example, uses polyphase filter banks with 16-tap FIR filters and a 1024-point FFT for each of the 96 stations, followed by distributed beamforming. Next a polyphase filter bank with 256 16-tap FIR filters and a 256-point FFT is used for each beam, followed by a centrally processed correlation, all together totalling 160 T ops [21] (also see section 3.3.3).

On the other hand, the complexity of the major operations of a beamforming system is not big. From table 3.1 it follows that, e.g. the complexity of the beamsteering operation is linear in the number of antennas and in the number of beams. Only for A-CMA beamcontrol, the complexity is quadratic in the number of antennas. Since the number of beams usually is small, the main factor in the complexity is the number of antennas.

However, even though the complexity of the individual components is not that large, the tight time constraints caused by the high data rates, makes that these operations can not be run on a single processor of limited size. Moreover, the power consumption of such a processor would be enormous. Since the goal of our work is to investigate possible architectures which are available for the consumer market, it is clear that the application has to be partitioned so that the separate parts can be executed on small and efficient processors.

The beamcontrol operation is not executed very often and the time constraints are not very tight. Therefore we choose not to partition this operation (see also

chapter 6). The operations that occur most often are beamforming and filtering. Partitioning of beamforming is described in chapter 2, and partitioning of filtering is straightforward. Clearly, this partitioning introduces communication; we will come back to this in section 3.3. Nevertheless, the application is suitable to distribute over different processing components, thus making tiled architectures an appropriate candidate for beamforming and comparable applications.

3.1.2 Communication infrastructure

An important aspect of the architecture is the communication between tiles. The tiles are connected by means of a network-on-chip (NoC). This NoC concept is also extended to higher hierarchical levels, i.e. to connections between ICs and boards.

Beamforming is an application with a relatively large amount of communication per computation, putting high demands on the NoC. The antennas signals have a sample rate of up to 200 MS/s, which requires at least 400 MB/s network connections. To support some flexibility, such as monitoring of data streams or injection of test or configuration data as well as some control signals, at least two connections per link are needed. The routing of the data-streams changes seldom for the phased array application, therefore a circuit-switched network is more efficient avoiding the overhead of a packet-switched network [116].

3.1.3 Flexibility

As already observed in chapter 2, a more flexible system is useful for a wider range of applications reducing cost because of the larger production volume. It would be beneficial if the same platform can also be used for high-volume consumer applications. Secondly, flexibility is needed to face the frequently changing standards, so that the platform must be adapted to new standards several times during its life-time.

In the context of beamforming applications it is also advantageous that the platform is flexible because of the various methods for beamforming. Time delay based beamforming is suitable for wide-band signals which is, for example, useful in the case of radar to achieve a higher range resolution for the distance to an object. On the other hand, time delay based beamforming is computationally more intensive than phase shift based beamforming. The consequence is that phase shift beamforming can compute more beams at the same time. However, with phase shift beamforming, the bandwidth of the signals must narrower. In addition to these methods there is also FFT based beamforming, with which many beams can be computed at the same time, however, they can not be steered independently. That means that the preferable method for beamforming depends on the situation and it is desirable to be able to switch from one method to another.

The above mentioned switch from one method to another is relatively local. On a larger scale we also want to be able to switch from searching to tracking. Searching is necessary to find the object of interest, e.g. a satellite for broadcasting. However, as we found in chapter 2, searching is computationally very expensive whereas

tracking an already found object is much cheaper. In practice, one might want to change to another satellite, or it is well possible that a satellite position is lost. In both cases one has to switch from tracking to searching and back.

3.2 Architecture

The need to distribute the beamforming application over simple processors, as well as the need for efficiency and flexibility leads to an architecture which consists of several processors connected by a flexible communication infrastructure. We choose for a so-called tiled architecture where multiple functional elements are combined on a SoC and communication proceeds via a NoC. In addition, we choose for a reconfigurable architecture to keep the execution of the various tasks efficient but flexible. Below we will motivate our choices in detail.

There are many aspects of tiled architectures that we do not discuss, such as predictability, composability, types of communication, since these aspects fall outside the scope of this thesis. We would like to refer to [39, 104] for a discussion on these issues.

3.2.1 *Tiled architectures*

Below we briefly discuss five aspects of tiled architectures that are of importance for this thesis.

Scalability First of all the architecture has to be scalable such that it is easily extendable with additional tiles if, for example, the platform has to be extended for a bigger number of antennas. For example, for the beamforming applications there is a wide range in the number of antenna signals to process and the required processing capacity (from 256 antenna inputs and 400 G ops for the DVB-S application to 14 784 inputs and 160 T ops for radio astronomy). Also in case the number of beams that have to be computed increases, the platform may have to be extended with extra processing cores. The communication network also scales with adding processing cores, whereas when cores are connected by a bus, scalability is not guaranteed. Hence, for reasons of scalability a tiled architecture is in our case advisable, the more so because the needed processing capacity is huge. It is so large that scalability is needed on multiple (hierarchical) levels, as multiple chips and even multiple boards are needed: a multiprocessor system-on-chip (MPSoC) is extended to multiple chips on a board (MCoB) and multiple boards in a system (MBiS).

Dependability A tiled architecture is dependable since in case of broken tiles the network can be reconfigured such that computation is relocated on different tiles and communication is rerouted. Thus a tiled system allows for graceful degradation. In case a tile is already broken during the production process of the tiled architecture, the broken tile may simply be disabled.

Heterogeneity Another advantage of a tiled architecture is that the tiles can be different in nature, i.e. the architecture can be heterogeneous. This is especially important in our case of a hybrid system which requires a high performance, such that tiles which are dedicated for specific tasks will be necessary. For example, some tiles will be analogue front-ends, while others will be processing elements with dedicated functionality or more general purpose processing elements.

Distributivity On a tiled architecture, the processing is distributed over multiple cores, thereby introducing communication overhead (as communication does not directly contribute to calculating the result of the application) and sacrificing programmability (as data communications have to be taken into account).

Efficiency Smaller cores, such as used on a tiled architecture, are typically more efficient as they are simpler, thereby allowing the hardware to be optimised and faster. Furthermore, using smaller cores with small local memories is more energy efficient by exploiting locality of reference [23]. Finally, to be cost-effective, it is useful to use consumer market components, such a generic tiled architecture, instead of dedicated hardware.

3.2.2 Reconfigurable architectures

The above discussed requirement of flexibility motivates that the architecture has to be reconfigurable. Reconfigurability exploits the property that for many embedded systems the functionality is fixed on a time scale much larger than the processing of individual data elements. The functionality is captured in a *configuration*; control signals remain fixed for a single configuration while data signals are processed, thus offering the possibility of efficient execution. After some time the system can be reconfigured to change (parts of the) functionality [43].

For reconfiguring a single tile, we need reconfigurable processing elements. The main characteristic is that the operation performed on the data is changed. For reconfiguring a tiled architecture, we also need reconfigurable communication. On this higher level, the main characteristic is that the flow of data is changed.

In our case, reconfiguration can be applied on multiple levels. On the smallest scale (with respect to impact as well as passed time) only parameters used for processing, such as filter coefficients, are changed. This has no impact on the operations performed. On a medium scale, the functionality of a single tile can be changed. In this case the operations performed on the tile are changed, but the data streams of the architecture are not causing further impact on the rest of the system. For large scale reconfiguration also the data streams between the tiles change, i.e. when re-mapping or changing the application. For example, for the beamforming application, a small scale reconfiguration can consist of new beamsteering parameters. A medium scale reconfiguration can be a different mapping of the application or changing the beamforming or tracking method (e.g. due to the weather or mobility). A large scale reconfiguration could consist of changing the beamcontrol algorithm, using sub-arrays or multi-function radar.

3.2.3 The programming challenge

Many-core (tiled) architectures are not easily programmable with traditional programming techniques [5]. Distributing or parallelising an application over multiple cores is therefore mainly a manual process (also see section 4.4.6). Thus, there are strong requirements put on the design process. In chapter 5 we will present a method to ease the design and use of tiled architectures and to improve the automatisation of parallelising applications with an approach based on a mathematical specification of an application.

3.3 Experiments with tiled reconfigurable architectures

In this section we discuss three examples of beamforming on tiled reconfigurable architectures. The first example is concerned with beamforming in an audio context, thus the data rate is rather low. This example still fits on one processor, though reconfigurability is needed because of the necessity to switch between the various beamforming methods (see section 3.1.3). Also the need to switch between searching and tracking motivates that we need a reconfigurable processor. In the first example we therefore choose for the Montium, a coarse-grained reconfigurable processor [43] which is optimised for streaming signal processing operations. The Montium is a very long instruction word (VLIW)-like processor with 5 arithmetic logic units (ALUs), 10 local memories and an interconnect between them. Several core operations, called kernels, for signal processing applications have been implemented on the Montium [44, 79] and [KCR:1]. For more details see appendix B.

In the second example we discuss a phase shift based beamformer with E-CMA based beamcontrol for satellite reception of DVB-S signals. Due to the high data rate of DVB-S, a single processor is not sufficient anymore. The only tiled platform with several reconfigurable processors that was available as a hardware prototype at that time[1], is the LEON SoC, a platform with three Montium processors, a LEON2 processor [2] and a NoC. Still, in order to fit on the LEON SoC, the data rate had to be reduced.

The third example deals with LOFAR, a large scale phased array for beamforming in radio astronomy. At present no tiled architecture big enough to handle this application is available, hence we discuss a conceptual tiled architecture for which we assume a processing capacity of 200 M ops per tile and 64 tiles per SoC. It turns out that for mapping the LOFAR system on such a platform we need 101 SoCs for each of the 77 LOFAR stations.

3.3.1 Audio beamforming on a single reconfigurable processor

This experiment is split in two parts: beamforming methods on the Montium, and beamcontrol on the Montium2.

[1] At the time of performing this research; recently also the CRISP platform, consisting of 9 reconfigurable processors, became available (see [20, 98])

Beamforming methods We have implemented time delay (TD), phase shift (PS) and FFT based beamforming on a single Montium [76]. Here, we exploited the reconfigurability of the Montium to switch between these three methods. For this experiment an audio beamformer was used consisting of an 8-antenna ULA of microphones. The microphone signals are sampled by ADCs which are connected to a Xilinx Virtex-II Pro Development System field-programmable gate array (FPGA) containing a Montium and a PowerPC processor, where the Montium is used for executing the beamforming methods and the PowerPC is used for control to switch between these methods. We remark that the TD implementation is based on a simple first order linear interpolation, whereas the PS and FFT implementations require a 16-tap Hilbert transform filter and are based on a complex multiplication. Further details of the implementation fall outside the scope of this text and can be found in [76].

For reconfiguring the Montium, its five ALUs each have a configuration memory which can contain a few configurations. Each of those fixes the ALU to a specific combination of operations which are repeatedly performed on a stream of data. In addition, there is a configuration memory for the communication between the ALUs such that, for example, they can be pipelined or used in parallel. Switching between the various beamforming methods now amounts to choosing the right configuration setting from the configuration memories.

Realised on the above mentioned FPGA the Montium runs at 20 MHz. The audio signal is sampled at 40 kHz, so there are 500 cycles available per sample whereas less than 30 cycles are needed (10 for TD, and 29 for both PS and FFT), such that many beams can be formed at the same time. Since the necessary configurations are already available in memory, reconfiguration is instantaneous. These results show the feasibility of the Montium for executing the beamforming methods and reconfiguring between them, i.e. a light beamforming application can still be run on a single reconfigurable processor and no tiled architecture is needed. Previous research has shown that the Montium is far more energy efficient than an ARM processor or an FPGA [43] and [KCR:1].

On the other hand, the implementation on the Montium takes a lot of effort, because of the large number of functional units in combination with the multi-layer structure of the configurations causes that the designer is confronted with a large number of low level details for which, in addition, there are no abstraction mechanisms available. Thus programming the Montium is a challenging task. This is in accordance with earlier experience [78].

Beamcontrol The beamforming operation is straightforward; in order to evaluate a more complex algorithm on a reconfigurable processor, such as a beamcontrol algorithm, we have implemented the MUSIC algorithm [111]. MUSIC is a spatial reference algorithm that determines the DoA of signals-of-interest by eigen-decomposition of the covariance matrix of the input signals (for more details see section 2.4.1). The algorithm is implemented on the Montium2, an experimental architecture based on the Montium. It has a comparable amount of functional units, but aims at higher clock frequencies by connecting more of these units di-

Table 3.2: Number of required resources per kernel of MUSIC

Kernel	Clock cycles	Percentage	Time (ms)
Covariance matrix calculation	283 356	18.92	1.41
Eigendecomposition	1 162 128	77.58	5.81
Spectrum calculation	52 048	3.47	0.26
Peak selection	415	0.03	0.00
Total	1 497 947	100	7.49

rectly to the interconnect. However, we expect that the differences between implementations on the Montium and the Montium2 are negligible.

The different kernels of the MUSIC algorithm (see table 3.2) were implemented and run sequentially on a single Montium2. The eigendecomposition is numerically approximated with an implementation of the QR algorithm and is by far the most complex. The results for an antenna array of 16 elements and the Montium2 running at 200 MHz are shown in table 3.2. The conclusion from this table is that a single Montium2 is sufficient to compute the DoA. However, since only 16 antennas were used, the angular resolution is too low for the applications considered (requiring at least 256 antennas). Since the complexity of the MUSIC algorithm is N^3, with N for the number of antennas, the limits of the Montium2 will be reached quickly. Hence, a single Montium or Montium2 will not be enough for beamcontrol when the angular resolution requires a larger number of antennas.

3.3.2 *A tiled reconfigurable architecture for a DVB-S beamformer*

We have implemented an adaptive beamformer on the LEON SoC platform [95]. It is based on the DVB-S application and includes an 8-element beamformer and E-CMA based adaptive beamcontrol for a single beam. Furthermore, a 25-taps FIR filter was included for baseband processing of the QPSK modulated signals.

The LEON SoC platform consists of a LEON2 processor [2] and three Montiums as shown in figure 3.1. The LEON2 is used for control tasks such as resource management and interfacing. The advanced high-performance bus (AHB)-bus connects the LEON2 to memory and to the serial and USB interfaces. The USB interface is also directly connected to the NoC to support streaming data over USB. In addition, the AHB-bridge connects the LEON2 and the peripherals to the NoC.

For the communication between the Montium processors, a predictable circuit-switched NoC is used [117]. Each network link between a router and another router or processor consists of four parallel 16 bit lanes that can be used simultaneously such that a high communication bandwidth between the tiles is achieved. The NoC is configured via a dedicated configuration interface which is accessed through the memory map of the AHB bridge.

The LEON SoC is realised on a Xilinx Virtex-4 LX200 FPGA. For this prototype realisation, the LEON2 processor and the NoC operate at 47.52 MHz, resulting in a bandwidth of about 380 MB/s for each link. The three Montium proces-

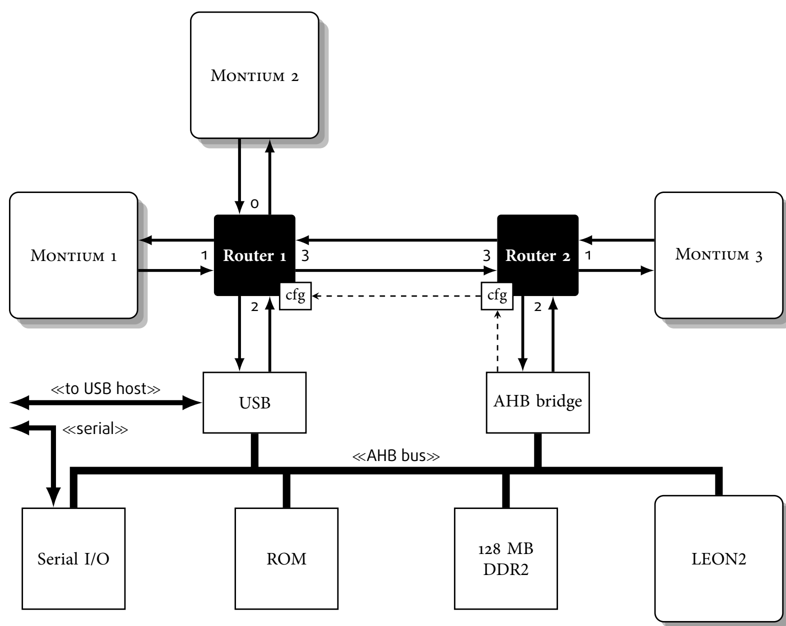


Figure 3.1: LEON SoC

sors operate at 15.84 MHz. The Montiums are rarely stalled waiting for data, since the NoC operates at three times the frequency of a Montium. Besides, a Montium can compute in parallel with communication over the network.

An application specific integrated circuit (ASIC) realisation of a similar architecture, consisting of an ARM-926 processor and four Montiums, exists in 130 nm technology [91, 92].

The beamforming and baseband processing is running on two of the Montiums and E-CMA is running on the third. As the DoA is changing at a much lower frequency than the sample rate, the beamcontrol algorithm is run only every 12 samples. The third Montium is therefore idle most of the time. The LEON2 configures the NoC and Montiums during the initialisation phase, and it reconfigures the NoC during operation.

The experimental realisation on the above mentioned FPGA can process about 1.5 MS/s, approximately one thirtieth of the 50 MS/s which are actually required for the DVB-S application. Besides, only eight antennas were assumed, where 256 would be more realistic. Hence, this platform is insufficient for the implementation of beamforming for DVB-S. Even an ASIC realisation of the LEON SoC platform will allow the Montiums to run at a frequency that is approximately only ten times higher, so the conclusion is that for a DVB-S beamformer more than three Montiums are required.

More details about the implementation can be found in chapter 6 and [11, 95].

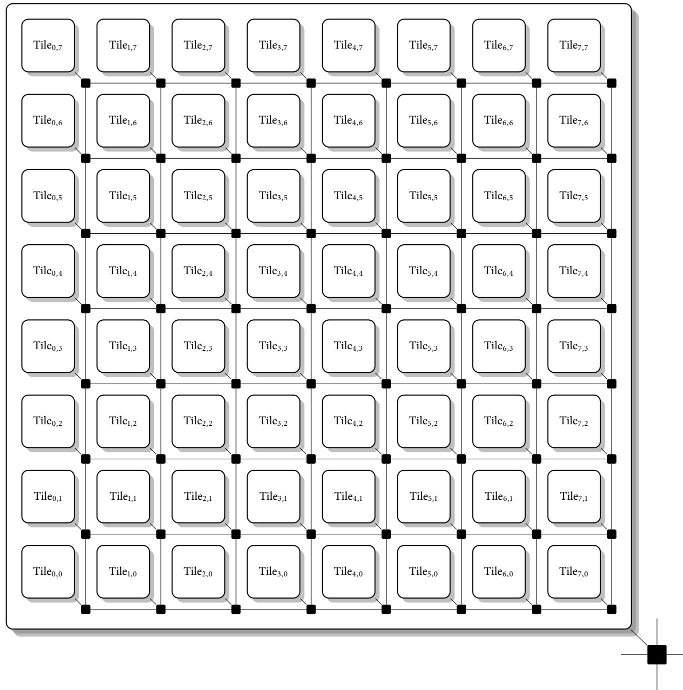


Figure 3.2: Concept architecture

3.3.3 *A conceptual tiled architecture for radio astronomy*

From the previous sections it followed that a single reconfigurable processor can handle a light beamforming application, but for heavier applications more processors are needed. In particular for the generic system for satellite reception, radar, radio astronomy and wireless and mobile communication, as defined in chapter 2, the architectures as discussed in the previous sections will be far too small.

In this section we introduce a concept architecture for the radio astronomy application LOFAR. This concept architecture is homogeneous and hierarchical, consisting of 64 processing tiles per SoC (shown in figure 3.2) and 64 SoCs per board. Each tile can compute 200 M ops performing MAC operations. The on-chip interconnect has a capacity of 400 MB/s (full-duplex), while off-chip connections have a bandwidth of 1 GB/s. These are quite realistic figures; in the EU CRISP project a chip with 9 similar cores running at 200 MHz was developed [20, 98].

LOFAR LOFAR is a phased array for radio astronomy applications at low frequencies (shown in figure 3.3). As explained in section 2.3.1, it consists of 77 stations with 96 antennas each. Each analogue front-end can select between a low band from 15 MHz to 80 MHz or a high band from 110 MHz to 240 MHz. Direct digital conversion with a 12 bit 100 MHz ADC is used allowing simple front-ends without mixers and improving the performance [21].

Figure 3.3: LOFAR station

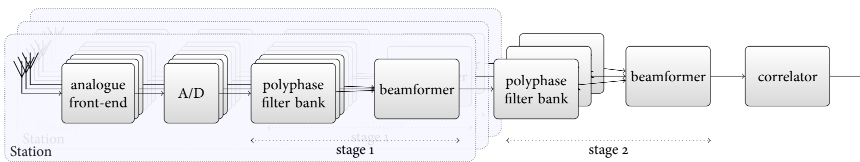


Figure 3.4: LOFAR processing chain

The processing chain is based on a Fourier transform followed by beamforming, in turn followed by correlation. All three techniques provide selectivity in their own domain, i.e. Fourier transform in the frequency domain, beamforming in the spatial domain, and correlation in the time domain. In LOFAR the original Fourier transform is replaced by a polyphase filterbank, where a polyphase filter bank consists of a decimation over a bank of FIR filters, followed by an FFT.

The (simplified) LOFAR processing chain is shown in figure 3.4. The polyphase filterbanks followed by beamforming are performed in two stages to realise hierarchical beamforming. The first stage is per station (96 antennas per station), the second stage combines the signals from all stations. In the first stage there is a polyphase filterbank for each antenna which splits the antenna signal into 512 sub-bands, of which 216 are selected. Each polyphase filterbank uses 1024 16-tap FIR filters and a 1024-point FFT, where the negative frequencies are thrown away resulting in 512 sub-bands. For each station of 96 antennas the results of the polyphase filterbanks are beamformed in a maximum of 24 beams.

In the second stage the polyphase filterbank splits the resulting signals into 256 sub-bands of about 1 kHz each, after which the beamformer combines the signals of all stations. Finally, the resulting signal is correlated.

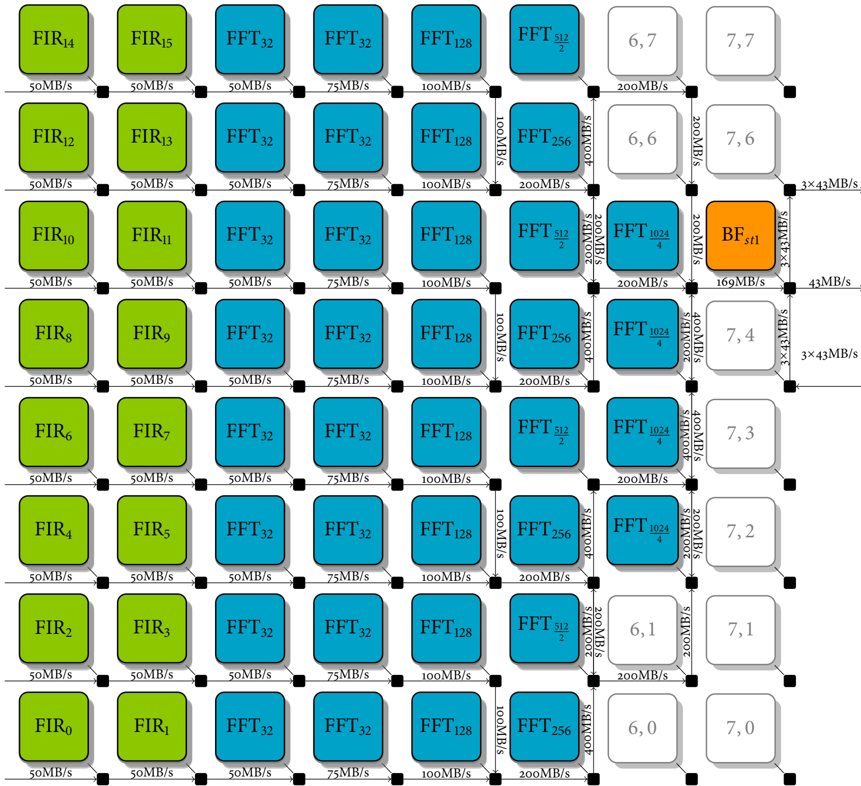


Figure 3.5: LOFAR mapping for each antenna

Mapping We have mapped the processing chain for LOFAR on our concept architecture [15]. The mapping of the first stage is shown in figure 3.5. Each tile in our concept architecture has sufficient processing capacity to calculate 64 FIR filters per sample, such that 16 tiles are needed for all 1024 FIR filters for each polyphase filterbank. To distribute the 1024-point FFT, 36 tiles are needed.

Concerning the communication bandwidth, each link has a 400 MB/s capacity in each direction. The actual bandwidth which is needed is indicated in figure 3.5. As can be seen, there are two links where the maximum capacity is reached.

Such a SoC is required for each antenna of a station, resulting in 96 SoCs. Finally we remark that the beamformer (BF) is distributed over all these 96 SoCs plus four extra SoCs to combine the results of the 96 outputs of the distributed beamformer.

Without going into details we remark that the data reduction of the first stage is so big that the second stage can also be executed by the four extra tiles mentioned above. The end result is a *single* beam. For the correlation another SoC per station is needed. For further details, see [15, 21, 22, 36].

In total each station requires 101 SoCs (1 for each antenna, 4 for the later stages and 1 for correlation) each having 64 tiles with 200 M ops, giving each station a processing capacity of $101 \cdot 64 \cdot 200 \approx 1.3$ T ops. With 77 stations, the total processing power is $77 \cdot 1.3 \approx 100$ T ops. About another 60 T ops are needed for computing maximally 24 beams, and for post-processing and control [21].

It turned out that the mapping of the computations is relatively straightforward, whereas the mapping of the communication is more difficult because there are many data-streams at different rates which must be synchronised. Besides, some of these data rates are close to the maximum capacity of the network links. For these reasons we performed a simulation of the communication infrastructure of our conceptual tiled architecture.

Simulation On tiled architectures, computations are distributed, thereby requiring communication. We have simulated the communication infrastructure in the tiled concept architecture in SystemC [15]. It uses FIFO buffers and back-pressure (full buffers stall the computation) for ordering and synchronisation, and for decoupling the sample rates of ADC tiles (that represent the antenna input signals) and the clock speed of the NoC. The use of FIFO communication and back-pressure are based on concepts from dataflow process networks (PNs). We will come back to this during the discussion below.

In this simulation we tested three scenarios. In the first scenario we simulated the decoupling of the sample rate of the ADC from the clock speed of the NoC. The simulation was performed with 200 MS/s ADCs and a NoC clock speed of 200 MHz, 100 MHz, 150 MHz and 101 MHz. Note that the ADCs produce 16 bit samples and the NoC operates on 32 bit words, so there is enough bandwidth in all four speeds. In the first two cases, the rates are the same or an even multiple requiring no decoupling. In the last two cases the rates are different, requiring buffer space to handle the delay when the clocks do not match. In all cases the FIFO buffers successfully decoupled the data rates.

The second scenario connects four ADCs to a single processing tile. The first connection uses five hops, the second four and the third and fourth use two hops. The ADCs again run at 200 MS/s and the NoC at 101 MHz. In the test scenario the buffer spaces were large enough to successfully stream data to the processing tile, be it that shorter paths need more buffer space to delay the samples until the sample from the longest path arrives. Thus, the buffers synchronise the data streams at the processing tile.

In the third scenario, configuration data is periodically added to the data stream varying the data rate. By adding the configuration data to the data streams, the processing tiles reconfigure as the data flows through the system ensuring all tiles reconfigure with respect to the same sample. This avoids the computation of useless data because the system is computing results that are half processed in one configuration and half in the next. Results show that spare network and processing capacity successfully deals with the varying data rate and the reconfigurations are synchronised with the data.

3.3.4 *Discussion*

The experiments show a large number of tiles are needed, i.e. even for the smallest application (DVB-S) we need much more than three tiles (at least 64 times more to process 256 instead of 8 antennas, and that is assuming a single tile can handle the data-rate of DVB-S).

The tiles provide modular building blocks which are used in combination with a NoC to extend the architecture with additional processing capacity. The reconfigurability provides flexibility. Yet, a tiled reconfigurable architecture requires that the application is partitioned so that it can be mapped on the SoC and that communication is explicit so that it can be mapped on the NoC.

In order to support the partitioning of applications and to reduce the effort of managing communications, thereby improving programability, the use of dataflow models is proposed. An introduction to dataflow is provided in appendix A; here we only mention the relevant properties of dataflow models.

A partitioned application is represented as a set of dataflow processes, connected by channels. Thus, processes represent computation and channels represent communication. Such a representation facilitates mapping the partitioned application (the processes) onto a SoC. Processes can only have explicit communication via channels, i.e. they can not have shared state. For the beamforming application, shared state introduces a central bottleneck to the memory requiring a huge amount of bandwidth because of the high data rates of the processed streams, and should therefore be avoided. Additionally, the explicit communication facilitates mapping the communication streams onto the NoC. For beamforming, data streams must be synchronised so that the correct data (data with the same sample time) is beamformed, even though the data can take different routes through the network thereby experiencing different delays. In addition, managing data becomes easier if the data can be assumed to be in-order and that no data loss occurs during communication. These properties are provided by the channels, as data in the channels remains ordered and if a channel is empty a process is stalled thereby synchronising the inputs of a process until all input data is available. These dataflow concepts were already used above in section 3.3.3 to decouple data rates and to synchronise data streams. In summary, the use of dataflow models forces the use of ordered data and separate state at the applications level, thereby allowing the application to be partitioned and mapped on a tiled reconfigurable architecture. A further more detailed motivation for the use of dataflow in embedded systems is given in section 4.1.3.

3.4 Conclusion

In this chapter we have explored tiled reconfigurable architectures for the application domain of beamforming applications. Resulting from the requirement of a generic beamforming platform, we find that such an architecture must support distributed processing as the large number of antennas (over 256) and high data-rate (over 50 MS/s) of the beamforming applications do not allow enough time to pro-

cess all computations on a single processor. Furthermore, with such high-data rate streams, even with distributed processing, each processor can compute only a few operations before the next sample arrives, thus there is a relatively large amount of communication per computation. Finally, the architecture must provide enough flexibility to execute the various beamforming applications, as well as the different beamforming methods and the ability to switch between searching and tracking algorithms.

Tiled reconfigurable architectures possess scalability, flexibility, and efficiency. As such, they could provide a suitable architecture for a generic beamforming platform.

Three experiments with beamforming on tiled reconfigurable architectures confirm the above characteristics. As a first example, we have implemented audio beamforming on a single reconfigurable tile, supporting multiple beamforming methods by reconfiguration, and a beamcontrol algorithm. Thus, reconfigurability provides sufficient flexibility, but for the applications presented in chapter 2 a single tile is not enough. The second example implements a beamformer for the DVB-S application on an architecture with three reconfigurable tiles. From this implementation it follows that beamforming can successfully be partitioned over multiple tiles with a beamcontrol running on another tile. However, available realised tiled architectures are too small for even the smallest of the beamforming applications. Therefore, the third example is a conceptual tiled architecture for a large radio astronomy application, LOFAR. The architecture consists of 64 tiles, that can perform 200 M ops each, and full-duplex 400 MB/s network links. LOFAR requires 101 SoCs for each of its 77 stations, totalling 100 T ops. The partitioning and mapping of the computations for LOFAR is relatively straightforward, however, mapping the data streams that are communicated is more difficult because there are many data streams at different rates which must be routed and synchronised.

We conclude from the above experiments that a tiled architecture does provide the necessary scalability and flexibility, but that a large number of tiles are needed for the beamforming applications presented in chapter 2. A second conclusion is that executing a large beamforming application on a tiled architecture requires partitioning of the application and thereby explicit management of communication. Furthermore, the implementation on the used reconfigurable processor requires a lot of programming effort.

The dataflow domain provides a useful model for partitioning; processes represent parts of the applications and channels represent explicit communication. Furthermore, FIFO-buffers as an implementation of channels provide ordering of data, synchronisation and decoupling of data rates. These features can be exploited to add configuration data to data-streams, thereby synchronising reconfiguration with the data stream, which was verified for the last experimental architecture.

Chapter 4

Model-based design of multi-domain systems

Abstract – *The complexity of designing embedded systems requires a unified model-based design approach. Analysis of the application domain has shown that we need a mixed CT and DT model to include the environment and analogue hardware for verification. Furthermore, the choice for a tiled architecture necessitates partitioning the application, for which we use the DF domain. In this chapter we will present a unified perspective on signals and components for these domains. There are many tools for mixed-domain modelling, but current tools have problems modelling time. Furthermore, a survey of existing tools shows there are no tools supporting model-based design with model transformations for the CT, DT and DF domain. Therefore, we will propose such an approach called UniTi. UniTi is based on mathematical definitions of models to support unified mixed-domain modelling, including exact time delay components and model transformations. It is supported by a design-flow that uses these model-transformations to transform a (formal) specification into a division over the environment, the architecture (analogue and digital hardware) and the application (software), as well as a partitioning of the software over a tiled architecture.*

Designing, modelling and verifying embedded systems is a big challenge; one of the key problems being the interaction of the system with the physical world (its environment) leading to different views on for example time. A second problem consists of the strong requirements concerning the correctness (the behaviour of the system adheres to the specification) and robustness (coping with errors and unexpected inputs from the environment) of the hardware and software. Consequently, the need to design embedded systems in an integrated fashion, including

Parts of this chapter have been published in [KCR:3], [KCR:8], [KCR:10] and [KCR:14].

that verifiable correctness, is widely recognised [13, 17, 53]. Furthermore, when designing such complex systems it is useful to apply model-based design, i.e. the iterative and incremental development of a single reference model, because it shortens the design cycles and integration is part of the design process early on.

A specification of an embedded system describes its functional behaviour and a set of requirements. For a design process based on model-based design, such a specification typically includes a formal specification of the functional behaviour. Such a formal specification forms the initial model for the design process.

In this chapter we will first motivate the need for model-based design in more detail. Such a model-based design approach needs support for multiple domains. This includes exact modelling of the CT domain to support the interaction of an embedded system with the environment (among others), as we have found for the phased array beamforming application in chapter 2. Furthermore, the DF domain needs to be included for modelling applications running on a multi-core SoC with a NoC, as we found in chapter 3 for beamforming applications on a tiled reconfigurable architecture. Finally, as the starting point is a formal specification, it very useful to be able to define models using mathematical definitions.

Thereafter in section 4.2, we will discuss signals and components in mixed domain systems and we will present a unified perspective on signals and components based on time for the CT, DT and DF domain, including their interaction. Current mixed-domain modelling tools perform simulations by discretising a global (simulation) time and advancing the simulation by passing values between model components at each time step. This introduces inaccuracies when time transformations such as time delays are used, which is analysed in detail in section 4.3.

A survey of current modelling tools is presented in section 4.4. As we will see, there are no tools supporting model-based design using mathematical definitions and model transformations, and supporting the CT, DT and DF domain, nor are there tools providing exact modelling of time transformations in the CT domain. Therefore, UniTi is proposed in section 4.5, a design flow and modelling and simulation framework that does support all these aspects. UniTi forms the basis of a design flow that uses model transformations for the design steps in section 4.6.

4.1 Motivation

The complexity of today's embedded systems requires a simultaneous consideration of the environment, the hardware and the software when designing the system. This includes aspects such as channels, an analogue front-end and digital processing, but also concurrency and robustness of the software. All these aspects are often interdependent and thus must be simulated and verified in a single model. Such an approach is supported by model-based design, which is presented first, followed by a discussion on the necessity to integrate the environment, and by an elaboration of the usefulness of the dataflow domain for modelling software. Finally, we will motivate the usefulness of mathematically defined models.

4.1.1 Model-based design

In this section we will motivate the need for a single model and a model-based design approach that iteratively develops this model with transformational design steps.

4.1.1.1 Systems engineering

A typical design approach for complex multi-domain systems such as embedded systems is systems engineering [13]. Systems engineering is a design approach which aims at using a holistic view with a life-cycle orientation that addresses all phases of the system design. Throughout the design it attempts to unify all involved contributors into an *interdisciplinary* effort. It uses well defined and specified *system requirements* which can be verified and validated down to a detailed implementation.

System design is greatly aided by the use of models, which provide an abstraction at different levels of detail or functionality. The models can also complement each other by providing different views of the system.

An often advocated method that fits the systems engineering approach well is model-based design [13]. Traditional design follows the waterfall model [13], which has the disadvantage that the next design phase has to wait for the previous to finish and therefore making changes late in the design cycle very costly. The incremental and iterative design steps of model-based design addresses these shortcomings by integrating part of the design as soon as possible and refining the design with small steps.

Designing is performed using a *top-down* approach to decompose a larger system into smaller blocks, to make sure these blocks effectively fit together and to manage complexity. The design steps consist of analysis (goals and research), synthesis (development and implementation) and evaluation (verification and validation) and are illustrated in figure 4.1). During the research phase the goals and requirements are analysed and different options to satisfy the requirements are explored (diverge). Ideally, this results in a formal specification of the selected solution (converge). The development phase uses this formal specification to synthesise a system in several steps. During evaluation the synthesised system is verified, by tests or simulation to conform to the requirements and specifications. Validation confirms whether the goals are met or not. Depending on the results the design can be refined, starting the next design cycle.

When designing a mixed signal embedded system the traditional approach uses a combination of e.g. mathematics for analysis, SysML/UML for (system) modelling, Simulink for functional simulations, SystemC or a hardware description language (HDL) for digital hardware implementations and C for software implementations [13, 63, 70, 71, 99]. This means a number of tools are used, each with its own model of the system. This complicates holistic iterative system design and makes the trade-off of what to do in which domain more difficult.

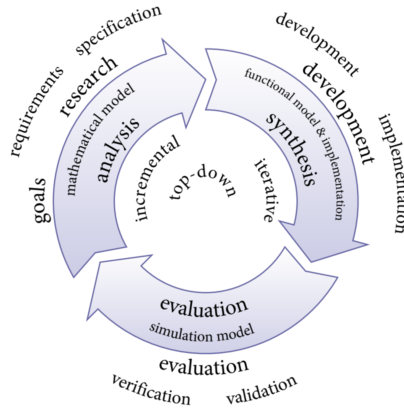


Figure 4.1: Model-based design process

4.1.1.2 Model transformations

Key to model-based design is the use of a single model with a transformational design process, i.e. model transformations are used during the design steps. For example, in order to decouple the system design from an architecture, a high level model should be architecture independent and a model transformation can be applied to generate a model for a specific architecture.

For each step of the design process, more detail is added to the model. The initial model uses a more abstract implementation of the functionality. During analysis the design is decomposed into smaller blocks. The smaller blocks are extended by supplying more detail per block. In the development phase the more abstract blocks model (ideal) functionality, and these blocks can be combined with more developed blocks that also include implementation details. This facilitates interdisciplinary work, such as when performing hardware/software co-design where it's not yet clear which hardware will be used or when different options are evaluated. The implementation itself can consist of for example block schematics, hardware or software. Simulating the model during the design process helps to evaluate design decisions and test implementations.

When performing a model transformation, we must ensure the transformed model is still correct, i.e. that the functionality has not changed. Thus, model transformations must be verifiable and correctness preserving.

4.1.1.3 Design space exploration

Another important advantage of model transformations is that it facilitates design space exploration. Decomposing a system involves division *over* domains as well as division *within* a domain and often includes design decisions and trade-offs which must be evaluated by the designer. Thus, it is very valuable for the design process to

evaluate the results after a transformation and to be able to revert the transformation and try alternatives. For example, a model can be transformed to an equivalent model which represents a mapping to an architecture, or the transformation maps the model to a predefined architecture.

4.1.2 Environment

Many embedded systems interact with their physical environment. For example, a mobile phone sends and receives radio signals, has light and proximity sensors, a microphone and speakers, etc., all interacting with the environment. This environment and the analogue interfaces are modelled in the CT domain. The computations of the system are performed in the DT domain by digital hardware. For example, in a radio receiver the CT processes include the actual radio signal together with the distortions introduced by a channel (noise, interferers, etc.) and the analogue front-end of the receiver, whereas further processing of the digital hardware consists of filtering, demodulation, error correction, etc. in the DT domain.

Moreover, the processing in the DT domain can adapt and react to changes in the environment. This introduces feedback that spans multiple domains. It is therefore important to be able to analyse and verify these interactions in a single model that includes all domains. A system with both continuous and discrete dynamic behaviour is called a *hybrid system*. As an example to illustrate the necessity to integrate the various domains, we mention an ADSL modem, which uses adaptive transmission based on cable conditions. We must include the noise and distortions of the cable to exactly verify the correct operation of the coding, modulation and error-correction, as worsening or improving cable conditions result in different communication modes that are negotiated between the modem and a cable network controller. Without including the environment in the model, a hardware prototype must first be developed to verify the correct operation. Clearly, this is far too late.

Current design tools (see [17] for an extensive survey of multi-domain system design tools) implement the interaction between the above mentioned domains by discretising a global simulation time and representing signals as a sequence of values. This prevents exact transformations with respect to time, such as for systems with variable time delays or multi-rate systems. This time step should be small enough to meet the requirements of *all* the components in the system, as the whole system under design is evaluated at this time step. However, this may result in extremely inefficient simulations as several components, such as e.g. integration, may require a very small time step to achieve enough accuracy. In addition, it will not be possible to capture all timing issues in one global clock. For example, a time delay may depend on changing circumstances in the real world and thus will not be predictable during simulation. Since existing simulation tools also consider CT as a sequence of discrete values, the best they can offer for the value of a delayed signal is an interpolation between known values on moments that are close to the delay. Here too, to make this interpolation accurate, the chosen time step has to be very small. Summarising, both aspects — using values for continuous

functions and a single global time step for simulation — cause that simulation either becomes less accurate or less efficient. We will discuss these problems in more detail in section 4.3.

Modelling the physical environment should be *exact* in order to correctly validate the design. Approximations like interpolation introduce inaccuracies that are modelling artefacts; we can not differentiate between the error caused by the modelling tool and an error caused by an incorrect specification or implementation. For example, testing algorithms for wideband beamforming or time shifted sampling relies on this; signals arrive at different antennas with different time delays because of path length differences. Even for narrowband beamforming; if the antenna elements are moving with respect to the source, they experience the Doppler effect. For antenna elements with different speeds, for example on a moving ship, this effect can not be represented by a simple frequency shift. Instead time-scaling must be applied. Other examples are path length differences for differential pairs or network delays.

Note that in this thesis the term *exact* is used in the context of a simulation tool executing on a computer. As such, the exactness of the results are inherently limited by the machine precision. However, in the approach we will present, the accuracy is limited *only* by the machine precision, while other tools will introduce much larger inaccuracies in the order of the step size used for approximation. As such, we will use the term *exact* for results limited by the machine used for simulation in contrast to limited by the tool used for simulation.

4.1.3 Dataflow

Software for streaming applications can be modelled quite naturally with dataflow models. A dataflow model is a graph of nodes (processes) connected by edges (channels); data tokens are processed inside nodes and sent from one node to another through the edges. Tokens are abstract in the sense that they may have any internal structure. A process may consume and produce several tokens at a time; when there are not enough tokens available on the input edges of a node, that node will not execute (fire). The condition that enables firing is called the firing rule. Consumption and production of tokens is instantaneous in the model, while a process can have execution time. Note that executions can overlap: if enough tokens are available to fire, the process directly executes even if the process is already executing. This can be restricted by introducing self-edges, i.e. channels that loop back to the process itself. Dataflow is presented in considerably more detail in appendix A.

Dataflow models are used for representing software for multi-core systems. The DF domain provides a model for stream processing with explicit communication. Processes in the dataflow model represent computations and channels represent communications. One or more processes can be mapped to processing tiles on a SoC, while the channels are mapped on its NoC.

It is important to differentiate between dataflow models, dataflow analysis and dataflow execution. The application executing on a multi-core platform is mod-

elled as a dataflow graph, representing a partitioned or parallelised application. Dataflow analysis of this model then offers a prediction of the real-time performance characteristics for a real hardware platform instantiation. Dataflow execution semantics ensure we do not have to worry (from the application's perspective) about losing data or receiving data out of order, i.e. communication and synchronisation, when running applications on this platform.

Application modelling Dataflow processes can be seen as mapping sequences of inputs to sequences of outputs, or functions on streams [55]. They are therefore useful for parallelising streaming applications [11] and mapping streaming applications on multi-core architectures [39, 45, 115]. Processes are mapped onto computational resources and channels are mapped onto communication resources.

As processes are independent, they may not influence each other besides the explicit inputs and outputs, i.e. dataflow processes must be side-effect-free. This ensures the processes have no shared state, all communication is explicit, thereby allowing us to map one or more processes to independent tiles. Furthermore, these processes can be moved to different tiles when necessary.

Verification When designing embedded systems, we must be able to verify the correct operation and the performance of the system. This also includes the software of the system. Verification can be performed by simulation or by using analysis techniques.

For analysis of the dataflow model we need to take into account execution time, communication, buffer capacities and scheduling. Dataflow models have no notion of time, only ordering. For metrics such as throughput and latency to make sense, and allow them to be determined by the analysis techniques, processes in the dataflow model are annotated with execution time. Methods are then available to analyse deadlock and race conditions, to calculate buffer sizes, and to determine or estimate latency and throughput of tokens streaming through the graph (see e.g. [115]). The result of the analysis is then used to properly dimension the system [39, 115].

For simulation, the dataflow model is executed, requiring an execution model (determining how a model is executed for simulation). Our execution model for the DF domain is presented in section 5.1.3.

Abstraction Dataflow models have a number of properties that provide a useful abstraction from scheduling, communication and synchronisation details.

All dataflow models have self-timed execution, i.e. a process can execute as soon as all input data is available. Therefore there is no need for global control of the execution. However, for restricted dataflow models with fixed token rates, a static schedule can be determined on beforehand, removing scheduling overhead completely [57].

In the DF domain, tokens in channels remain ordered, as dataflow models are order-preserving. As dataflow models are deterministic, data can not be lost during communication.

Channels are of unbounded capacity, but buffers between processes are modelled by two channels in opposite directions; one carries the tokens to be communicated and the so-called back-edge models empty space in the buffer. A process can therefore only execute when there is enough space in its output buffers. Thus, a process is stalled if the tokens are not consumed from the output buffer fast enough by the next process. This is called “back-pressure” and results in automatic synchronisation in parallel execution.

4.1.4 Mathematical foundation

The model-based design approach typically uses a formal specification, which provides a mathematical description of the functionality of the system. This specification forms the initial model used during the design process. As such, it is very useful to represent models using mathematical definitions[1].

4.1.4.1 Mathematical definition

A mathematical definition of models allows one to easily define and extend the model from the (formal) specification. The specification thus provides a convenient initial model, against which model transformations can be verified. Such model transformation are also formally specified, and can thereby be proven to be correct.

A mathematical model also represents a structural hierarchy. Functions represent model components and such functions can be defined using other functions (sub-components). Furthermore, the arguments and results of functions can be named and as such be used to represent connection between components.

The mathematical model is evaluated by calculating the result when applying the function to an input. Evaluating the function over a range of inputs is akin to performing a simulation of the model. In a sense we use executable mathematics for simulation. We define that the output of a function is calculated as soon as all arguments of the function are available.

Mathematical equations define relations and all equations are valid simultaneously. A mathematical model is therefore parallel by nature.

A mathematical model is very suitable as a model for system design, because of the qualities described above. The idea of executable mathematics is not new. In fact there is a whole research field on reasoning about programs and ensuring a “correct and meaningful correspondence between programs and mathematical entities in a way that is entirely independent of an implementation” by providing the denotational semantics of a program [89]. Surprisingly, there are only a few model-based design tools with a mathematical basis. We will discuss these tools further in section 4.4.

[1] As a model only provides an abstraction which can be represented in many ways, a mathematical definition is not a necessity for model-based design.

4.1.4.2 Functional languages

In a functional language, a computation is considered as evaluating a mathematical function, i.e. the basic method of computations is applying a function to an argument [47]. Programs are defined by equations instead of statements as in imperative languages. A functional language is therefore close to mathematics and provides a nice fit to represent mathematically defined models in.

Functions are first-class in functional languages; they can be used as arguments and results of other functions. Such functions are called *higher-order functions*. Higher-order functions also enable *partial application*; a function is applied to part of its arguments and returns a new function on the remaining arguments. Instead of applying a function on a tuple of arguments, a function can also be applied to its arguments one by one, called *currying*.

We choose the functional language Haskell [1]. Haskell is a statically typed, lazy, pure functional programming language. *Pure* means that a function has no side-effects (all observable effects of the function are its results) and the evaluation of a function always gives the same results for the same arguments, as we expect of a mathematical function. *Lazy* means the evaluation of the arguments is delayed until its value is actually required. *Statically typed* means the types of functions are checked at compile time. Haskell also has type interference, i.e. the types are deduced from the function definition and its context. For more background on functional languages, we refer to [47, 72].

4.2 Time, signals, components and systems

Systems in the field of engineering are often divided into *components* that interact, in order to manage the complexity of designing systems. For CT and DT systems, the interaction between components is by *signals*, which represent measurable quantities over time for transmission of information, or more general, signals represent data over an independent value such as space or time. A system is characterised by how it responds to input signals, in other words, components denote *signal transformations*.

4.2.1 *Continuous and discrete time signals*

We differentiate between continuous time and discrete time. Continuous time is unbroken or whole, i.e. defined for all time. Discrete time quantises time to distinct separate moments in time. Thus there are two kinds of signals:

- Signals in the CT domain are functions of time, i.e. they represent the value of the signal over all time.
- Signals in the DT domain are values at discrete moments, also called samples.

These representations are a conscious choice. For a CT component, the input signal represents a function over all time. Therefore, its time reference can be changed,

the signal can be delayed or time can be scaled (e.g. speed-up). However, for a DT component the input signal is a value at a discrete moment in time. This value is linked with a sample time, but the DT component can not and should not be able to influence this time. From the perspective of the DT component, the input signal is just a value, but a value that changes over time. Therefore, the input signal of a DT is also a function of time, however, this time is not accessible to the DT component, i.e. time is defined at a higher level for the DT component and is irrelevant for its operation. We do not want a DT component to operate on a list of values over all time either, because this would give the DT component access to all future and past values. Only having access to the current value forces a DT component to make its state explicit, i.e. state is an explicit input and output of the component. As such there is only a notion of the next value. Thus, the state and output of an operation change with a new input value, irrelevant at which time this is.

Summarising, a function that takes a signal and transforms it to a new signal is called a *signal transformation*. In case of DT signals, the implementation of a signal transformation corresponds to an operation or function on a value, i.e. the signal is a value. In case of CT, a signal transformation corresponds to a function on a function (the signal as a whole), i.e. it is a *higher order function*. As such we have extended the classical meaning of signals a bit, although signals still conceptually represent a time-varying quantity. Furthermore, a CT signal transformation can change the time reference, e.g. the output signal is a time delayed input signal. Other transformations with respect to time are reflection or scaling.

4.2.2 *Signal flow diagrams*

Signal flow diagrams or block diagrams are popular in system design tools because of their intuitive use and ease of understanding [17]. We will shortly discuss their connection with components and signals. Components in such diagrams denote signal transformations and arrows denote signals. Composition of signal transformations is analogous to connecting blocks in a signal flow diagram.

To compose components from different domains, the signal representation must be changed. To go from the CT domain to the DT domain, the signal is sampled at specific sample times by an ADC. To go from the DT domain to the CT domain, the sample is held until the next value by a digital-to-analogue converter (DAC).

4.2.3 *Signals and components in dataflow models*

We will generalise this notion of signals and systems to include the DF domain. This enables us to include DF in a model with CT and DT, where components and their interaction have the same meaning for the DF domain as for the CT and DT domain.

An important observation is that processes of a dataflow model also transform data. Therefore, components in the DF domain can also be represented as *signal transformations*. In the DF domain, signals are lists of tokens. In *all* domains com-

ponents are signal transformations. However, the perspective on the data that is transformed in each domain is fairly different, as is the representation of signals in each domain. To go from the DT domain to the DF domain, samples become input tokens for the DF component. To go from the DF domain to the DT domain, output tokens become samples for the DT component. A DF component can be combined with a CT component via a DT component.

Furthermore, a dataflow model has no notion of time but the CT and DT domains do. However, when a DF component is combined with CT or DT components, the latter domains determine the time that tokens for the DF component are produced (sampled) and therefore introduce a time reference to the DF domain. Execution time of the DF component then determines the time at which the output tokens are produced. Note that the DF component can only be connected to a DT component, because we assume the sample time of the DT domain determines the production time of the corresponding DF token.

4.2.4 *Other domains*

The DF domain is related to the synchronous/reactive (SR) domain. Synchronous means that computations are considered instantaneous and reactive means the model reacts to events from the environment. In the SR domain, physical time is replaced by an ordering at global clock ticks. The DF domain abstracts time even further; there is no *global* ordering, only dependencies. For completeness: the discrete event (DE) domain defines passage of time between clock ticks, while for the DT domain all samples have a corresponding physical time [60]. In this thesis we will limit the scope to the CT, DT and DF domains.

4.3 The problem with time

Consider a simple system consisting of a continuous time sine source connected to a time delay block, followed by an ADC, a bias (offset) and a sink which plots the output. The CT part and DT part are indicated in the figure:

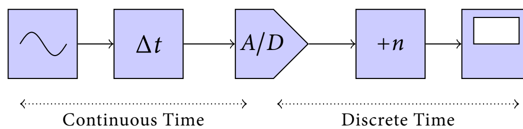


Figure 4.2: Mixed CT/DT system block diagram

4.3.1 *Notions of time*

In such systems we identify different notions of (modelled) time:

- the instants when the designer wants to know the behaviour of the system, the *simulation time*,

- the instants when continuous information from the environment is sampled by, say, an ADC, the *sample time*,
- the time steps that are necessary to numerically approximate functions (e.g. an integral), the *approximation time*,
- the time that has elapsed during processing, the *execution time*,
- the time locally, possibly transformed by e.g. a time delay, the *local time*.

The last notion of time is necessary to represent relativity: different distances from a source lead to different local time references relative to the source. From the perspectives of components at different distances, the source is at a different time, yet the source is defined for a single time reference. Therefore, each component at a certain distance must have its own local time reference to the source, i.e. time is a local property and time is relative. This occurs for example for a front-end with multiple signal paths, which might have slightly different path lengths, thereby modelling non-ideal common mode noise rejection.

We will show that time must be a local and relative property of a component in the model to accurately model time transformations.

4.3.2 Global solver

There are many mixed continuous/discrete time modelling tools [17]. Existing tools perform a simulation by extracting a set of ordinary differential equations (ODEs) from the model. Some tools such as MapleSim (see section 4.4) optionally perform symbolic simplifications on this set. Then the set of equations is (in the general case) solved numerically, i.e. such solvers numerically approximate the differentials. For a mixed CT/DT model, signals in the DT domain are represented as piecewise-continuous for the solver.

Typical solvers used in tools are the Euler methods or the Runge-Kutta methods [61, 99, 102]. Such solvers operate iteratively with a fixed or variable step size. Each step the equations are evaluated and the results updated according to the algorithm.

This iteration step is a time step, i.e. iteration is performed over time. Time is thus a global property of the model as it is applied to the complete equation set, which represents the complete model. Furthermore, the same solver is used for all components in the model for the same reason.

4.3.3 Discretisation of time

Simulation for existing tools discretise global time into time steps to iteratively solve the set ODEs over time. Thus, although the different notions of time are in principle unrelated, they are coalesced into a global time. At each simulation step the complete model is evaluated and a resulting value is calculated.

So, the inputs and outputs of the model components are the values at a certain global time step. This means that the time step determined by the solver for numerical approximation of a differential equation is applied to all equations. The

global time step causes the whole system to be evaluated, while it is very well possible that most of the system does not need to be evaluated at this fine-granularity, reducing efficiency; for example, the DT domain most likely has a sample period much larger than the approximation step, of e.g. an integral in the CT domain, which often has to be very small for sufficient accuracy. We have experienced this when simulating one second of a phased array antenna system, which took hours in Simulink (also see chapter 6) because of the small time step needed for sufficient accuracy.

However, as we will see next, discretisation of a global time is even more problematic in case of time transformations.

4.3.4 Time transformations

In summary, existing simulation tools do not distinguish different notions of time. All the notions of time are coalesced into a global (simulation) time. The solver must determine the time instants to solve the set of equations. Furthermore, there is a single representation of signals as values at a certain simulation time.

Consider the very simple Simulink system in figure 4.3 consisting of a sine wave source, a (variable) time delay and a scope. A 1 Hz sine with a 0° initial phase is used. The (maximum) time step size for simulation is 0.04 s, so 25 simulation results per period. The step size is not adjusted by Simulink, because the model contains no differential equations (only algebraic). The time delay is 0.1 s, which is deliberately chosen not to fit the step size.

A plot of the simulated output is shown in figure 4.5, a shifted sine wave as expected. The ideal result is a sine wave with an initial phase of $-2\pi \cdot 0.1$, but when compared to the output of the delayed sine wave of figure 4.3 the Simulink output has an error (shown in figure 4.6). This error is caused by interpolation. The time delay block buffers values each time step and retrieves a value for the delayed time. When a value at the delayed time is not available, the result is interpolated from the surrounding values. The error is directly related to the frequency of the signal and the step size. Higher frequencies need smaller steps or will give larger errors. When the simulation time steps are large compared to the frequency, this error can be quite substantial.

One might expect that a solution is to make the time step a *multiple of the delay* as the delay block can then retrieve the *exact* value. There are at least three problems with this:

a) if the delay is small the step size needs to be small, resulting in many (fixed) steps and thus inefficiency,

b) if multiple time delays are used, without a common factor, separate time steps for each time delay are needed for accurate results,

c) if the delay is variable, the current time step depends on a unknown delay in the future.

As in general the time delay *can* be variable, in Simulink the delay is not even taken into account when determining the time step. The consequence is that if a value

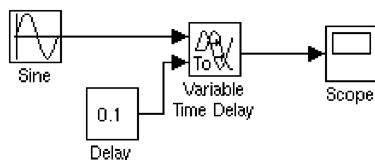


Figure 4.3: Simulink example

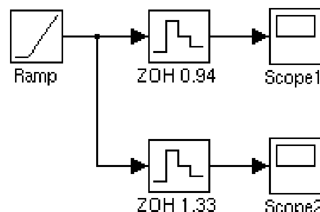


Figure 4.4: Model with multiple ADCs

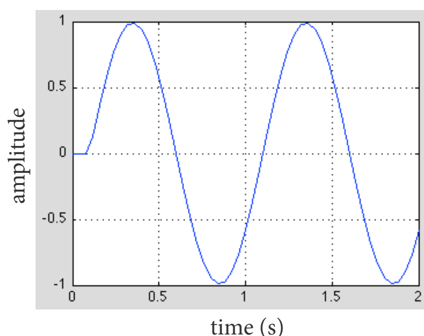


Figure 4.5: Delayed sine wave

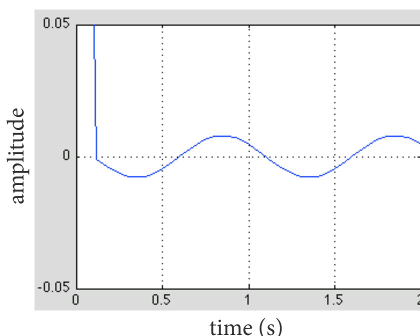


Figure 4.6: Time delay error

is not available at the exact time, it is interpolated between available values, giving results that are not *exact*.

A second example of a problematic model is shown in figure 4.4. An ADC, implemented as a zero-order-hold (ZOH), *is* assumed to have a fixed sample rate, i.e. a ZOH holds the value of the input at the beginning of a fixed period for the rest of the period (otherwise a sample-and-hold should be used, which has an explicit input for triggering a possibly variable sample moment). The simulation time steps are indeed synchronised with and determined by the sample rate for both fixed and variable step sizes in Simulink. For a multi-rate system with more ADCs the time steps match the sample times of *all* ADCs with a variable step. However, a fixed step size results in a very small time step, matching the common denominator. Already for more than two ADCs the time step becomes too small and generates an error in Simulink.

In section 4.4 we will present a survey of mixed domain modelling tools in relation to their ability to model time in the CT domain among others. We will find that there is no such tool that can deal with time in an adequate manner. Therefore and because of other shortcomings we propose a novel modelling and simulation framework in chapter 5 that enables local control over the time, i.e. by locally applying time transformations or time steps the problems described here can be resolved while retaining efficiency.

4.4 Survey of existing tools

There are many mixed domain modelling tools. We will first give an overview of the major players and then relate them and others to exact continuous time modelling, multi-domain modelling support, support for mathematical definitions, model transformation support, and parallelisation support.

4.4.1 Major tools

An extensive survey of languages and tools for hybrid systems can be found in [17]. Some of the better known tools are Simulink for CT and DT modelling and finite state models, Ptolemy (II) [24] which supports many domains with a strong basis on DF and aims to be a testbed for multi-domain modelling, SystemC-AMS [102] as an extension of SystemC for system-level mixed signal modelling, and Modelica [32] which is an object-oriented declarative modelling language. We will discuss the major tools in more detail next.

MATLAB/Simulink Simulink [99] is a graphical environment for dynamic and embedded systems, using block-diagrams for modelling and simulation for functional analysis. It is often used together with MATLAB, an imperative language for numerical computing. Simulink is the de facto standard for mixed CT/DT system modelling [17]. It is, however, limited in its support for multiple domains, only supporting DT as piecewise CT and Stateflow for finite state machines.

Simulink has support for hierarchical models and allows for code generation to C or VHDL. It supports many different numerical solvers. Models can be interpreted or compiled for simulation, and simulation is supported by different plotting scopes and showing the data type and sample time of the signals in the model. There are many specialised toolboxes, for application domains such as signal processing, providing implementations of common blocks.

Simulink is not very suitable for modelling digital hardware and software for multi-processor architectures, where architecture definition, reconfiguration and programming come into play. The reason for that is that the graphical interface does not offer much flexibility when you (structurally) want to change your design.

In the context of this thesis, however, the important drawback of Simulink is that it does not offer adequate support for the integration of the various notions of time. This either causes inadequacies in the simulation results or large inefficiencies during simulation. Furthermore, Simulink does not support DF models.

Ptolemy The Ptolemy project [24] studies design, modelling and simulation of concurrent, real-time, embedded systems. The project provides a framework for system simulation using diagrams and focuses on experimenting with various domains with the goal of researching their interaction. Models can be created using Java, XML or with a graphical tool. Ptolemy [24] supports many domains, including CT and DF and experimental DT support. The CT domain is defined as part of HyVisual, the hybrid system visual modeller which is built on top of Ptolemy.

Ptolemy uses an actor model and tagged signals for integration. Actors are concurrent computational entities that acts in response to messages. In the tagged signal model [58], signals are a collection of events and an event is a pair of a time and value. Ptolemy also has a notion of "super-dense time"; time is a real number with an index for ordering events that have the same time such as with discontinuities in a signal. Further, Ptolemy supports higher order components, but not higher order signals.

Since Ptolemy, just as Simulink, uses a global solver, the problems with the integration of time domains also are the same as with Simulink. Furthermore, both Simulink and Ptolemy offer little support for model transformations, apart from code generation.

SystemC-AMS SystemC is a set of C++ classes to provide discrete event simulation aimed at system-level modelling. It is used as a HDL, but also aimed at system-level modelling. However, systems are regarded from an implementation viewpoint, while the other tools are more from a requirement and specification viewpoint, i.e. SystemC aims at modelling at the architectural level in between the function level and the implementation level [35]. SystemC-AMS extends SystemC for mixed signal modelling [102], adding support for multi-domain signal flow models.

SystemC-AMS supports three modelling formalisms: timed dataflow, linear signal flow and electrical linear networks. Timed dataflow extends the un-timed DF domain by assuming discrete time steps between tokens, a similar approach to ours for including the DF domain. Linear signal flow is similar to models in Simulink, i.e. an equation system is abstracted from the model and a numerical solver is used for simulation (see section 4.3). Electrical linear networks use the same approach, however, now connections between components are bi-directional, i.e. it is an energy based model with effort and flow, such as voltage and current or force and velocity. Such effort and flow based models are also known as bond-graphs.

Again, since SystemC too uses a global solver, it has the same problems with the integration of time domains as Simulink and Ptolemy.

Modelica Modelica [32] specifies a object-oriented, declarative, multi-domain modelling language for modelling physical systems. Modelica has similar structure to our approach: components define relations and hierarchy, equations define functionality. Again models are described by differential equations. Components can be defined as uni-directional signal-flow blocks or bi-directional network components. Modelica is designed to be domain-neutral, but a large set of domain specific components are available in the standard library.

Modelica is only a modelling language definition; the simulation engine is unspecified. There are several implementations of the language such as Dymola or MapleSim. MapleSim [61] is built on top of the symbolic math engine of Maple. The ODEs are first simplified by using Maple's analytical algorithms before a numerical solver is used, resulting in faster simulations.

In the Modelica language there is no module predefined for time delays, and to the best of our knowledge all tools which implement Modelica use a global solver again and thus encounter the same problems as before. In addition, Modelica is not aiming at dataflow models.

Functional (reactive) programming The field of functional reactive programming (FRP) [26, 46] uses higher-order functions, a standard feature of functional languages, to model the CT and time-ordered discrete events for the SR (see section 4.2.4) domain. FRP has made excellent progress in being applied to different domains (for example animation [26], user interfaces [19] or robotics [75]) and providing formal semantics. The original work [26] focused on interactive animation with switching behaviours (animation) and events (interaction) [26]. In later work explicit behaviours and events are combined and only signal transformations are used [46, 113] in order to avoid space and time leaks (i.e. a backlog of remaining old data in the memory or large remaining computations that have not yet been evaluated because of laziness). FRP does not identify and use different notions of time; time is considered to be global and time is progressed globally by the FRP framework, depending on e.g. the processing load.

In [101] higher-order model transformations for parallelisation are introduced as strategies. Strategies complement an algorithm with a parallelisation approach.

ForSyDe [86, 87] is a system modelling and design refinement approach for embedded systems. ForSyDe aims at raising the modelling abstraction level; a specification is iteratively refined to implementation with (high level) model transformations. The first version of ForSyDe supports SR and DE models [86] (signals are tag-value pairs). Later versions also support automated hardware synthesis to an HDL, but to allow this ForSyDe has a so-called "shallow" embedded domain specific language (EDSL) variant for simulation and a "deep" EDSL variant for synthesis of models. For the "deep" variant the models also include their structure in the specifications, which an embedded compiler converts to an HDL. Inclusion of other domains such as the CT and DT domain in ForSyDe is ongoing work.

Acumen [96] is a language for hybrid systems inspired by FRP. Executable mathematics are used for modelling and simulation. The latest version supports (directed) signal flow equations and uses, as standard, a global notion of time and an ODE solver for simulations.

4.4.2 *Exact continuous time domain modelling*

All the researched tools and languages [17, 32, 102, 119] use ODE solvers for simulating the CT domain; an equation system is set up and a global time step is applied for numerical approximation, thereby implementing CT signals as a sequence of values. Time transformations such as a time delay therefore buffer values and interpolate between available values introducing inaccuracies caused by the modelling tool (section 4.3).

Tools represent signals as either a value at a global time step or a time-value pair. Although Ptolemy for example has a notion of super-dense time, this is still a time

(plus index) and value pair. Ptolemy and SystemC-AMS do offer some support for influencing the step size. For example, a block in a Ptolemy model can reject a step size of the ODE solver until all blocks agree. SystemC-AMS supports module and port time step propagation as a consistency check.

However, as we saw above, all of the above tools do not offer possibilities to adequately integrate time domains.

4.4.3 *Multi-domain modelling*

There are many tools supporting the CT and DT domains, often implementing DT signal as piece-wise CT signals [17] as discussed above. However, there are few tools that support CT, DT *and* DF; we only know of Ptolemy and to some extent SystemC-AMS (which uses timed dataflow). Together with ForSyDe, they are also the only tools that do not use fixed domains [24], i.e. in these tools the supported domains can be extended with additional domains by specifying their interaction with the existing domains.

There are several SR languages popular in embedded system design, such as Esterel [10], Lustre [37], Lucid [18] and Signal [52], because they have clear semantics and effective formal verification techniques [60]. However, none has support for more domains than SR because of this.

There are also tools supporting DF. For example, LabVIEW [65] has a graphical dataflow programming language, where signals are discetised streams of values. A simulation context is used to add time and a solver to the model. We have found no formalisation or implementation of dataflow that matches the semantics of signals and components as used in CT and DT signal flow diagrams.

4.4.4 *Mathematical definitions*

As modelling tools, to some extent, all of the above tools provide mathematical definitions, i.e. models represent some mathematically specified functionality. However, for many the mathematical equations are not readily identifiable in the model. For example, Simulink uses a graphical block diagram supported by the MATLAB language. MATLAB is a language for numerical computing, i.e. numerically approximating mathematics, but uses an imperative language for specifying this. Ptolemy relies on graphical models supported by Java, while SystemC relies on C++, both imperative languages. Only Modelica and FRP use declarative languages, which are close to specifying equations as discussed in section 4.1.4.

4.4.5 *Model transformation support*

A model-based design approach uses a single model for the (formal and functional) specification, verification, simulation and implementation of a design. This model is refined with model transformations from specification to implementation. The Object Management Group (OMG) provides a specification of the kind of models and diagrams used for model-based design as part of the SysML profile

of UML [70], and the abstraction levels as part of the model-driven-architecture (MDA) specification. How model transformations are performed is still actively researched. Current support for model transformations is typically limited to code-generation [42, 71].

Simulink and SystemC-AMS have no support for modelling transformations. Ptolemy only recently added initial support for model transformations using higher-order components [30]. Projects researching automated system level modelling with model transformations are Sesame [27] and Daedalus [69]. Their focus is on automatically parallelising applications, with the restriction that they consist of static affine nested loop programs (see below).

4.4.6 *Automatic parallelisation*

Part of the design process for embedded systems involves the partitioning of an application to divide processing over multiple computational resources. Partitioning of the application can also be performed as a model transformation, but it is more commonly known as automated parallelisation. The discussed tools are not normally concerned with matching the functionality with an architecture, other than the model transformations discussed above.

Simulink and Modelica have code-generation support, but this is a separate process and not part of the model. Ptolemy has preliminary code generation with template files with code blocks. On the other hand, SystemC-AMS models are already supposed to be specified on the architectural level.

A common approach for programming multi-processor architectures is to start with the digital processing part of the design as an application in C [39, 115]. An implementation in C translates the mathematical equations or expressions of the streaming application in a set of sequential statements. The statements update the state in memory word-at-a-time, creating a bottleneck to a central memory store [7]. A variable in an imperative language is mutable, i.e. it can be changed at any time. This makes it difficult to partition the program as it is hard to derive dependencies, i.e. what part of the program updates what memory at what time. Therefore, a common restriction on programs that are auto-parallelisable, is that they consist of static affine nested loop programs. Such programs consist of loops which communicate via arrays and in which each loop only updates values in its own scope at most once per execution [12], i.e. single assignment. From this restricted program a graph is extracted to create a dataflow model.

There are a number of projects taking the route of auto-parallelisation of sequential code. They have in common that they all use a Kahn process network (closely related to a dataflow model) to model the communication and synchronisation of the partitioned application, and the applications are limited to static affine nested loop programs. The Leiden Embedded Research Center (LERC) has a number of tools within the Daedalus project [69], trying to bridge the gap between system-level models and implementation. The applications are provided in C or MATLAB. A spin-off company, Compaan Design performs the auto-parallelisation by graph extraction and dependency analysis [94]. The Computer Systems Archi-

tecture group of the university of Amsterdam also uses a Kahn process network for system level design for the Sesame project. The implementation is based on C++ and XML. The Apple-Core project [4] researches the design and use of many-core architectures with support for light-weight threads and dataflow scheduling.

The problematic character of automated parallelisation is confirmed by the extensive research in this domain. Even for simple applications, it is already difficult to derive a dataflow model from a C application, because of unnecessary data dependencies and data dependent control [39].

4.5 Unified modelling based on time

We have found that current modelling tools do not adequately support multiple domains, model transformations, mathematical definitions and exact continuous time modelling in an integrated approach, yet these features are required for effective model-based design of embedded systems. Therefore, we propose a design flow and modelling and simulation framework that supports all these aspects, called UniTi. UniTi is a framework implemented in Haskell, a functional language, but first and foremost it is a modelling and simulation approach supported by an accompanying design flow.

In UniTi, components are (higher-order) functions that transform signals. The representation of signals is different in each supported domain. In the CT domain, signals are implemented as functions of time, in the DT domain signals are implemented as values, and in the DF domain signals are implemented as one or more tokens, just as presented in section 4.2. As in *all* domains components are signal transformations, UniTi uses unified composition operators for sequential, parallel and feedback composition of components. For integration, a CT function is evaluated at the sample time to determine the input value for the DT domain. A value from the DT domain is held constant for the CT domain until the next value, resulting in a piece-wise continuous signal. For DF integration, values from the DT domain become input tokens and output tokens become values.

UniTi is unique in being based on function composition instead of value-passing. Therefore, signal transformation in the CT domain are composed such that for example time delays are included in the final function, independent from the time used for simulation. This allows exact modelling of time transformations without loss of efficiency in simulation.

A formal description and detailed discussion of UniTi is presented in chapter 5. In this section we will relate UniTi to the desired characteristics discussed above, before discussing the accompanying design flow in section 4.6.

4.5.1 Model-based design

Following from the description above, UniTi supports the CT, DT and DF domains (which are arguably the most important domains for embedded systems) in a single model. We have found no formalisation or implementation of dataflow that

matches the semantics of signals and components as used in CT and DT signal flow diagrams, thereby allowing unified composition of mixed domain models as we provide with UniTi.

UniTi aims at raising the modelling abstraction level; a specification is iteratively refined to implementation with (higher-order) model transformations. It applies model transformations based on the mathematical properties, such as associativity, of the model components. This is similar to strategies as used in functional languages [101]. UniTi also uses model transformations for parallelisation. As the application is specified mathematically, this has the advantage that no dependencies are introduced other than intended in the specified algorithm. Therefore, it is easier to partition and parallelise.

Hardware synthesis for the DT and DF domain in UniTi is also possible with CλaSH [6], as UniTi components and the hardware components in CλaSH have the same structure. However, they are not integrated yet.

Model transformations are parameterised, thereby providing the designer a handle to try different alternatives, i.e. design space exploration. UniTi provides a functional evaluation during design space exploration. Of course, it is also very useful to evaluate other costs of the design such as the required computational and communication resources. A mapping function can determine the optimal solution for a given architecture, or the optimal architecture for the given performance figures according to some cost function. These costs can be included as meta-data for the components in the model. For now this is a manual process, but it would be a very useful extension to UniTi.

Overall, UniTi provides a single unified framework supporting model-based design, model transformations and design space exploration.

4.5.2 *Exact continuous time domain modelling*

In UniTi we differentiate between the representation of the model and the simulation of the model. A common view seems to be that (quoted from [119]): "[The] continuous evolution of variables is outside the domain of discourse of today's computers. Thus, while a denotational semantics for a hybrid systems language might embrace continuous evolution of the variable values, an operational semantics can only define values at discrete points in time." However, it is only necessary to define values at discrete points in time for simulation, *not* for representing CT signals. Using higher-order functions, CT components are defined as signal transformations on functions of time, which are composed into a final function *before* being evaluated. The evaluation of the model, which as mentioned before is a simulation of the specified system, is then performed at discrete points in time.

UniTi is the only tool that provides exact mixed-domain modelling using function composition. As higher-order functions are a standard feature of functional languages, FRP also uses higher order functions. However, FRP does not identify and use different notions of time nor does it support the CT, DT and DF domain and their different notions of signals.

4.5.3 *Mathematical definitions*

A functional language is close to mathematics, providing a nice fit to represent UniTi in. By representing the models mathematically one provides a formal specification which can be checked for correctness. Using mathematics for implementing functionality also means the inherent parallelism in the formalism is retained. Additionally, it allows for correctness preserving transformations and offers a unified abstraction mechanism to integrate CT, DT and DF modelling. Simulation of the design is done by straightforwardly evaluating the model. This saves us from the need to develop a specific solver (i.e. equation system and solution algorithm) that evaluates the model, as is standing practice in current tools. In fact, for a component that needs to be numerically approximated, such as integration, a solution algorithm is applied locally as part of the component. A different solution algorithm and approximation step can be chosen independently for each component.

UniTi relies on quite a few features of the functional language Haskell. A key feature exploited is the use of higher-order functions, used for model interactions and model transformations. More imporantly, we will show in section 5.1 that we can directly implement the formalisms of the CT, DT and DF domains in Haskell. As a consequence we can also use all of Haskell's tooling, such as the compiler and libraries, anda component in a UniTi model can use the full power of the Haskell programming language. In addition, the purity of Haskell restricts the programs so that they can be used safely on parallel or distributed systems such as MPSoCs. Finally, the type system is used to define interfaces for components, including all kinds of meta-data such as a visual representation of the component, model requirements, and cost figures and constraints.

4.6 Design flow

Using the domains defined in the previous sections, we will present a transformational model-based design flow to identify and guide typical steps encountered when designing embedded systems. Iterative, verifiable steps transform a single model into a division of functionality over the environment, the architecture (analogue and digital hardware) and the application (software), as well as a partitioning of the software over multiple cores. Although these steps described by themselves are not new, it is important to match them with the presented domains and with model transformations. A connection that is not trivial, as is evident from the lack of support for this in current tools.

Figure 4.7 illustrates the flow. The rounded rectangles represent models and the arrows represent transformations. A single multi-domain model includes the environment, the architecture and the application.

The design flow uses a top-down divide-and-conquer approach. The initial (formal) specification of a system is readily implemented and verified in the CT domain. We will discuss the co-design and partitioning steps; the mapping and code generation are beyond the scope of this thesis. Co-design can be seen as a division *over* the domains, while partitioning can be seen as a division *within* a domain (which we will limit to the DF domain in this thesis).

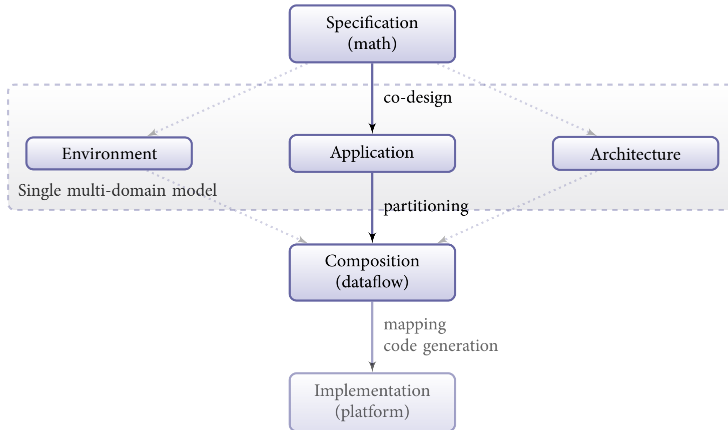


Figure 4.7: Design flow for tiled architectures

4.6.1 *Co-design*

During the co-design process, functionality is divided over the different domains. We distinguish a number of tasks:

- Decide what is needed from the environment for simulation and verification of the designed system. The environment is modelled in the CT domain.
- Define the architecture and decide what is implemented in analogue hardware (CT domain) and what in digital hardware (DT domain).
- Decide what to do in fixed hardware (ASIC, FPGA) and what to do in programmable hardware and software (DF domain), thereby refining the architecture and defining the application.

Co-design emphasises that the different perspectives in the domains are part of the system design and need to be included.

4.6.1.1 Analogue/Digital co-design

Analogue design uses continuous time mathematical models. Going to the digital domain involves sampling and quantisation by an ADC as well as choosing a representation such as fixed or floating point, and determining the required accuracy. Determining what to do in the analogue domain and what to do in the digital domain, i.e. where to place the ADC, involves taking into account implementation aspects in both domains as well as ADC limitations. It is often beneficial to move the ADC as far forward as possible, as in software-defined radio (SDR), because of the flexibility the digital domain brings. But especially in embedded systems, it is not possible to totally replace the analogue hardware by digital hardware. Consider for example mobile phone designs, in which the frequencies are too high to be able to implement everything in digital hardware.

4.6.1.2 Hardware/Software co-design

Processing systems often have a trade-off between what to do in hardware and what in software. Hardware refers to specific functionality with limited flexibility, but high efficiency (area, power, performance, cost), while software refers to processing on some kind of processor which can be programmed and is therefore much more flexible, but at the cost of efficiency. Hardware/software co-design refers to designing the hardware and software together in co-operation, thereby defining an architecture, and mapping functionality to hardware and software for this architecture. This involves balancing a trade-off between flexibility and efficiency.

4.6.2 Partitioning

After functionality is assigned to hardware or software, the software is partitioned over the programmable hardware (cores) in case of a multi-core architecture. The performance and efficiency of the software is determined by computation and communication costs. The computation is the actual work to be done, while the communication ensures the data is available at the right place and time. Having the data close to the computation, increases the efficiency by lowering the communication costs, exploiting so-called "locality of reference". As partitioning separates the computation, it introduces extra communication and has an influence on the performance.

We use a dataflow model where the processes contain the functionality and the communication is made explicit via channels. Such a model thus represents a partitioning of the software and transforming the dataflow graph changes the partitioning. Execution and channel content can be monitored with UniTi.

4.6.3 Example

As an example of applying the design flow, we design a low-pass filter for a CT source using the presented design flow. We choose to use a FIR filter in the DT domain, because this allows us to easily change the filter coefficients.

The specification consists of the requirements and a mathematical specification of the functionality. The filter requirements are a 20dB attenuation low pass filter with a 5MHz bandwidth. This results in a set of coefficients for the FIR filter, of which the specifics are not relevant for this example. The FIR filter is mathematically defined as:

$$y[t] = (h * x)[t] = \sum_{n=1}^{N} h_n \cdot x[t-n] \tag{4.1}$$

where N defines the filter order, h is the set of coefficients, x denotes the input data and y denotes the filter response.

As the FIR filter uses a DT signal ($x[t]$), an ADC is added before the filter:

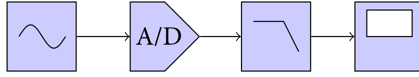


FIGURE 4.8: Filter block diagram

The environment generates the source signal. For the architecture, assume a 3-core MPSoC with an ADC. The application consists of the FIR filter. To fit the application to the architecture, the application is partitioned and state is introduced to limit communication, as we will discuss next.

A direct implementation of the filter of equation (4.1) on a single core would require read operations on the input data $x_{t-1,t-2,\ldots,t-N}$ (in memory) for output value y_t. For the next output value y_{t+1}, the values $x_{t,t-1,\ldots,t-N+1}$ are read. Hence, there is an overlap in read operations on $x_{t-1,t-2,\ldots,t-N}$. By storing these values in a local state, only the new value x_t needs to be read. So, the introduction of state significantly reduces the communication bandwidth due to *locality of reference* [23].

Consider the FIR filter from equation (4.1). When introducing state s_t (a vector of state values at time t), this equation can be rewritten as follows:

$$\begin{aligned} s_t[0] &= x[t] \\ s_t[i] &= s_{t-1}[i-1], \quad \text{where} \quad i = 1 \ldots N-1 \\ y[t] &= \sum_{n=1}^{N} h_n \cdot s_t[n] \end{aligned} \tag{4.2}$$

where the recurrent relation between s_t and s_{t-1} (shown at the second line in equation (4.2)) can be implemented very efficiently by a shift register. Although the state derivation seems to be trivial when done manually, an automated approach is much harder. This requires an advanced analysis of dependencies that influence the possibilities for partitioning, in order to obtain an efficient solution.

When state has been introduced, the communication bandwidth is reduced. However, the performance may still be too low for the execution on a single processor. Therefore, the operation is partitioned:

$$\begin{aligned} y[t] &= \sum_{n=1}^{N} h_n \cdot s_t[n] \\ &= \sum_{n=1}^{\frac{N}{2}} h_n \cdot s_t[n] + \sum_{n=\frac{N}{2}+1}^{N} h_n \cdot s_t[n] \end{aligned} \tag{4.3}$$

Note that the rightmost sum shown in equation (4.3) requires the state value $s_t\left[\frac{N}{2}\right]$. Using equation (4.2), we find $s_t\left[\frac{N}{2}+1\right] = s_{t-1}\left[\frac{N}{2}\right]$, which only existed in the leftmost sum shown in equation (4.3). Hence, if both sums are mapped on different

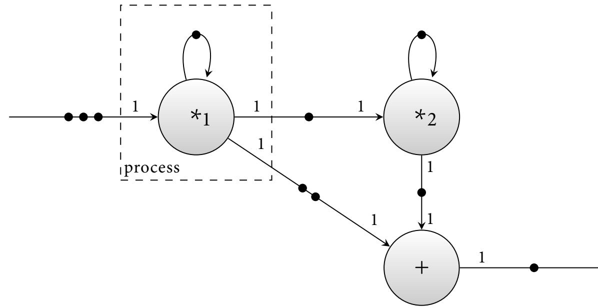


Figure 4.9: Dataflow model for the FIR filter application

processors, this intermediate state value has to be communicated. Also note that the equation parts are similar to the original FIR filter equation.

After partitioning the application, the resulting computations are assigned to processes in a dataflow model, as shown in figure 4.9.

4.7 Conclusion

In this chapter we have motivated the need for a model-based design approach that supports the CT, DT and DF domains. The CT domain is used for representing the environment and analogue hardware of an embedded system. The DT domain is used for the digital hardware, and the DF domain is used for representing software that is intended for a multi-core SoC with a NoC. Such a model-based design approach should also support mathematical definitions and model transformations. A model-based design process is often supported by formal specification, which forms the initial model. Using mathematically defined models allows a designer to directly define the equation of the specification in the model. Furthermore, mathematically defined models enable model transformations by exploiting the mathematical properties in the definition.

Thereafter, we have presented a novel unified perspective on time, signals and components in such models. This also includes the DF domain. In all domains, components represent signal transformations, i.e. they transform input signals to output signals. However, signals represent functions of time in the CT, values in DT domain, and one or more tokens in DF domain. This representation is deliberate because CT components are allowed to change the time reference, but DT and DF component should not be able to do this. Integration of CT components and DT components is achieved by sampling a CT signal or by holding a DT value for sample period. Integration of DT components and DF components is achieved by mapping samples to tokens and tokens to samples. For CT and DF components, the DT domain is used as an intermediate. Additionally, the integration of the DF domain defines time for tokens in a dataflow model, as the token arrival time is then defined by the sample time of the DT signal. Therefore execution time for

dataflow processes also has meaning and determines the (production) time of output tokens.

When simulating mixed-domain systems in current tools, time transformations, such as time delays, in the CT domain introduce artefacts. This occurs because such tools use a global solver which defines discrete global time steps to evaluate the system at. At each time step components pass values and components such as time delays must therefore be implemented by buffering values and using interpolation. This requires the designer to either accept less accurate simulations or reduce the simulation time step to improve accuracy at the cost of efficiency in simulation.

A survey of current modelling and simulation tools show that there are no tools that support *exact* CT domain modelling, few tools that support DF modelling and even fewer tools that also support mathematical definitions and model transformations. Therefore, in the next chapter UniTi is proposed. UniTi supports a unified perspective of the CT, DT and DF domains, and also provides unified composition of components in different domains. This allows exact CT modelling because time transformation are composed into a final function before simulation. UniTi is implemented in a functional language and therefore close to mathematics. As such, models can be directly evaluated for simulation, and model transformation based on the mathematical properties of components are supported.

Finally, these model transformations are used in a design flow for the design of embedded systems. This design flow uses a co-design step for a division of the model over the domains; a specification is transformed into a representation of the environment, the architecture (hardware) and an application (software). Next, a partitioning step performs a division within a domain, which we use for partitioning the application in the DF domain. Mapping and code-generation give a final implementation.

Chapter 5

UniTi

Abstract – *UniTi is a modelling and simulation framework for embedded systems supported by an accompanying design flow. It supports model-based design, model transformations, and mathematical definitions for multi-domain models in a single model, as follows from the analysis of the requirements and shortcomings of current tools in chapter 4. In this chapter we will present the formalism of UniTi. UniTi provides a novel unified perspective on time, signals and systems in the CT, DT and DF domains. As a consequence, it supports unified sequential, parallel and feedback composition of multi-domain systems. Signals in the CT domain are represented as functions of time, thereby enabling the accurate inclusion of time transformations (e.g. time delays) in the formalism, while signals in the DT domain are values and signals in the DF domain are lists of tokens. For integration, DF components are embedded in DT components, and DT components are embedded in CT components. Finally, model transformations based on the formalisms are used for the design steps of the design flow. This involves defining a mixed-domain model from the specification, and partitioning or parallelising the software.*

A design flow based on model-based design and supported by model transformations was presented in chapter 4. It results from the need for a design approach supporting modelling and simulation of multiple domains including the CT, DT and DF domain, mathematical definitions of modelling components, and model transformations, which current tools fail to deliver. This approach is called UniTi, emphasising that the unification is based on time.

In this chapter we will present the formalisation and the modelling and simulation framework of UniTi. The unified formalisation includes the CT, the DT and the DF domains such that components in these domains can all be specified in the same formalism as a signal flow graph. The CT domain is well established

Parts of this chapter have been published in [KCR:8], [KCR:9] and [KCR:10].

in engineering for modelling the environment or the analogue hardware [93]. An advantage of our approach is that there is no need to discretise the time of continuous signals for simulation purposes, thereby allowing exact time transformation such as (variable) time delays. The DT domain is used for representing the digital hardware of the system. For the software, especially signal processing applications and multi-core systems, DF models are very useful [56]. Processes in the dataflow model represent computations and channels represent explicit communications. As such, it can be used to partition an application. Furthermore, it offers methods to analyse and guarantee consistent real-time performance [39, 115]. The integration of DF in the same model therefore is a major advantage.

We will also define composition operators (for sequential, parallel, and feedback composition) which are valid for all three domains, leading to a flexible modelling of the system under design, as well as supporting model transformations and design space exploration. These composition operators allow for a block diagram like specification of the design.

UniTi is mathematical in nature, because the specification of an embedded system is usually given in (or at least supported by) a mathematical form (see section 4.1.4). Additionally, it allows for correctness preserving transformations and offers a unified abstraction mechanism to integrate CT, DT and DF modelling. UniTi is unique in being based on function composition instead of value-passing. Since functional languages are close to mathematics (in the sense that computations are represented by functions instead of statements), we express the framework and models in the functional language Haskell. Simulation of the design is done by straightforwardly evaluating the model. UniTi also includes a new execution model for the DF domain based on the representation of signals and components in UniTi.

Essential for model-based design is a single unified model and support for model transformations. Because of the integrated approach of UniTi, we can apply model-based design using transformation steps, thereby guaranteeing the correctness of the design. Guidelines are presented for transformations between and within domains.

This chapter is organised as follows. First, a formalisation of the domains is presented in section 5.1, a formalisation that also provides a solution for modelling time transformations exactly and integrating the DF domain. This is followed by an explanation of composition and integration of the domains in a single model in section 5.2 and section 5.3. Section 5.4 will elaborate on simulating models with the UniTi framework. Finally, we present the use of model transformations during the co-design and partitioning step of the design flow of UniTi, as well as guidelines for enabling such transformations, in section 5.5.

5.1 Formalisation of the domains

In this section we propose a novel way to simulate CT/DT systems which is exact while retaining efficiency. In the CT domain we consider signals as *functions of*

time such that the values of a signal can be exactly determined at every instant during the simulation. By implementing CT signals as *functions of time*, time delays or multi-rate systems can be implemented *exactly* without losing efficiency. In the DT domain we consider signals as piecewise horizontal from the last sample of the ADCs. Thus, our simulation technique coincides with the standard mathematical modelling of such systems. Additionally, in our approach time is kept local, i.e. every continuous component may have its own discretisation of time in time steps. Thus, components may be numerically calculated at a fine time scale without causing inefficiencies in those parts of the system which do not need such a fine time scale.

In order to deal with signals as functions of time, our approach uses *higher order functions* to express transformations of signal functions. Hence, we choose for a functional programming language (Haskell) to simulate a mixed-signal system.

In addition, the DF domain is presented in a way that is consistent with the representation of signals and components in the CT and DT domain (see also section 4.2). For this representation, DF signals represent token updates to channels, and DF components represent processes together with its input channels and firing rules. By generalising this notion of signals and systems we can integrate components in the DF domain with components in the CT and DT domain. As a consequence, we provide a novel unified perspective of time, signals, components and systems in the CT, DT and DF domains.

There are several execution models for the DF domain (e.g. concurrent processes, compilation of dataflow graphs, tagged token model) [57]. The most common is to implement dataflow processes as concurrent processes with static scheduling and implement the firing rules as a sequence of "read", "execute" and "write" phases [56, 57, 64, 82, 115]. However, these execution models do not match with a signals and components representation of dataflow, as they all use buffers or queues representing channels, i.e. the channel contents, while signals represent channel updates. We will present a new execution model for dataflow, following from representing dataflow models as DF signals and components.

Integration of the CT domain, the DT domain and the DF domain in one design framework is a problem that is not satisfactorily solved by existing tools (see section 4.4). In this section we also present an approach which supports the design process on these aspects, filling in a gap that is left by current design tools (such as [17, 24, 102]) which are able to solve this challenge only partially. We present a unified formalisation of the CT, the DT and the DF domains such that components in these domains can all be specified in the same formalism. There are few tools that integrate the DF domain with the CT and DT domain. Furthermore, these tools and the many tools that offer mixed CT and DT modelling (without DF) have problems with time transformations.

In the context of this thesis, in which we limit ourselves to streaming applications running on tiled multi-core architectures, the CT and DT domains are mainly relevant for the hardware side of an embedded system, and the DF domain deals with the software side.

5.1.1 Continuous time

Consider the following system consisting of a sine source, a time delay and a scope:

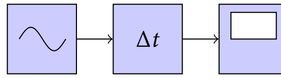


Figure 5.1: CT delayed sine wave block diagram

In the CT domain the physical environment of the system or the analogue hardware is represented. In this domain, time is represented by the real numbers, and a signal is represented by a *function* over *all* time. Thus, if f is a signal, then $f(t)$ is the value of that signal at time t. This leads to the following type definitions:

$$\begin{aligned} Time &= \mathbb{R} \\ Sig_{CT} &= Time \rightarrow \mathbb{R} \\ Component_{CT} &= Sig_{CT} \rightarrow Sig_{CT} \end{aligned}$$

where $A \rightarrow B$ denotes the class of all functions from A to B. Note $Component_{CT}$ is a "higher order type", i.e. it has a function (of type Sig_{CT}) as argument and delivers another function (also of type Sig_{CT}) as result, thus expressing that a component transforms signals.

A component can have multiple inputs and outputs. Multiple inputs and outputs are denoted as tuples (nested ordered pairs). The type of a component with n input signals and m output signals follows as:

$$Component_{CT} = Sig^n_{CT} \rightarrow Sig^m_{CT}$$

The first component in figure 5.1 is a sine *source* and as such does not really "transform" a signal. That is, it transforms a vacuous input:

$$source\ () = t \mapsto a \cdot sin\,(\omega t)$$

where a is the amplitude, ω the frequency and t is time. The notation $t \mapsto ..t..$ denotes the function which maps t to $..t..$.

The next component in figure 5.1 is a time delay. The delay can have any value and can even be variable. Existing modelling tools have problems with a time delay because the time of CT signals is discretised for simulation. Mathematically, however, a time delay component is simply defined as (where $\delta = \Delta t$):

$$delay_\delta\ (f) = t \mapsto f\,(t - \delta) \tag{5.1}$$

where f is a signal, i.e. a function of time, and $delay$ adjusts the time of f with δ. Thus, to find the function value on time t *after* a 0.1 time delay, one has to know the function value at time $t - 0.1$. As the input signal of $delay$ is a function of time, the time is delayed before f is applied to it. Thus we can locally control or change the

time reference of a signal. Note that $delay_\delta$ indeed is of type $Component_{CT}$. From the perspective of a CT component the input signal as a whole over all time is transformed, there is no discretisation of time needed for implementation purposes. In combination with the fact that the continuity of time is immediately present in the type definitions as well, this makes it possible to express the fact that a component can locally control or change the time reference of a signal.

The final component in figure 5.1 is a scope *sink*. The scope plots the signal, hence, as a transformation it may be rather meaningless. That is, the input signal is transformed to a vacuous output, with a plot of the signal as a side-effect:

$$sink\ (f) = t \mapsto ()$$

Now suppose a 4 Hz sine with amplitude 3 and a 0.1 time delay. The input signal of the sink, that is plotted, is then:

$$delay_{0.1}\ (source\ ()) = delay_{0.1}\ (t \mapsto 3\ \sin\ (2\pi 4t)) = t \mapsto 3\ \sin\ (2\pi 4\ (t - 0.1))$$

The time delay is accurately included in the final function, independent from the time used for simulation. In Simulink and other existing tools it is exactly at this point that inaccuracies are introduced.

5.1.2 *Discrete time*

We extend the system with an ADC and a signal bias in the DT domain:

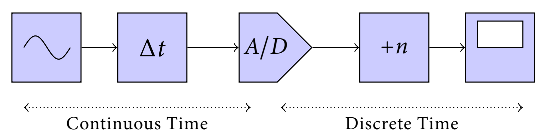


FIGURE 5.2: Mixed CT/DT system block diagram

In the DT domain the digital hardware (such as a FIR filter) of a system is represented, in which signals constitute the value of the signal at discrete moments in time. When an ADC is used to sample a CT signal, these values are also called samples.

This leads to the following type definitions:

$$Sig_{DT} = \mathbb{R}$$
$$Component_{DT} = Sig_{DT}^n \to Sig_{DT}^m$$

It is important to note that from the perspective of a component a signal now is a *single* value (numerical; here we assume $\mathbb{R}$), i.e. it is a value at a certain time t as in the CT domain but we have no control over t. Thus, from the perspective of the component, time is abstracted away. This representation is deliberate as a DT component should not have control over time (also see section 4.2), as the

digital hardware it represents does not either (except for sample delays, which are represented as state). Note that the type of components has the same structure as in the CT domain.

A component produces output values dependent on the current input sample and possibly on previous inputs. In order to express the influence of the history of the processing, a component has an internal state which keeps track of the relevant history. Looking at a component as a signal transforming (mathematical) *function* this means that the state has to be modelled as an additional argument to (and result of) that function. However, it is possible to hide the state, here only explained briefly, by directly feeding back the output state leaving only the input signal to be applied to the function. We will discuss this in more detail in section 5.4.3. Hence, a component can still be seen as a signal transforming function.

Returning to figure 5.2, the ADC component transforms the signal f into a discretised signal with time interval d. This is achieved by flooring the time of the CT input to the latest sample time:

$$adc_d\ (f) = t \mapsto f\,(\lfloor t/d \rfloor \cdot d) \tag{5.2}$$

Applying the resulting function to a (local) time t gives the latest value that was sampled before t. The output of the adc (which is still a CT function) is then a piecewise horizontal-function. This is implemented efficiently by re-using the results from the latest sample in between sample times.

The next component in figure 5.2 *is* a DT component and adds a constant n to a signal x (as in a bias, or a level shifter):

$$add_n\ (x) = n + x \tag{5.3}$$

Note that x is a numerical value, in contrast to f (from the $delay_\delta$ specification in the CT) which is a function of time. Therefore, there is no time in the definition of add. From the perspective of the DT component, the input value x is just a value at some moment in time; a time-varying value, but that is outside the influence of the component itself.

For a mixed domain model, components from both the CT and the DT domain have to be connected. These components use different types of signals. The output of the ADC gives the latest sample for time t and is a function of time. The add_n component must therefore be changed so it accepts functions of time as input, in order to connect them:

$$\widehat{add}_n\ (f) = t \mapsto n + f\,(t)$$

The notation $\frown$ is called “lifting” from a function on values to a function on functions. Lifting changes or embeds the DT component add_n to a CT component $\widehat{add}_n$ in order to combine the component with the CT adc component. However, as the lifting is performed on the original definition add_n, it still has no access to time t, i.e in the lifted version n is added to the value of input signal f at time t regardless of t.

Now suppose a delay of 0.3 time units, a sample period of 0.3 time units and an addition of 1. Then the result of (the interesting part of) the signal flow diagram in figure 5.2 is:

$$add_1\left(adc_{0.3}\left(delay_{0.2}\left(source\left(\right)\right)\right)\right) = t \mapsto 1 + sin\left(2\pi\lfloor\left(t - 0.2\right)/0.3\rfloor \cdot 0.3\right)$$

Note that the time delays are accurately included in the final function. Thus, the final function combines CT components and DT components in a single expression.

5.1.3 *Dataflow*

In the DF domain the software of the system is represented. The DF domain provides a model for stream processing with explicit communication.

In this section we will represent dataflow models as a DF component with input signals and output signals, i.e. the DF domain is represented in the same formalism as the CT and DT domain. This enables the integration of DF models with CT and DT models. Therefore, the environment, the hardware and the software of a system can be represented and simulated in a single unified model.

5.1.3.1 Processes and channels

As explained in section 4.1.3, a dataflow model or dataflow process network is a cluster of several independent *processes* that perform computations, and communication between these processes is made explicit via *channels*. A channel is an unbounded FIFO *token* container, where tokens are atomic data elements. Processes consume and produce tokens by reading from and writing to channels.

The amount of tokens consumed and produced, the rates, can be variable. A single-rate dataflow (SRDF) model always consumes and produces a single token, a multi-rate dataflow (MRDF) model has a fixed token rate at each input and output. In a cyclo-static dataflow (CSDF) model, the token rates cycle through a number of phases with fixed token rates (possibly zero) at each phase. Variable-rate phased dataflow (VPDF) models have a limited form of data-dependent token rates[115], where the token rate is determined by a parameter from an input channel. Finally, dynamic dataflow (DDF) model have fully data-dependent token rates.

5.1.3.2 Components and signals

Above, in the CT and DT domain, a component transforms signals, and signals connect components. When applying that approach to the DF domain, a *component* thus corresponds to a node in a dataflow graph, and a *signal* is the data that a component sends (and which consists of a sequence of tokens). This data then is received by another component which stores the tokens in its internal state. As soon as it has enough tokens it will execute, immediately followed by sending the produced tokens.

The above leads to the following type definitions, in which $Token$ is an abstract type to be defined for each application separately (the notation $[Token]$ denotes a list of $Token$s):

$$Sig_{DF} = [Token]$$
$$Component_{DF} = Sig_{DF}^{n} \rightarrow Sig_{DF}^{m}$$

Here too, the structure of a component is the same as before. Note that the list of tokens does not represent *all* tokens as in a channel, but only the currently communicated tokens, similar to the current value in the DT domain.

Therefore, the definition is different from the standard implementation of dataflow in which channels store current and previous (unconsumed) tokens and connect processes, while we use signals for connections (current tokens) and store (input) tokens in a component *with* the process. It might seem to the reader that it is easier to represent processes as components and channels as signals. However, this representation does not match well with the semantics of components and signals in the CT and DT domains. The reason is that signals do not have state (memory and state is represented using feedback as will be explained in section 5.2), but channels are token containers and as such *do* have state. Using signals with state is not a satisfactory option, because as explained in section 5.1.2 using a mathematical function requires state to be an explicit input and output, i.e. reading a token from a channel involves returning the token and the new state of the channel. Thus, a component reading from channels must also return the new state of all these channels as output. Furthermore, the process writing to this channel needs this updated state of the channel in order to output a once again updated channel state including produced tokens. Clearly, this representation of signals is more complex than just connecting two components.

A better representation of DF is to include the input channel(s) as state of the component. Signals then represent updates, in the form of tokens, to the channels (as channels are unbounded, new tokens can always be added to the channel). Components also implement firing rules, which are directly verified against the number of tokens available in the input channels. This matches well as firing conditions only change with channel updates. Furthermore, the component contains the current production and consumption rates and execution phase as state, needed for determining the firing rules.

Note that according to this definition of DF components, a component transforms a signal each time it receives an update, also in case it has not collected enough tokens to fire (or in case a process has an execution time that has not yet elapsed). To model that, it is possible that a component sends an empty signal containing zero tokens, i.e. a component, applied to an input signal, that does not fire results in an empty output signal. Although an empty signal does not indicate a change to the input channels, it does indicate an execution step so that components update the progressed time represented as execution steps. We will come back to this in the section about integrating the domains (section 5.3), because then execution steps are linked against "real" time in the CT and DT domains. The other way

around, i.e. a signal contains more tokens than a component needs in order to fire, is modelled by allowing a component to execute more than once (if possible), as standard in dataflow models, and to combine the produced tokens (in order) in a single result.

5.1.3.3 Definitions

Similar to the CT and DT domain, the user specifies the functionality of the component and the connections between them. UniTi takes care of managing channel contents, firing rules and execution.

Consider a DF component that consumes three tokens with a fixed rate, where the tokens are numbers, and calculates their average. The functionality of the process is denoted as:

$$mean_3\ ([x, y, z]) = [(x + y + z)\,/3]$$

where the input signal of $mean_3$ is a list of three tokens $[x, y, z]$, and the output signal is a list containing the averaged result as a single token. Clearly, this process can only execute when there are three values available and produces one value. It is thus a MRDF process with a token rate of 3 for the input and 1 for the output. Assume an execution time of the process of 1 execution step. Finally, a DF model typically has initial tokens in the channels. Suppose, initially there are two tokens (2 and 4) in the input channel.

The *average* component as a whole, i.e. its functionality together with its initial state, is now formulated as:

$$\begin{aligned} average &= \boxdot\, mean_3 \Uparrow S \\ &\quad \text{where } S = ((3,1,1)\,,[\,]\,,[2,4]) \end{aligned} \tag{5.4}$$

Herein, S is the initial state with the token rates and execution time as the first elements of the tuple, the currently processed tokens as the second element of the tuple (initially empty and explained further below), and the input channel contents as the third element of the tuple.

The $\boxdot$ operator and the $\Uparrow$ operator are implemented by the UniTi framework. The above is all that a designer needs to define when using the DF domain. The $\boxdot$ operator is required to add the management of channel contents and firing rules to $mean_3$, so as to embed the functionality defined in $mean_3$ in a DF component *average*. The $\Uparrow$ operator is required to provide the initial state of the component. These operators are discussed in more detail in the next section.

5.1.3.4 Definitions provided by UniTi

The complete structure of a DF component is illustrated in figure 5.3.

The functionality of a dataflow process is denoted by P. The operator $\boxdot$ extends the functionality of P with firing rules and execution. The $\boxdot$ operator takes care that when the component is applied to a signal i, it will:

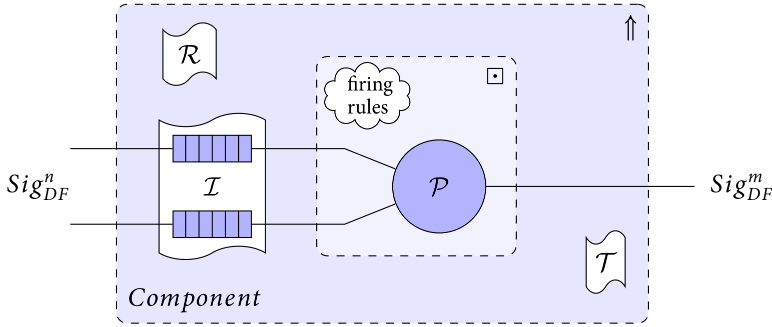


Figure 5.3: DF component structure

- add the tokens from i to its internal state,
- then apply the function P as many times as possible (possibly zero times), each time removing input tokens from the state,
- and finally packing the results in an output signal o.

The $\boxdot$ operator returns a function on state. The $\Uparrow$ operator applies this function to the initial state S following it. The state (with type $\mathcal{S}$) of a DF component is a 3-tuple:

$$\mathcal{S} = (\mathcal{R}, \mathcal{T}, \mathcal{I})$$

The first element of the state is a data structure for the token rates (of type $\mathcal{R}$):

$$R = (r_i^n, r_o^m, t)$$

where r_i^n are the token rates for n input channels, r_o^m the token rates for m outputs, and t the execution time, corresponding to $(3,1,1)$ in the example above.

When a process fires, tokens are consumed from the input channels. However, the output is typically not produced instantaneously, i.e. the execution time of a process is also modelled. This requires that the produced outputs are remembered until the execution time has passed. In addition, there can be multiple outstanding outputs as a process can fire as long as there are enough input tokens. Therefore, the second state element (of type $\mathcal{T}$) is a list of "timers" for storing outputs as state until their execution time has passed (as pairs of $Time$ and Sig^m_{DF}):

$$\mathcal{T} = [(Time, Sig^m_{DF})]$$

and is initially an empty list.

The final state element (of type $\mathcal{I}$) is the content of the input channels:

$$\mathcal{I} = Sig^n_{DF}$$

which is initially $[2,4]$ in the example.

The functionality of a DF process P of type $\mathcal{P}$ is a function from inputs to outputs:

$$\mathcal{P} = \mathcal{I}^n \rightarrow \mathcal{O}^m$$

Note that the input and output of P are of the same type as the inputs and outputs of the DF component:

$$Component_{DF} = Sig_{DF}^n \rightarrow Sig_{DF}^m$$

However, for the component the signals are channel *updates*, while for the process P they are the r_i input tokens and r_o output tokens of the process.

The $\boxdot$ operator converts $\mathcal{P}$ to a function on input signals and state. So the type of $\boxdot$ is:

$$\boxdot : \mathcal{P} \rightarrow (\mathcal{S} \times Sig_{DF}^n \rightarrow Sig_{DF}^m \times \mathcal{S})$$

Herein, the resulting function has a current state (S of type $\mathcal{S}$) and input channel updates (i of type Sig_{DF}^n) as inputs, and returns the output channel updates (o) and the updated state (S') as outputs.

After applying the above function to an initial state and an input i, it returns output o and the next state S'. The function is then already applied to next state S' using so-called *partial application* while the next input will follow later. The result is then an output o and a new function to use for the next input i. By repeating this each time, the state is not visible from the outside but hidden in the next function to use. This is performed by the $\Uparrow$ operator:

$$\Uparrow : (\mathcal{S} \times Sig_{DF}^n \rightarrow Sig_{DF}^m \times \mathcal{S}) \times \mathcal{S} \rightarrow Component_{DF}$$

Thus, the $\Uparrow$ operator applied to the result from the $\boxdot$ operator applied to P, and the initial state S result in a DF component $Component_{DF}$.

The state of a DF component contains all the information needed for execution. At execution:

- new tokens (from the signal) are added to the input channels of the component (with $+\!\!+$),
- the component is recursively executed (exe) as long as firing conditions are valid,
- and the timer structure T is checked if any execution times have passed resulting in output tokens o by the function $timer$.

These tasks are defined by the $\boxdot$ operator:

$$\begin{aligned}
\boxdot\ (P) = (S, i) \mapsto (o, S') \\
\text{where} \\
(R, T, I) &= S \\
I' &= I + i \\
(T', R', I'') &= exe\ (P, (T, R, I')) \\
(o, T'') &= timer\ (T') \\
S' &= (R', T'', I'')
\end{aligned}$$

where $'$ indicates an updated structure. The first definition in the "where" clause unpacks the state S into its three elements. The next three definitions correspond to the three steps presented above. The first step updates I by adding new tokens i to it and returning the updated input channel contents as I'. The next two definitions will be discussed in more detail below. The last definition packs the updated elements of the state together in S'.

The exe function executes a process P if enabled by the firing rule. Therefore, it uses the rate structure R to determine how many tokens are needed for firing. The current input channel content I' is checked to determine if enough tokens are available. If a process can fire the output tokens are added to the timer structure together with the execution time. The definition of exe is as follows:

$$exe\ (P, (T, R, I)) = \begin{cases} exe\ (P, (T', R', I')) & \text{, if } fire \\ (T, R, I) & \text{, otherwise} \end{cases}$$

$$\begin{aligned}
\text{where} \\
fire &= check\ (r_i^n, I) \\
(r_i^n, r_o^m, t) &= R \\
(i, I') &= read\ (r_i^n, I) \\
T' &= T + (t, P\ (i)) \\
R' &= (r_i^n, r_o^m, t)
\end{aligned}$$

Herein, $fire$ is a boolean that indicates if the process can fire. If not, T, R and I are returned as is, i.e. nothing changes. Otherwise, the input and output token rates and execution time are extracted from R. The input token rate r_i is used to read that many tokens from the input channels I. The process P is applied to the resulting inputs i to compute the output tokens. The outputs are added to the timer structure T together with execution time t. The updated state (T', R', I') is used for the recursive definition of exe, so the exe is repeated until the firing rule is false.

In the definition of exe, $check$ determines if a process can fire by comparing the input channel contents with the input token rate required for firing:

$$check\ (r_i^n, I) = \begin{cases} True & \text{, if } |I| > r_i^n \\ False & \text{, otherwise} \end{cases}$$

where $|I|$ is the number of tokens in the channel.

Finally, the function $timer$ updates the timer structure T by decreasing the execution time of each time-value pair with one and returns all output tokens of which the execution times have passed (in order):

$$timer\ ((t, v) : T) = \begin{cases} ([v], [\,]) : timer(T) & \text{, if } (t - 1 \leq 0) \\ ([\,], [(t-1, v)]) : timer(T) & \text{, otherwise} \end{cases}$$

Herein $[\,]$ denotes an empty list, and : is the list constructor operator. Note that this is a recursive function for which the output tokens whose execution time have passed (v) are collected in a list as the first argument of the tuple, and for the rest the execution time is decreased by 1 ($t - 1, v$).

5.1.3.5 Generalisation

There are several classes of dataflow models. The $mean_3$ example was a MRDF process. In fact we have generalised the rate structure R to all dataflow classes except DDF by noting that:

$$SRDF \subset MRDF \subset CSDF \subset VPDF$$

The most general rate structure, the one for VPDF[1], is a 5-tuple:

$$R = \left(p, t^k, q^l, rate_i, rate_o\right)$$

with the current execution phase p, a list of execution times per phase (t^k), parameters used to compute the current rate (q^l), and two rate functions $rate_i$ and $rate_o$ which compute the input and output depending on p and q^l. For example, the token rate at the input for phase p and parameter q^l is:

$$r_i^n = rate_i\ \left(p, q^l\right)$$

Besides changing the rate structure, the function exe must also be changed accordingly as follows. The updated rate structure is now defined as:

$$R' = \left((p+1)\ mod\,|t^k|, t^k, q^{l'}, rate_i, rate_o\right)$$

where $|t^k|$ denotes the number of elements of t^k, and mod is the modulo operation. Thus, each execution or firing, the phase is increased by one modulo the number of phases.

[1] for more about VPDF see [115]

We have defined functions for each of the different classes of dataflow to generate an R structure. SRDF, for example, only has an execution time; the token rates are always 1 and it has no phases or rate parameters. These functions also initialise T to an empty list and initialise I with initial tokens i^n. The functions return a state structure S and can therefore be used directly following the $\Uparrow$ operator. The generation functions are defined as:

$$\begin{aligned} sr\ (t, i^n) &= ((0, [t], (), _ \mapsto 1^n, _ \mapsto 1^m), [], i^n) \\ mr\ ((t, r_i^n, r_o^m), i^n) &= ((0, [t], (), _ \mapsto r_i^n, _ \mapsto r_o^m), [], i^n) \\ cs\ \left(\left(t^k, r_i^{n\times k}, r_o^{m\times k}\right), i^n\right) &= \left(\left(0, t^k, (), (p, _) \mapsto r_i^{n\times k}!!p\right), [], i^n\right) \\ vp\ \left(\left(t^k, q^l, r_i, r_o\right), i^n\right) &= \left(\left(0, t^k, q^l, r_i, r_o\right), [], i^n\right) \end{aligned}$$

where sr creates a rate structure and state for SRDF, mr for MRDF, cs for CSDF and vp for VPDF.

5.1.4 Representation in Haskell

The mathematical formulations of the CT, DT and DF domain can be straightforwardly represented in a functional language. In particular the support for higher-order functions to express signal transformations, and partial application (a function is already applied to part of the arguments, while the rest follows later), that functional languages offer, are essential. Furthermore, side-effects are not allowed in any of the domains, so we choose for the pure (side-effect-free) functional language Haskell. Haskell also provides a *type class* feature to conveniently overload algebraic and composition operators, so that the same operator can be used in all domains. That means that the type of the signal determines the specific operator implementation that is used.

We will first present the representation of the $delay$ component as presented for the CT domain, and the adc and add components as presented for the DT domain. The $delay$, adc and add components are straightforward reformulations of equations (5.1) to (5.3) as can be readily checked (`\t ->` corresponds to $t \mapsto$ and the standard function `floor` returns the greatest integer not greater than `x`):

```
delay  delta  f  =  \t -> f (t-delta)
adc    d      f  =  \t -> f (floor (t/d) * d)
add    n      x  =  n + x
```

Note that these functions have two arguments, one of which was given in the form of a subscript in the mathematical formulation. In the Haskell formulation the first is given when used as a component, while the second (`f` and `x`) represent the input signals which follow when a component is composed with another component.

For implementing the DF domain, the `df` operator is the Haskell formulation of the $\boxdot$ operator and `^^^` is the formulation of the $\Uparrow$ operator. The DF component $average$ (equation (5.4)) is then implemented as:

```
average = df mean_3 ^^^ [2,4]
```

The definition of `df` is exactly the same as ⊡, and `^^^` as ⥣, i.e. all definitions are immediately readable as Haskell code.

Of interest to the representation of the DF domain is the use of *type classes* by the framework. Type classes provide a polymorphic interface, so that the implementation is generic with respect to the number of inputs and outputs used by a DF process, i.e. one may overload the same operation symbol for different types by making some type `a` an *instance* of a type class and defining the operations of the type class for type `a`. All the provided definitions of the DF domain are polymorphic, except for using channels ($+\!\!+$, $check$, $read$) and generating token-rate data-structures (sr, mr, cs, vp), as these are directly dependent on the number of inputs. The inputs of a dataflow process can have different types, therefore, they are implemented as tuples because Haskell does not directly support heterogeneous lists (lists are homogeneous). However, a tuple is a type and Haskell has strong typing. Hence, an implementation must be provided for every combination of n-tuple input and m-tuple output. We have implemented these functions for up to four inputs and outputs (adding more is straightforward).

5.2 Composition

The standard mathematical interpretation of a signal flow diagrams is that the arrows express *signals* and the components denote *signal transformations*. Furthermore, the diagram as a whole then is a *composition* of these transformations.

The CT, DT and DF domains presented in section 5.1 all have components that transform input signals to output signals. This is intentional, so we can provide generic rules for composition in *all* domains. The generic component structure is defined as:

$$Component = Sig^n \rightarrow Sig^m$$

Note that Sig^n can be any combination of signals from the various domains and is implemented as a nested tuple, e.g. $(Sig_{CT}, (Sig_{DT}, Sig_{DF}))$.

Next, we will define operators for sequential, parallel and feedback composition. With these composition operators we can define arbitrary signal flow diagrams [93].

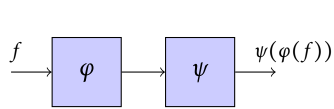


Figure 5.4: Sequential

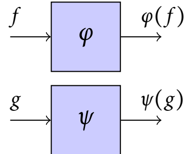


Figure 5.5: Parallel

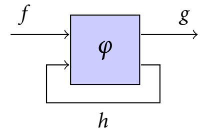


Figure 5.6: Feedback

5.2.1 *Sequential*

Sequential composition (illustrated in figure 5.4) combines components sequentially and is defined as:

$$\varphi \triangleright \psi = f \mapsto \psi\left(\varphi\left(f\right)\right) \tag{5.5}$$

where φ and ψ are transformations, i.e. components, and $\triangleright$ is the operation to compose transformations sequentially. That is, $\varphi \triangleright \psi$ is the transformation that takes a signal function f as an argument and determines its result by first applying φ to f and then ψ to the resulting signal function. Thus, $\triangleright$ returns a new component with the input signal f of φ and the output signal of ψ.

As an example, consider a sine source that is accelerating into the direction of an (stationary) observer which causes a change in the observed frequency; the Doppler effect. The resulting signal increases in frequency with time, which is modelled with time scaling in the CT domain as:

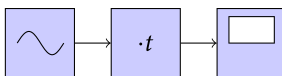


Figure 5.7: Accelerating source

and denoted as:

$$source \triangleright tscale_a \triangleright sink$$

The *source* and *sink* were defined in section 5.1 and are repeated here for clarity. The definition for $tscale_a$ is:

$$\begin{aligned} source\left(\right) &= t \mapsto sin\left(t\right) \\ tscale_a\left(f\right) &= t \mapsto f\left(t \cdot \left(a \cdot t\right)\right) \\ sink\left(f\right) &= t \mapsto \left(\right) \end{aligned}$$

As before, these definitions can be straightforwardly represented in Haskell, and evaluated for simulation. The simulation result is shown in figure 5.8 and as expected shows a frequency that increases over time.

It is sometimes useful to connect a single output of a component to multiple inputs of another components, thereby duplicating the signal. Since the number of inputs to connect to is known from the context of the sequential operator, we can define a generic definition:

$$\varphi \triangleright^* \psi = f \mapsto \psi\left(g, g, \ldots\right), \text{ where } g = \varphi(f)$$

Herein, φ is applied to the input signal f, and the resulting output signal g is used for as many input signals of ψ as needed.

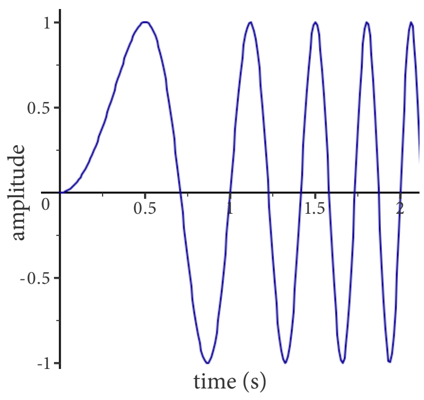


Figure 5.8: Chirp signal

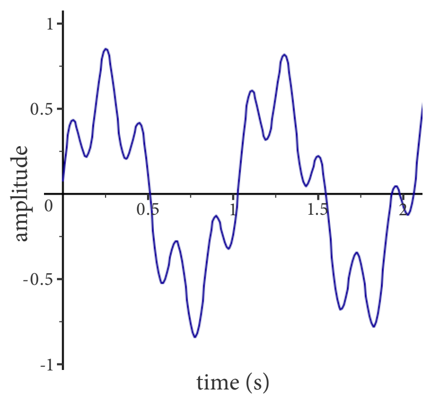


Figure 5.9: Two added sources

5.2.2 *Parallel*

Likewise, parallel composition (figure 5.5) is defined as:

$$\varphi \parallel \psi = (f, g) \mapsto (\varphi(f), \psi(g)) \tag{5.6}$$

i.e. multiple inputs are represented as tuples and the composition connects φ to the first and ψ to the second.

As an example, consider the following system:

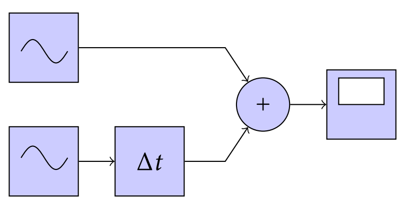


Figure 5.10: Simple beamformer block diagram

The system consists of two sources with different delays which are added, corresponding to a simple beamformer. This system is denoted as:

$$(source_1 \parallel (source_2 \triangleright delay_\delta)) \triangleright (+) \triangleright sink$$

The simulation results are shown in figure 5.9.

An alternative definition is:

$$\begin{aligned} system &= (\varphi_1 \parallel \varphi_2) \triangleright (+) \triangleright sink \\ &\text{where} \\ &\quad \varphi_1 = source_1 \\ &\quad \varphi_2 = source_2 \triangleright delay_\delta \end{aligned}$$

By using a `where` clause, we introduced hierarchy, i.e. in the definition of *system* we define two blocks to be in parallel (φ_1 and φ_2) and in the where-clause we define what these blocks are. So structural hierarchy is easily achieved by naming subsystems. This is possible because a composition of components is itself a component.

It is sometimes useful to connect a duplicate of a single component to each of the input signals in parallel. Since the number of inputs known from the context of the parallel operator, we can define a generic definition:

$$\|^{*}\ \varphi \ = \ \varphi \ \| \ \varphi \ \| \ \ldots$$

Herein, $\|^{*}$ creates as many duplicates of of φ (in parallel) as needed.

5.2.3 Feedback

In many systems there is a signal later in the system that is also used earlier in the system, i.e. there is a feedback loop in the system. Figure 5.6 shows a component φ with two inputs f and h and two outputs g and h. The signal h forms the feedback loop, i.e. the second output of φ is also used as input. From the outside, the resulting component only has an input signal f and an output signal g. However, g is determined by applying φ to f and h, where h is also determined by applying φ on f and h resulting in a recursive dependence. Thus, at some point φ must be able to determine h at the output using only f. Feedback composition (figure 5.6) is then defined as:

$$\circlearrowright \ \varphi = f \mapsto g, \quad \text{where} \quad (g, h) = \varphi \ (f, h) \tag{5.7}$$

Herein $\circlearrowright$ connects the second output of φ to its second input, thereby returning a component with input f and output g.

As an example we take an RC low-pass filter:

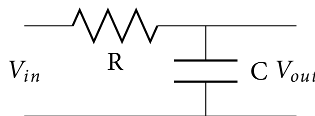


Figure 5.11: RC low-pass filter

Applying Kirchhoff's current law we get:

$$\frac{V_{in}(t) - V_{out}(t)}{R} = C \frac{\mathrm{d}V_{out}(t)}{\mathrm{d}t}$$

which we can rewrite to:

$$V_{out}(t) = \frac{1}{RC} \int_{-\infty}^{t} (V_{in}(t) - V_{out}(t)) \ \mathrm{d}t$$

and which corresponds to the following block diagram:

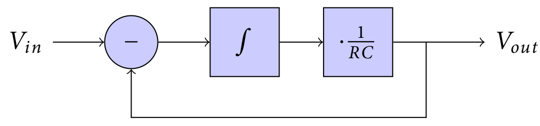


Figure 5.12: RC filter block diagram

This block diagram corresponds to a component with V_in as input signal and V_out as output signal. Furthermore, it has three sub-components ($-$, $\int$ and $\cdot 1/RC$) which are sequentially connected and of which the result is fed back to the input. Hence, it is defined in UniTi as:

$$filter_{RC} = \circlearrowleft \left((-) \; \triangleright \int \; \triangleright \left(\cdot\frac{1}{RC}\right)\right)$$

5.2.4 *Representation in Haskell*

The composition operators $\triangleright$, $\|$ and $\circlearrowleft$ of equations (5.5) to (5.7) are written in Haskell as `>>>`, `||` and `loop` respectively, and are also in direct correspondence to their mathematical definition:

```
phi >>> psi = \f      -> psi (phi f)
phi ||  psi = \(f, g) -> (phi f, psi g)
loop phi    = \f      -> let (g, h) = phi (f, h) in g
```

These definitions are exactly the same as in mathematics, except that the arguments `f`, `g` and `h` are not written between brackets as is standard in Haskell. Also in these representations there is a recursive dependence on `h` in `loop`. Therefore, the arguments (`h` in particular) of `phi` should not be evaluated before `phi` is applied to them, otherwise there is an evaluation of `h` that will never terminate. Haskell supports delaying the evaluation of an expression (such as the arguments of a function) until it is actually needed, called *lazy evaluation*, to allow the above definition of `loop`.

The representations of $\triangleright^*$ and $\|^*$, written in Haskell as `>>>*` and `||*`, are a little more involved. As mentioned, multiple input signals are represented as tuples in Haskell, as the input signals can have different types and lists must have a homogeneous type in Haskell. However, each n-tuple requires a different implementation for each n. Therefore, we have to define type classes for `>>>*` and `||*` and provide instances for each n-tuple.

The type class `CompSeq` for `>>>*` is defined as:

```
class CompSeq b c where
    (>>>*) :: (a -> b) -> (c -> d) -> (a -> d)
```

Here `c` represents an n-tuple of type `b`, where the number of elements of `c` determines which instance to use. The types `a` and `d` are free, i.e. they are not restricted by the type class. We must provide an instance for each n-tuple:

```
...
instance CompSeq (b) (b,b) where
    phi >>>* psi = \f -> let g = phi f in psi (g,g)
instance CompSeq (b) (b,b,b) where
    phi >>>* psi = \f -> let g = phi f in psi (g,g,g)
...
```

where the output signal `g` of component `phi` is used for each of the inputs of `psi`. The actual number of inputs of `psi` determines which instance is used.

Similarly, we define a type class `CompPar` for `||*`:

```
class CompPar a b c d where
    (||*)  :: (a -> b) -> (c -> d)
```

Here `c` represents an n-tuple of type `a`, and `d` represents an n-tuple of type `b`, and the number of elements of `c` and `d` determines which instance to use. Again, we must provide an instance for each n-tuple:

```
...
instance CompPar a b (a,a) (b,b) where
    (||*) psi = \(f,g) -> (psi f, psi g)
instance CompPar a b (a,a,a) (b,b,b) where
    (||*) psi = \(f,g,h) -> (psi f, psi g, psi h)
...
```

where the number of input signals determines which instance is used and thus how many copies of `psi` are created.

The requirement to specify an instance for each n-tuple is a Haskell restriction, not of the mathematical definitions. This forms a practical issue, as UniTi must now provide many instances of essentially the same concept. Of course the definitions are regular and can be generated, however, a generic definition over any size tuple would be preferred. Furthermore, it is sometimes necessary to explicitly provide the types of signals or components, if the type interference in Haskell can not derive the right types. Solutions to these problems are an ongoing discussion in the Haskell community and fall outside the scope of this thesis. For the use in this thesis, the provided definitions are sufficient.

The composition operators can be used to compose components of arbitrary domains, e.g. a CT component can be composed with DT and DF components etc. The composition operators are overloaded to support this, e.g. in case `phi` is a CT component and `psi` is a DT component, `psi` is automatically converted to or embedded in a CT component before being composed. The same holds for DF components. This is explained in detail in section 5.3.

5.2.5 *Algebra*

Normal algebraic functions operate on values, but signal transformations operate on signal functions. We can transform a normal function so that it operates on signals, called *lifting*. Lifting makes it possible to use the same operators such as $\sqrt{}$, $+$ and $*$ as components in *all* domains.

Unary operations, such as $\sqrt{}$ or $+1$, change the signal independent of the time:

$$\widehat{\sqrt{f}} = t \mapsto \sqrt{f(t)}$$

Implementation is straightforward:

```
sqrt f = \t -> sqrt (f t)
```

Binary operations such as + and · are similar; we evaluate both inputs at time t and then apply the operator:

$$f \widehat{+} g = t \mapsto f(t) + g(t)$$

As mentioned before, Haskell has type classes. We can use that abstraction mechanism to define e.g. arithmetic operators for numerical functions. The type class `Num` has some standard arithmetic operations, so we can use them to compose functionals:

```
instance Num (Time -> value) where
    f + g  =  \t -> (f t) + (g t)
    f - g  =  \t -> (f t) - (g t)
    f * g  =  \t -> (f t) * (g t)
    ...
```

where `value` is some type of the values.

For the DF domain, a unary operator is applied to each token of signal:

$$\widehat{\sqrt{\vec{x}}} = \langle \sqrt{x_1}, \sqrt{x_2}, \sqrt{x_3} \ldots \rangle$$

This can be seen as lifting an operation to operate on a vector, in our case the DF signal. As a vector can be seen as a function from indexes to values, this is in accordance with lifting an operation to operate on functions. The Haskell function that performs this is called `map`:

```
hatsqrt xs = map sqrt xs
```

Binary operations similarly perform their operation pairwise on the elements of the lists:

$$\vec{x} \widehat{+} \vec{y} = \langle x_1 + y_1, x_2 + y_2, x_3 + y_3 \ldots \rangle$$

In Haskell the function that performs this is called `zipWith`, which we use to define the `Num` type class for lists of tokens:

```
instance Num ([token]) where
    xs + ys = zipWith (+) xs ys
    xs * ys = zipWith (*) xs ys
    xs - ys = zipWith (-) xs ys
    ...
```

Note that these lifted operators only provide the functionality of a dataflow process. In order to define a DF component, we have to extend this functionality with firing rules and channel management using the $\boxdot$ operator, and with an initial state using the $\Uparrow$ operator, e.g.:

$$plus_{DF} = \boxdot\,(+) \Uparrow ([\,],[\,])$$

5.2.6 Calculus

Until now we have only discussed algebraic composition (we did not define the integral in the feedback example of section 5.2.3), for which the interesting simulation times are *at* the sample times of the ADC. Calculus is about change *over* time. For example, the voltage over the capacitor in figure 5.11 is proportional to the integration of the current through the capacitor until that time. We can solve the integral (or a differential) symbolically or numerically. A problem with symbolic integration is that for many functions an analytical solution does not exist. Thus, simulation tools solve the general case with numerical integration. Haskell has good possibilities to define analytical solutions to integral definitions whenever possible, but we will use numerical integration for generality, as in the standard approach in simulation tools.

The component for integration is defined as:

$$\int(f) = t \mapsto \int_{t_0}^{t} f(t)\ dt$$

Numerical integration relies on a recurrence relation to approximate the integral. A simple numerical integration method is the Euler method[2]:

$$y_{n+1} = y_n + h \cdot f(t_n), \quad \text{where} \quad h = t_{n+1} - t_n$$

So for multiple steps:

$$y_t = y_{t_0} + \sum_{0}^{n-1} h \cdot f(t_n), \quad \text{where} \quad n = (t - t_0)/h, \ t_{i+1} = t_i + h \tag{5.8}$$

Here, h is the approximation time step that is used locally. For this definition, t determines the number of steps n to compute from time t_0 using time step h. Furthermore, t_n is the time for step n to evaluate input signal f at.

Note that for each use of the integral component a different approximation step can be chosen by the designer. The accuracy of the approximation depends on the correspondence between the dynamics of the signal and the step size. As this can differ at different places in the system, it is very useful to be able to define the time resolution per integral (or differential). However, to the best of our knowledge, all simulation tools use a single implicit time step to update the whole system, therefore potentially unnecessarily calculating simulation results for much of the system. We conjecture that current tools use a global simulation time step, because it is difficult to determine the different time steps at each place in the system. However, with our approach, by locally applying the time steps, the time used for evaluation is only propagated back to the input signal, i.e. only the input signal and the components that determine that input signal are evaluated using the local time step. So, signals at the input are evaluated each approximation time step, but how often the result at the output and the following blocks is evaluated is not influenced.

[2] Of course we can also use more sophisticated numerical algorithms such as Runge-Kutta, which determines the time step based on the tolerance in accuracy of the result.

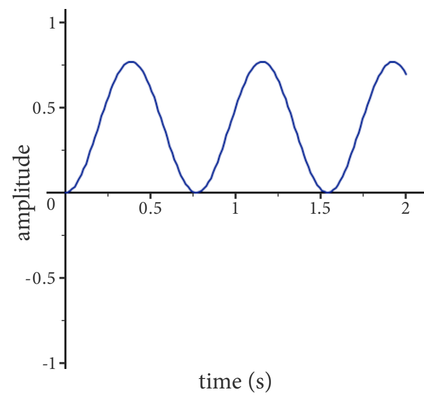


Figure 5.14: Integration of a sine wave

An example is the following system:

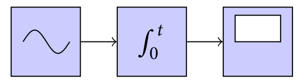


Figure 5.13: Integrator

which is implemented as:

$$source \rhd \int_{h,(t_0,y_0)} \rhd sink$$

with h the approximation step size, t_0 the initial time, and y_0 the initial value of the integral.

Following from equation (5.8), the input is calculated from time t_0 to t in steps of h. Each result is multiplied with h to get the approximate area and these results are summed and added to the initial value. The integration of a sine wave with 10 steps per sample period is shown in figure 5.14, illustrating its use. Note that we provide the sample time t to the integral function, which then itself determines how often to calculate the input signal f using the step size.

The integral definition includes a recurrent dependence on itself. This can also be represented with a feedback composition operator ($\circlearrowleft$):

$$\int = (\cdot h) \rhd \circlearrowleft ((+) \rhd^* (id \parallel delay_h))$$

where id is the identity function and $\rhd^*$ duplicates the input signal, which is delayed. The delayed signal is fed back to one of the inputs of the addition, i.e. one of the addition arguments is a delayed version of itself, which recurses back until the initial value.

In these implementations, the integral is recalculated from time 0 to t each simulation step. A more efficient implementation that re-uses previously calculated values needs state. State is discussed in section 5.4.3.

5.3 Integration of the domains

So far, we have discussed how to define components in the various domains and how to compose components within each domain. Now we will discuss how to compose mixed CT, DT and DF components for a multi-domain simulation. This is achieved by embedding a DF component in a DT component and a DT component in a CT component such that for simulation purposes the CT domain is the unifying domain.

Below we first describe how the designer can explicitly embed the DT domain in the CT domain, and the DF domain in the DT domain. Next we describe how the `Num` class (section 5.2.5) automatically embeds these domains. Finally we give an outline for a further automatisation of the embedding of the domains such that it also works for arbitrary operations.

5.3.1 *DT ⇒ CT*

To embed a DT component into a CT component it must accept a function of time instead of a single value (see section 5.1), i.e. we have to "lift" the DT component to a CT component. In section 5.1.2 we introduced the notation $\widehat{add_n}$ for the "lifted" version of add_n. Here we will generalise that notation into a *lifting operator* $\widehat{\ }$.

For unary functions the operator $\widehat{\ }$ is defined as follows:

$$\widehat{g}\ (f) = t \mapsto g\ (f\ (t))$$

whereas for binary operations (say h) it is defined as follows:

$$\widehat{h}\ (f, g) = t \mapsto h\ (f\ (t), g\ (t))$$

Clearly, this can be generalised immediately for n-ary functions. So, when a designer has a DT component C in his design, he can simply replace it by $\widehat{C}$ to get a CT component. However, in the chosen application domain the typical operations that are performed are numerical. Here, the `Num` class in fact already performs the lifting operation automatically (see section 5.2.5).

Example DT signal transformations are not applied for all time, but only at the sample times. The boundary between the CT and DT domain is the ADC, which samples the CT domain to provide that value to the DT domain. Thus from a CT perspective, the ADC floors the time to the latest sample time and holds that value until the next sample time (with d the sample period) as we have seen before:

$$adc_d\ (f) = t \mapsto f\ (\lfloor t/d \rfloor \cdot d)$$

So the composition of a CT component with a DT component is a CT component, but one that only evaluates and returns the result at the latest sample time for any time in between.

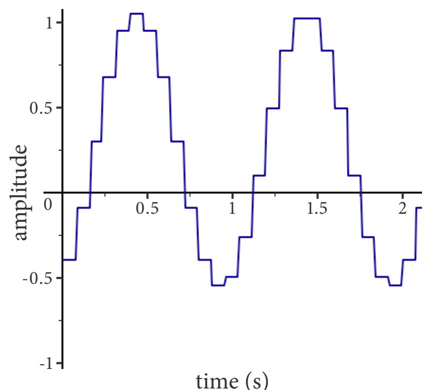


Figure 5.16: Delayed, sampled and biased

As an example, let us revisit the following mixed CT and DT domain system:

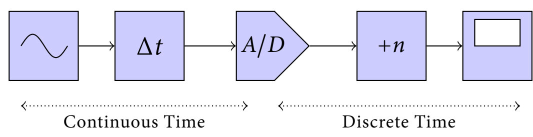


Figure 5.15: Mixed CT and DT domain system

which is defined as (with a delay of 0.15 time units, a sample period of 0.08 time units and an addition with 0.25):

$$source \triangleright delay_{0.15} \triangleright adc_{0.08} \triangleright add_{0.25} \triangleright sink$$

Note that in this example the necessary lifting is taken care of by the `Num` class. The simulation results are shown in figure 5.16, showing a delayed, sampled and biased sine wave as corresponding to the definition of the mixed-domain model.

In case a DT component is composed with a CT component there are two situations. If there is a CT component connected somewhere in front of that DT component, that composition causes the DT component to be lifted to a CT component. If there are only DT components in front of that component, the sample time of those DT signals must be provided. This is achieved by $rate_d$ with d the sample period, which is defined as:

$$rate_{d,(t_0,v_0)}\ (\varphi) = t \mapsto \begin{cases} (v_0,\ (t_0, v_0)) & \text{, if } (t < t_0 + d) \\ (v,\ (t_0 + d, v)) & \text{, otherwise} \end{cases}$$
$$\text{where}$$
$$v = \varphi\,()$$

Herein, $rate$ uses state to remember the last sample value v_0 to output until the time t is larger than the saved sample time t_0. If a next sample v is output, φ is evaluated, where $rate$ required φ to be a source component with a vacuous input.

Automatic lifting in general It falls outside the scope of this thesis to discuss automatic embedding of the DT domain in the CT domain in detail, for that too much knowledge of Haskell is needed (see [47, 62, 72, 73]). Besides, as mentioned above, it is not necessary for the chosen application domain. Nevertheless, we implemented this automatic embedding and give an outline of it below.

The implementation uses the Haskell type class abstraction mechanism. In particular, the type classes `Functor` and `Applicative` are used to express lifting, and the type class `Category` is used to define composition operators such that lifting is included.

For unary functions, the type `Time -> value` has to be turned into an instance of the type class `Functor` (`Time` is a synonym for the type `Double`, `value` stands for the relevant type of values). In that type class an operator *fmap* exists, which has to be redefined for the instance of the functor that we need:

```
instance Functor (Time -> value) where
    fmap g f = \t -> g (f t)
```

Hence, for unary functions, $\frown$ is represented by `fmap`.

For n-ary functions in general, the type `Time -> value` has to be turned into instance of the type class `Applicative`. The definitions of the appropriate operators in the type class `Applicative` then are:

```
instance Applicative (Time -> o) where
      <$> op = \_ -> op
    f <*> g  = \t -> (f t) (g t)
```

We define $\frown$ for unary, binary, ternary, ... operations as follows:

```
<^>     op f       =  <$> op <*> f
<^^>    op f g     =  <$> op <*> f <*> g
<^^^>   op f g h   =  <$> op <*> f <*> g <*> h
<^^^^>  op f g h k =  <$> op <*> f <*> g <*> h <*> k
```

Note that the unary version `<^>` coincides with `fmap` above.

Embedding is only required for sequential composition of mixed domain components. For parallel composition, different domains can coexist in parallel. For feedback composition, there is typically a dependence between the input feedback signal and the output feedback signal, causing the sequential composition within the feedback loop to lift the feedback signal to the same domain. Otherwise, the feedback composition does not really represent a loop and the definition will only be valid if the types match.

In order to indicate how composition operators can automatically perform the lifting operation, we restrict ourselves to composition for unary functions as an example. Thus, let φ be a CT component, and ψ a DT component. In order to define $\varphi \triangleright \psi$ in Haskell, we define a type class `Lift` with a composition operator `>>>`:

```
class Lift a b c where
    (>>>) :: a -> b -> c
```

As mentioned before, all domains ultimately (for simulation purposes) have to be embedded in the CT domain. Thus, we have to build instances of the `Lift` type

class for all combinations of the CT, DT and DF domain. We will present an example of a composition of a CT component with a unary DT component. In this example the type parameter `a` becomes `ComponentCT`, the type parameter `b` becomes `ComponentDT`, and `c` becomes `ComponentCT`. This leads to the following instance of the type class `Lift`:

```
instance Lift ComponentCT ComponentDT ComponentCT where
    phi >>> psi  =  phi >>> (<^> psi)
```

Thus the composition operator from the CT to the DT domain first applies `<^>` to the DT component, such that `<^> psi` now also is a CT domain component. For the composition operator >>> on the right hand side the correct version of the >>> has to be chosen. However, in order to give Haskell sufficient information such that it can decide which instance to choose, functional dependencies on types have to be used.

Above we described the basic idea how the type class `Lift` is used for defining a composition operator which integrates the various component types. As said, presenting the details of the full implementation falls outside the scope of this thesis.

5.3.2 *DF ⇒ DT*

DF signals are a list of tokens, while DT signals are values. Thus, to embed a DF component in a DT component, it must accept single values (samples) instead of a list of tokens. This presents a problem, because in DF models data is abstracted away into tokens, i.e. tokens are arbitrary data, while samples are values. We could also abstract from data in the CT and DT domains; CT signals would then be functions of time to tokens, but these have no sensible physical representation. So instead we limit embedded DF models to values as tokens at the boundaries. This is achieved by writing the value from the DT domain as a token into the input channel of the DF component at the sample time. So a value from a DT signal is converted to a DF signal consisting of a singleton list with that value as token, i.e. from the perspective of the dataflow model, the DT domain produces single tokens at a fixed rate. Vice-versa, a single token output DF signal is converted to a DT value (possibly with a delayed sample time because of the execution time).

Automatic lifting in general We shortly mention the possibilities to let the composition operators do the packing-unpacking (`<~>`) automatically:

```
<~> psi = \x -> let [y] = psi[x]  in  y
```

By means of example we give the definition of >>> for DT to DF:

```
instance Lift ComponentDT ComponentDF ComponentDT where
    phi >>> psi = phi >>> (<~>psi)
```

5.3.3 Unified model

Now components from all domains can be composed (by taking the DT domain as an intermediate step in case of a DF component). An example of a mixed domain system then follows as:

$$source \triangleright delay_{0.2} \triangleright adc_{0.05} \triangleright add_{0.15} \triangleright average \triangleright sink$$

where $source$, $delay_{0.2}$ and $adc_{0.05}$ are CT components, $add_{0.15}$ and $sink$ are DT components, $average$ is a DF component, and their composition $system$ is a CT component. The simulation results are shown in figure 5.17. The $average$ component averages three samples, as defined in section 5.1.3, and therefore only outputs a token every three input tokens or samples from the DT domain. In between no output is provided, which is plotted as zero. The plot function connects those point with lines, as shown.

5.3.4 Time

As said, the DF domain is untimed and only models the ordering of tokens. In the DT domain samples are linked to a sample time. So from the perspective of time, which is of primary importance in a simulation model, a DT model contains more information than a DF model and it makes sense to embed a DF model in a DT model. Because the DT values are linked to a sample time, the DF process now is extended with time. Therefore, execution time of a DF process also has meaning; the produced tokens after execution are considered values in the DT domain (with a delayed sample time because of the execution time).

For a simulation, we are interested in the behaviour of the system, i.e. the results of the model over time. Therefore, simulation is a CT process, i.e. we evaluate the model over time. As such, it makes sense to embed a DT model in a CT model for simulation. This is still efficient, because time is floored to the latest sample time (as with the ADC) and the sample value is re-used for every evaluation using state.

5.3.5 Multi-rate

Multi-rate systems contain DT domain samples that are generated by ADCs operating at different rates. Such systems are typically problematic for simulation because the data must be aligned with a global clock tick. Thus if we consider the samples of the ADC over time as a list, then in multi-rate systems the positions in the lists correspond with different times, which are difficult to merge. As we have separated these notions of time, this is not a problem in our approach.

Consider a multi-rate mixed-signal model with an ADC with rate 0.3 and an ADC with rate 0.35. A simulation at time 2, means that the ADCs should output the latest samples, which are the samples from time 1.8 and 1.75 respectively. Thus at the ADC we floor the simulation time to the last sample time and use that to

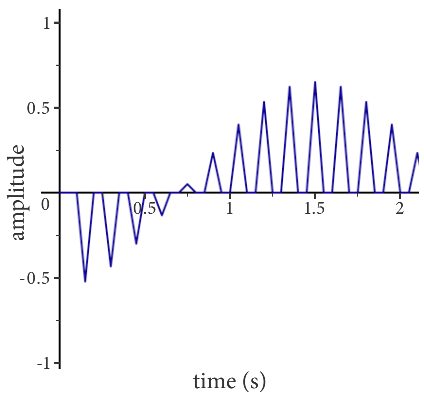


Figure 5.17: Averaged sine wave over three samples

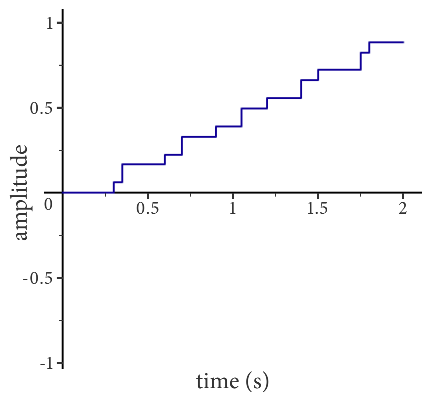


Figure 5.18: Addition of two DT ramp signals with different sample rates

evaluate the signal from the CT domain. This system is defined as follows:

$$(ramp_{0.2} \;\triangleright\; adc_{0.3} \;\|\; ramp_{0.3} \;\triangleright\; adc_{0.35}) \;\triangleright\; (+) \;\triangleright\; sink$$
$$ramp_r\,() = t \mapsto r \cdot t$$

The input signals of the ADCs are linearly increasing signals with a slope of 0.2 and 0.3 respectively. The different slopes of the input signals allow us to differentiate the two ADC signals more easily. The simulation result is shown in figure 5.18. The sample times of the two ADCs are clearly visible in the figure, including the latest samples at time 1.8 and 1.75. Furthermore, there is no common clock between the signals, making such a multi-rate system difficult to align with a global clock, especially if there are even more DT signals with a different rate.

Note that simulation steps could be larger than ADC sample time steps, so DT components with state should be applied with all the samples since the latest state. This is implemented by tracking the latest sample time and the local sample rate in the DT domain combinators. Also, blocks that combine data with different rates should take care of generating an appropriate rate at the output. This is only a designer issue, however, because the framework supports mixed rate signals as can be seen from the simulation results above.

5.4 Simulation

Simulation of a model consists of evaluating the model and visualising results. Defining the mathematical definitions of the model in Haskell provides the advantage that the model can be “executed” for simulation. Additionally it is useful to show a block diagram or dataflow graph of the systems and the state of components. We will discuss both evaluation and visualisation.

A major difference between our approach and other mixed-signal modelling tools (see section 4.4) is the way the model is simulated. Our approach is based

on function composition, while other tools are based on value-passing between components. Such tools do not allow *exact* time transformations such as time delays (section 4.3). Furthermore, solvers are used for simulation. Simulation is performed with a global time step, set by the solver, and the whole model is evaluated at this set time. Since our approach uses functions of time instead of values, a component has local control over the time step, and this time step is automatically propagated backwards in the model as we will show.

5.4.1 *Evaluation*

As models are a composition of functions, simulation is simply a matter of evaluating the composed function or model. Since simulation is evaluation over time, the top level component needs to be a CT component. The output of the top component is a function of time to a vacuous result, of which the time is used as a time step for updating the visualisations, i.e. the model is updated until that time. In addition, the top component has a vacuous input.

The visualisation update or simulation time step can be larger than the time step used locally for e.g. sampling or an integral approximation. In that case, the ADC or integral component are evaluated a few times with their local time step until they have reached the simulation time. The output signal of a component is applied to a time. This component, for example a plot component, then uses its local time step to evaluate its input signal. The time thus propagates backwards through the components. Only when necessary, for example for numerical approximation, are the input signals evaluated with smaller time steps, and only at the input of the component.

As an example, consider the evaluation of the model from section 5.3.1 repeated here:

$$source \;\triangleright\; delay_{0.15} \;\triangleright\; adc_{0.08} \;\triangleright\; add_{0.25} \;\triangleright\; sink$$

Assume a sine source, i.e. $source\ () = t \mapsto \sin(t)$. The input signal of $sink$ which is used for visualisation is then:

$$\begin{aligned} &add_{0.25}\ (adc_{0.08}\ (delay_{0.15}\ (t \mapsto \sin(t)))) \\ &\qquad = t \mapsto 0.25 + \sin\left(\lfloor (t - 0.15)\,/0.08 \rfloor * 0.08\right) \end{aligned}$$

As explained before, the important aspect from this example is that t is unaltered by add, floored by adc and shifted by $delay$ before sin is applied to it. In case of an integration component often a numerical approximation is used with a step size much smaller than the simulation step size. For such an integration component we can locally apply the input signal to a time with the smaller step size until we have calculated the result for the simulation time. So the component has local control over the time granularity.

In summary, the overall model determines the step size for simulation, while the local time steps are used for the plotting resolution, approximation accuracy or sampling.

5.4.2 Visualisation

Visualisation of signals, components and models is actually a side-effect of the model. Hence, above we have left it out of the definitions. For example, a scope sink visualises its input signal but does not generate an output signal, i.e. it has a vacuous result. It is also possible to visualise a block diagram of the system or display the internal state of components. During simulation, the visualisation is updated each time the model is evaluated by producing a list of "plot commands". Since side-effects are not part of a function result in a computational sense, this can not be done directly in Haskell. Hence, they have to be an explicit output of the function. In this section we will explain how to deal with side-effects in UniTi, in this case for visualisation but we use the same approach for state in the next section.

Visualisation updates are implemented as an additional output vs (of type $[\mathcal{V}]$) of a component, which we call *views*. A view is a list of commands to the graphical environment, such as "draw a line in figure 1 from the last point to (1,1)". Thus, the type definition of the component is:

$$Component = Sig^n \to (Sig^m, [\mathcal{V}])$$

There are also commands for plotting a block of a block diagram or a process in dataflow graphs.

An example of a component is then:

$$add_n(x) = (n + x, [rectangle(\text{“+”} + show(n))])$$

Herein, the component for addition again has $n + x$ as output as in equation (5.3), but also a command to draw a rectangle with a plus symbol and a character representation of n using $show(n)$.

However, now components have two results, of which only the first is relevant to connected components. Therefore, the composition operators are redefined to only provide the first output to following components. The second output consists of the views, aggregated into a single list. The resulting definitions are:

$$\begin{aligned}
\varphi \ \triangleright \ \psi &= f \mapsto (h,\ vs + ws), \\
&\quad \text{where} \ \ (g, vs) = \varphi\ (f),\ (h, ws) = \psi\ (g) \\
\varphi \ \| \ \psi &= (f, g) \mapsto ((f', g'),\ vs + ws), \\
&\quad \text{where} \ \ (f', vs) = \varphi\ (f),\ (g', ws) = \psi\ (g) \\
\circlearrowleft \ \varphi &= f \mapsto (g,\ vs), \\
&\quad \text{where} \ \ ((g, h), vs) = \varphi\ (f, h)
\end{aligned}$$

Each component can have a view. A plot sink is typically used for visualising signals as in figures 5.8 and 5.16. Other components can also present a visualisation, such as an illustration of its functionality, besides performing a signal transformation, although positioning of graphical elements is still a manual process. For example, figure 5.19 shows a block diagram of mixed CT and DT components (a beamforming system), and figure 5.20 shows a DF graph.

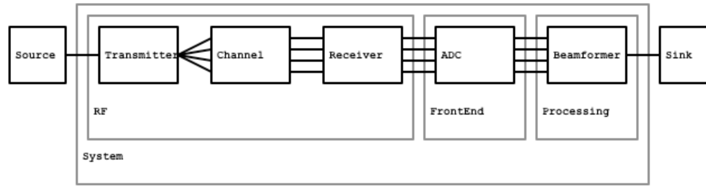


Figure 5.19: Beamforming system block diagram

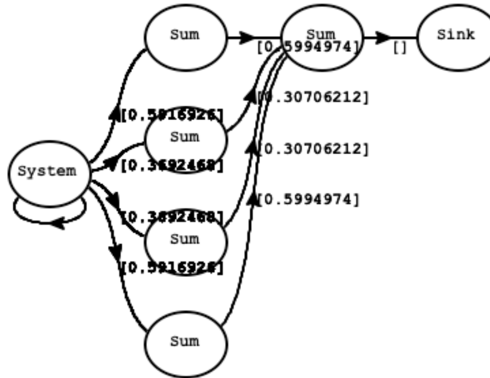


Figure 5.20: Dataflow graph

5.4.3 *Memory and state*

The use of memory or state has been mentioned several times. DF components have state for remembering the input channel contents. An example of a DT component with state is a FIR filter. A FIR filter uses a history of recent inputs for calculating the current output. An example of a CT component with memory is an integration operation.

Essentially, memory or state is not necessary for correct simulation results. Previous inputs (in case of DT) or an approximation over time (in case of CT) can simply be recalculated every time they are needed. However, simulation would quickly become very inefficient because of these redundant calculations. If we restrict t to be totally ordered, we can re-use previously calculated results. So for reasons of efficiency, previously calculated results are remembered.

We will first present how state is used and then we will present how the state is hidden for the final component.

5.4.3.1 **Using state**

The functionality of a component with state is defined with a function that has an extra input and an extra output for the state. The state for the integral, for example,

is the last time t_0 and value y_0 it calculated (using the integral definition from section 5.2.6):

$$\int_{h,(t_0,y_0)} (f) = t \mapsto (y, (t, y))$$

with

$$y = y_0 + \sum_{i=0}^{n-1} h \cdot f(t_i), \quad \text{where } n = (t - t_0)/h, \; t_{i+1} = t_i + h$$

Note that (t, y) at the output is the next state.

When the integral is used as a component, we have to explicitly manage the state, as can be seen from the following definition:

$$system_s(f) = t \mapsto ((g, s'), v)$$

where

$$((g, s'), v) = (source \;\triangleright\; int_{h,s} \;\triangleright\; adc_d \;\triangleright\; sink')\,()\,(t)$$

$$sink'(f) = t \mapsto (((), s'), plot(x)) \;\text{ where }\; (x, s') = f(t)$$

where g is the output signal, s' the updated state and v the aggregated views. Here, adc does not have to be changed, because of the extra state input from int, as adc only changes the time the integral is evaluated at; the result of the integral at that time is just some structure to pass on, with or without state. However, $sink$ does something with the resulting values, namely plot them. Therefore, $sink$ extracts the output value x and the state from the integral s', where x is used for the plot and s' is passed on to the output and back to $system$.

The difference in performance between a simulation with state and without state is quite substantial, especially for multiple integrations in sequence. For example, simulating the system presented above with 150 simulation time steps, and with about 10 integration steps per simulation time step, takes 2.833 s without state and 0.119 s with state on a 2 GHz Core 2 Duo system (a 24× speed-up). For two integration components in sequence, simulations without state almost become unmanageable taking 1863.201 s (±30 minutes) against 0.718 s with state (a 2600x speedup).

An example with state in the DT domain is a FIR filter. A FIR filter calculates the convolution of the impulse response of a filter ($\vec{h}$) with the input ($\vec{x}$):

$$y[t] = \left(\vec{h} * \vec{x}\right)[t] = \sum_{n=0}^{N-1} h_n \cdot x[t-n]$$

where N defines the filter order, $\vec{h}$ is the set of coefficients, $\vec{x}$ denotes the input data and y denotes the filter response. As can be seen it uses $N-1$ previous inputs. The FIR filter implementation thus has the previous $N-1$ inputs as state s:

$$fir_{\vec{h},\vec{s}}(x) = (\vec{h} \bullet \vec{x}, \vec{x})$$

where

$$\vec{x} = tail(\vec{s}) +\!\!+ [x]$$

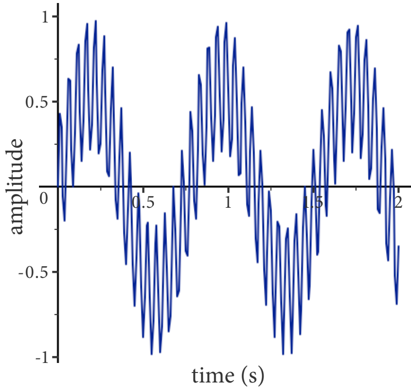


Figure 5.21: Two sources simulation

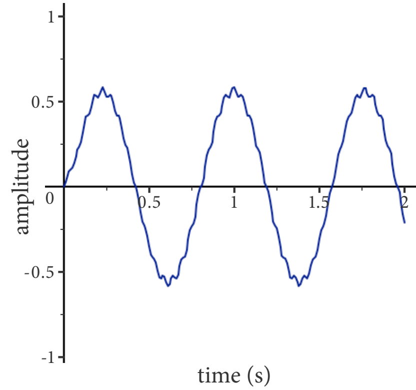


Figure 5.22: Filtered sources simulation

The function fir computes the convolution between the coefficients $\vec{h}$ and the inputs $\vec{x}$, where $\vec{x}$ is determined by dropping the oldest input of the state $\vec{s}$ (the head of the list) using $tail$, and adding a new input x to the end of the list. The state $\vec{s}$ of fir is the reversed list of the previous inputs, and the next state is the updated list of inputs $\vec{x}$. The dot product ($\bullet$) is denoted as:

$$\vec{h} \bullet \vec{x} = \sum_{n=1}^{N} h_n \cdot x_n$$

The FIR filter is applied to an input signal consisting of the sum of two sinusoidal signals, one of which has a 15 times higher frequency, as shown in figure 5.21. We filter this signal with a 8-taps low-pass FIR filter with filter coefficients $\vec{h}$ as follows:

$$\begin{aligned}
\vec{h} &= [0.069, 0.099, 0.140, 0.165, 0.165, 0.140, 0.099, 0.069] \\
\vec{s_0} &= [0, 0, 0, 0, 0, 0, 0, 0] \\
system_s(t) &= (\vec{s'}, y) \\
&\quad \text{where} \\
&\qquad (x, \vec{s'}) = ((src_1 \parallel src_2) \triangleright (+) \triangleright adc_d \triangleright fir_{h,s})\,()\,(t) \\
&\qquad y = sink(x)
\end{aligned}$$

where the initial state $\vec{s}$ consists of all zeros. As the FIR filter is in the DT domain, the output signal of fir is a value and a new state. Now, we can directly extract the state $\vec{s'}$ (i.e. the state does not have to pass through the $sink$), and just apply $sink$ to the value x.

The plot of the resulting values after evaluation is shown in figure 5.22. As can be seen, the source with the higher frequency is strongly attenuated.

In this example, the representation using composition operators and corresponding to a block diagram representation is split in two parts to extract the state.

The reason is that the state of the FIR filter must be bound to a name, so the *system* function can return it.

Clearly, there is a lot of additional effort managing the state that is not related to designing the system. Because the state depends on the time, the CT components have to manage the state explicitly as presented. For the DT components we can use special composition operators to pass the state over the component. Yet, the DT component itself can have state, requiring yet more operators to combine the state again.

Furthermore, state hierarchically moves up all the way to the top level component, as can be seen for *system*. For the top level component, the state of all sub-components is combined in a single state. This state has to be packed in and out, at each level, and each time a function with state is used. Therefore, state is globally managed, while it is a local property of a component. What we would like is to provide an initial state to the component and keep it local, i.e the state should not be visible when composing components. This can be achieved by making use of continuations to hide the state, as discussed next.

5.4.3.2 Hiding state

There are several options for implementing state hiding. We choose to use continuations, because it matches well with our representation for components and composition. A continuation represents a function, or in our case a component, that is to be used for the next input, i.e. it represents a function to continue the computation for the next input. Using continuations, the composition operators can hide all the plumbing, for supporting state, from the user.

To apply this, a component is a function from an input to an output *and* a new version of itself with updated state, the continuation:

$$Component = Sig^n \rightarrow (Sig^m, Component) \tag{5.9}$$

This is similar to the implementation of views in section 5.4.2; UniTi combines both views and continuations for a component.

As discussed above, the functionality of a component is defined using a function with an explicit input and output for state:

$$\begin{aligned} f &: \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{O} \times \mathcal{S} \\ f\ (s, i) &= (o, s') \end{aligned}$$

This function is applied to an initial state s_0 and an input and returns an output and a new state s'. Thus, the new state is already available before the next input. If f is partially applied to s', we get a new function f':

$$\begin{aligned} (o, s') &= f(s_0, i) \\ f'(i) &= f\ (s', i) \end{aligned}$$

where f' is the continuation, i.e. the function to use for the next input.

This principle is recursively applied for each new input using the $\Uparrow$ operator, which is defined as:

$$f \Uparrow s = i \mapsto (o,\ f \Uparrow s') \\ \text{where } \ (o, s') = f(s, i)$$

When a component is defined the initial state is provided ($f \Uparrow s_0$), after that, each input results in an output o and a continuation ($f \Uparrow s'$), with s' the next state. At the outside of $f \Uparrow s_0$ the state is not visible.

Of course, now we have to manage the continuations instead of the state. However, in contrast to the state, the continuations do compose. This is performed automatically by the composition operators. Thus, components with state compose just as components without state using these operators. All that remains are definitions for the composition operators which are similar to the definitions provided for composing views:

$$\begin{aligned} \varphi \triangleright \psi &= f \mapsto (h,\ \varphi' \triangleright \psi'), \\ &\quad \text{where } \ (g, \varphi') = \varphi\ (f),\ (h, \psi') = \psi\ (g) \\ \varphi \parallel \psi &= (f, g) \mapsto ((f', g'),\ \varphi' \parallel \psi'), \\ &\quad \text{where } \ (f', \varphi') = \varphi\ (f),\ (g', \psi') = \psi\ (g) \\ \circlearrowleft \varphi &= f \mapsto (g,\ \circlearrowleft \varphi'), \\ &\quad \text{where } \ ((g, h), \varphi') = \varphi\ (f, h) \end{aligned}$$

Herein, component φ is applied to input f, resulting in an output g and a continuation φ'. For ψ, it is applied to g resulting in h and a continuation ψ'. The result is a component with input f resulting in output h and a continuation: the sequential composition of the continuations of the sub-components.

The $\Uparrow$ combinator was already presented for the DF domain in section 5.1.3. It can also be used directly for the definition of the FIR filter above, i.e.:

$$fir_{DT} = fir_h \Uparrow \vec{s_0}$$

In both the DT and DF domain, a new input value or new input tokens change the state of the components. Multiple components with state can be composed using the composition operators. Components without state need to be represented in the same form as equation (5.9) in order to compose with component with state. This is easily achieved using:

$$\uparrow (f) = i \mapsto (f(i), \uparrow (f))$$

where the continuation is just the same function f again.

By embedding a DF component into a DT component, components with state from both domains can also be composed. This is achieved using `<~>` from section 5.3.2, straightforwardly adapted for `phi`, and also returning a continuation:

```
<~> psi = \x -> let ([y], psi') = psi[x] in (y, psi')
```

State in the CT domain is more difficult. State represents a result that is remembered over time. For the DT and DF domain, the next input represents a step in time, and the state changes because of the new inputs. In the CT domain, the inputs represent functions of time. Hence, a new input to a component does not represent a step in time in the CT domain. In fact, in the CT domain functions are composed only once, resulting in a function of time which is then evaluated over time.

However, for an integral as in the example above, we compute the input signal from t_0 to t in steps of h. Efficiency would be very much improved if the last computed result from the integration can be re-used. This is achieved by using a signal with state in the CT domain.

$$Sig_{CT} = Time \mapsto (\mathbb{R}, Sig_{CT})$$

where the second output is a continuation containing the updated state for the components. The integral is then defined as:

$$\int_{h,(t_0,y_0)} (f) = t \mapsto (y, \int_{h,(t,y)} (f'))$$

where the second output is now the $\int$ function again, but with updated state, *and* with an updated input signal f'. These are now computed as:

$$\begin{aligned} sum_f(t_0, t_1 \ldots t_n) &= (h \cdot x) + sum_{f'}(t_1 \ldots t_n) \text{ where } (x, f') = f(t_0) \\ (y, f') &= y_0 + sum_f(t_0, t_1 \ldots t_n) \\ &\text{where } n = (t - t_0)/h, \ t_{i+1} = t_i + h \end{aligned}$$

Herein, *sum* is computed recursively, and each step $f(t)$ is extracted into value x and continuation f', where x is added to the computation and f' is used for the next step. The final result is the value of integration up to t and the continuation f' updated up to t.

Each CT component now needs to be adapted to a signal that provides a value when applied to a time and a new signal to use next time. Therefore, the lifting operator is changed:

$$\widehat{g}(f) = t \mapsto (g(x), \widehat{g}(f')) \text{ where } (x, f') = f(t)$$

Unfortunately, such a signal representation does not compose when using feedback composition in the CT domain. We will discuss this problem next.

5.4.3.3 Feedback with state in the CT domain

For a feedback loop in the CT domain, the feedback signal typically steps back in time to some start condition, i.e. an integral from t_0. If it does not step back in time somewhere in the loop there is an infinite recursion or algebraic loop that will not terminate. However, this recursion back in time causes all results from t_0 to t to

be recalculated each time we evaluate the output signal. In the DT and DF domain the feedback loop also steps back in time, however, now the state can be used to update the start condition to the last computed value.

Though modelling the environment of beamforming applications does not include feedback, from a perspective of generalising UniTi, it would be beneficial to be able to support feedback in the CT domain. Our experiments so far did not yield a solution. It is not possible to add the state to a CT component, as this component is applied to the input signal only once, to compute the final output signal. Therefore, we have added state to the CT signal, as presented above. Each evaluation, the signals update their state with the latest value of the signal, such that it does not have to be recalculated when used in a feedback loop. Unfortunately, such a signal can not be used for a feedback composition. When such a signal is used for a feedback loop, the signal h used at the feedback input should be updated with the latest value at the feedback output h' of the component. However, at that point we do not have access to the continuation of h anymore, as h was needed to compute h'. Therefore h still contains the old value and will recursively do so back to the start condition again. As such, it remains an open problem.

In Haskell there are several options to explore for possible solutions. For example, a sort of hash table can be used to store already computed values, called *memoisation* [25]. However, it then must be decided when such values can be thrown away. Exploring these directions to solve the problem with feedback is left as future work.

We do note that this is a matter of efficiency and not correctness, although it is a serious efficiency issue. Furthermore, it is possible to use a DT or DF feedback loop to replace the CT one as a workaround.

5.5 Model transformations

In chapter 4 we have presented a design flow for dividing functionality over the domains and within a domain. UniTi allows for a single model during the design process. Because of the integrated approach presented, we can apply model-based design using transformation steps. Furthermore, the transformations preserve correctness of the design. Nevertheless there is very limited support for model transformations in existing tools (see section 4.4). We will discuss model transformations for the co-design and partitioning steps.

5.5.1 *Co-design*

UniTi supports a number of features to assist the co-design step of he design flow. The basic algebraic mathematical operators such as $+$ and $\cdot$ are overloaded so the same operator can be used for all domains using the *type class* feature of Haskell. That means that the type of the signal determines the specific operator implementation that is used and that the semantics of the operator in each domain are the same. Therefore, a mixed CT and DT model is transformed from a CT model by only adding an ADC.

For example, a domain independent definition of an addition of 1 (bias) followed by a multiplication with 0.12 (gain) is:

$$(+1) \;\triangleright\; (*0.12)$$

where the input signal determines whether functions of time, values, or tokens are added and multiplied. However, without changing the definitions and by only adding an ADC, the addition is in the CT domain while the multiplication is in the DT domain:

$$(+1) \;\triangleright\; adc_{0.1} \;\triangleright\; (*0.12)$$

Of course, the placement of the ADC and in general the division over the domains is a manual operation by the designer, as the relevant properties for assessing the trade-off, such as cost in terms of money or energy, are not part of the model. That is not to say they could not be; further research into this direction would be interesting.

5.5.2 *Partitioning*

During the partitioning step in UniTi, an application is parallelised using model transformations. However, parallelising an application is not straightforward, as the dependencies between computations must be derived. Here, we present guidelines on specifying applications so they can be parallelised more easily. This involves identifying parallelism (in an application), and defining an application such that the parallelism can be exploited by model transformations in UniTi. Specifying applications as such is performed by using the mathematical (model) definitions supported by UniTi. Then we will present how such applications can be partitioned, with an example of a distribution model transformation.

We identify two kinds of parallelism: data parallelism and control parallelism. In the first kind of parallelism the data is split. Examples are bit-level and data-level parallelism. In the second kind the control, i.e. the operations on the data, is split. Examples are instruction, task and pipeline parallelism.

5.5.2.1 **Control parallelism**

Control parallelism occurs when some operations or functions are executed in sequence. A section can already continue with the next data, while later sections are still operating in parallel on previous data. To keep execution functionally correct, the sections may not influence each other besides the explicit input and outputs, i.e. the function must be side-effect-free with respect to the calculation.

These restrictions are captured by the DF model. Passing arguments to mathematical functions is similar to communicating values between processes. In the DF domain, data in channels must remain ordered, making sure the operations are performed in sequence. Back-pressure (a process is stalled if the tokens are not consumed from the output buffer fast enough by the next process, see section 4.1.3)

ensures automatic synchronisation in parallel execution. Thus, the computation (functionality) and communication (the inputs and the outputs) are made explicit to fit to the dataflow model and are wrapped in a DF component.

5.5.2.2 Data parallelism

Data parallelism occurs when some operation or function has to be executed on the data in aggregate data structures such as lists, arrays or trees. There are at least two elementary forms of such operations, the first applies an operation to each element of an aggregate data structure separately, the second gathers the elements together into a single outcome (as in "map-reduce"). The dot product below explains this in further detail.

5.5.2.3 Aggregate operations

In order to recognise and isolate data and control parallel properties of operations in an application, it is beneficial to formulate the application on a level that is as high as possible. That is to say, to specify operations on the aggregate level rather than on the element level.

As an example, consider the standard definition of the dot product of two vectors (such as used for a FIR filter or beamformer):

$$\vec{a} \bullet \vec{b} = \sum_{i=1}^{N} a_i \cdot b_i \tag{5.10}$$

In this definition the operations for addition and multiplication occur on the element level, where the individual elements are indicated by the index i. This formulation strongly suggests a *for*-loop in which for each pair of elements both operations are performed, aggregating the results step by step into a final sum. However, in general it is difficult to parallelise such an implementation, since the operations are entangled with each other at every step of the for-loop, leading to algorithmic structures which are hard to disentangle, especially when side-effects arise. This problematic character is confirmed by the extensive research to automatically parallelise for-loops in existing code [27, 69].

We will choose a different approach by looking at such algorithms from a more abstract perspective: instead of defining the dot-product by using indices and by intertwining the operations $+$ and $\cdot$ together into one computational activity, we will "lift" the operations to the aggregate level, in this case to the vector level. From equation (5.10), it can be seen that we need:

- pairwise multiplication of the elements of the vectors. We use the notation $\widehat{\cdot}$ for this lifted version of multiplication. Note that this usage of the operator $\widehat{\ }$ is in accordance with earlier usage, since a vector can be seen as a function from indexes to values.
- the reduction of the resulting values to a single value by using $+$. We use $\bigoplus$ to denote this interpretation of addition, i.e. the expression $\bigoplus \vec{x}$ means that the elements of vector $\vec{x}$ are summed.

We remark that this can be generalised to other operations than multiplication and addition as well.

Clearly, the dot product of two vectors $\vec{a}$ and $\vec{b}$ can now be defined as follows:

$$\vec{a} \bullet \vec{b} = \bigoplus \left(\vec{a} \,\widehat{\cdot}\, \vec{b}\right) \tag{5.11}$$

Note that our notation does not involve reference to the individual elements in the vectors, so no indices are needed. What is further important to observe is that the operations $\widehat{\cdot}$ and $\bigoplus$ are now visible on aggregate level. In the algorithm for the dot product these operations are *separated*.

Now, it is possible to use such definitions at the aggregate level for partitioning, so that we can formalise it as a model transformation. The $\widehat{\cdot}$ operates on the data independently, so it is easy to parallelise. However the reduction operation $\bigoplus$ must be associative to be able to use parallelism. For example, splitting vector $\vec{a}$ and $\vec{b}$ in three sub-vectors $\vec{a}_1, \vec{a}_2, \vec{a}_3$, respectively $\vec{b}_1, \vec{b}_2, \vec{b}_3$ (where $\vec{a}$ and $\vec{b}$ are equally long) leads to the following parallelisation of the dot product:

$$\begin{aligned} e_i &= \bigoplus \left(\vec{a}_i \,\widehat{\cdot}\, \vec{b}_i\right), \quad \text{where} \quad i = 1, 2, 3 \\ \vec{a} \bullet \vec{b} &= \bigoplus \left[e_1, e_2, e_3\right] \end{aligned} \tag{5.12}$$

Note that the indices here indicate that we have chosen to partition the dot product in three parts, i.e. they are part of the partitioning, not of the application definition. Further, the calculation of e_i and $\bullet$ can be pipelined and has a tree-like computational structure.

Thus, data parallelism is provided by defining the operation on aggregate data and control parallelism is provided by separating the $+$ and $\cdot$ operations and by staging the reduction operation in a tree. This last approach is an example of a divide-and-conquer strategy [101]. Next, we will present the model transformation to perform this partitioning automatically.

5.5.2.4 Transformation

In the previous section it has become clear that how the functionality is specified influences how much parallelism can be exploited. It is ongoing research how to transform such structures to the aggregate level automatically, and for now this is a manual process. However, when the algorithm is specified on the aggregate level, we can automatically partition it to execute data-parallel or with a divide-and-conquer strategy. This is done with a higher-order function, that takes the aggregate operation and generates a number of connected dataflow processes, i.e. the step from equation (5.11) to equation (5.12) is automated (assuming the reduction operation is associative). The granularity (the vector is split in three in our example above) is a manually specified parameter. The amount of computation and communication per process must be matched with the capabilities of the processors and the network.

Distribution As an example we define the higher-order transformation for the implementation of the dot-product of equation (5.12):

$$\vec{h} \bullet \vec{x} = \bigoplus (\vec{h} \,\widehat{\cdot}\, \vec{x})$$

To distribute the dot-product the inputs are split every n values. The function $split$ cuts a vector $\vec{x}$ in a sequence of sub-vectors of length n:

$$split_n\ \vec{x} = \langle \vec{x}_1, \vec{x}_2, \ldots \rangle$$

It can be defined recursively as follows:

$$split_n\ [\,] = [\,]$$
$$split_n\ \vec{x} = \vec{a} : split_n\ \vec{b}$$

where

$$(\vec{a}, \vec{b}) = splitAt_n\ \vec{x}$$

Herein, $splitAt_n$ splits the vector $\vec{x}$ into a vector of the first n elements ($\vec{a}$) and a vector of the remaining elements ($\vec{b}$).

Furthermore, we normalise the coefficients $\vec{h}$ (h is the first element of $\vec{h}$, and $\vec{h}'$ the remaining part):

$$normalise\ \vec{h} = 1 : \widehat{(/h)}\ \vec{h}'$$

which is only allowed if the function we distribute, the dot-product in our case, is distributive over addition.

Finally we define a generic distribution transformation for any reduction function f that takes two arguments ($\vec{h}$ and $\vec{x}$) and is distributive. For a singleton vector $\langle x \rangle$ we have

$$distribute_n\ f\ \vec{h}\ \langle x \rangle = x$$

whereas for arbitrary vectors $\vec{x}$ we have

$$distribute_n\ f\ \vec{h}\ \vec{x} = distribute_n\ f\ \vec{h}'\ \vec{y}$$

where

$$\mathbf{x} = split_n\ \vec{x}$$
$$\mathbf{h} = split_n\ \vec{h}$$
$$\vec{h}' = \widehat{head}\ \mathbf{h}$$
$$\mathbf{h}' = \widehat{normalise}\ \mathbf{h}$$
$$\vec{y} = \mathbf{h}'\ \widehat{f}\ \mathbf{x}$$

where the inputs are split every n values, the coefficients are normalised and the results are recursively distributed again. Only the granularity n and the reduction function f need to be specified. Note that we write the application of f to corresponding elements from $\mathbf{h}'$ and $\mathbf{x}$ in infix notation.

5.5.3 *Design space exploration*

We have touched upon a number of trade-offs that the designer must specify:

- analogue/digital co-design
- hardware/software co-design
- the way the functionality is specified
- the granularity of the partitioning

As these are trade-offs, it is very helpful for a designer to try a few different alternatives, so-called design space exploration. With the help of UniTi, in many cases it is simply a matter of moving the ADC or applying a different parallelisation strategy.

By using aggregate operations the model is independent from the number of data elements. However, the number of data elements does influence the number of processes or the amount of computation and communication per process. As the partitioning is automated we can quickly explore the results with a different number of elements and with different granularities, without any changes to the model besides the granularity.

5.6 Conclusion

In this chapter we have presented the formalisation of UniTi. UniTi provides a framework for multi-domain modelling and simulation, and as such unifies time, signals and components in the CT, DT and DF domains.

In each domain components are signal transformations, but the signals themselves are different. In the CT domain, signals are functions of time; a transformed signal is thus also a function of time. Moreover, the time reference can be changed, enabling the framework to include time transformations in the formalism, and allowing exact simulations of models including such transformations. In the DT domain signals are values; values can change over time, but that is outside of the influence of the DT component. In the DF domain signals are lists of tokens representing channel updates. Therefore, DF signals represent the change of tokens over time to match with the standard interpretation of signals in the CT and DT domain and allowing the integration of the DF domain. This is significantly different from the standard representation of dataflow models, where channels contain tokens and provide the connection between processes. In UniTi, the current contents of the input channels are part of the DF component, together with an implementation of the firing rules and channel management. These tasks are provided by the UniTi framework.

As components in all domains are signal transformations, we can use unified sequential, parallel and feedback composition. Sequential composition uses the output of the first component as input of the next. Parallel composition provides the first signal to the first component and the second signal to the second component. Feedback composition feeds back the second output signal to the second input of the same component, resulting in a component with a single input and a

single output thereby hiding the feedback signal. Feedback is also used to support components with state.

In UniTi the notions of time are separated. The time used locally at a component can be different from the simulation time used for the final composed model. The sample time also does not have to match with the simulation time, as the simulation time is changed to the latest sample time, locally at the ADC component. Components that deal with change over time, such as integration or differentiation, in the general case need a solver. In all tools, this solver is global and uses a global approximation time step. In UniTi the used solver is locally applied for the component, enabling the designer to choose a specific solver, as well as the approximation time step used, at each component.

For integration, a DT component is embedded into a CT component. This is achieved by letting the time at which a CT output signal is evaluated determine the value of a CT input signal to which the DT component is applied. Furthermore, the embedding of a DT component is automated if a CT and DT component are combined. To integrate a DF component, it is embedded in a DT component by using the values of a DT signal as token updates, and by using a single token as output value. However, their integration limits the tokens at the boundaries of a dataflow model to have a value representation. The sample time of a DT signal determines the token arrival time at input of the dataflow model, thereby providing a time reference for the execution time and thus token production time of dataflow processes. The embedding of DF components is also automated. The final composed and integrated component is a CT component, thereby integrating the time in the model. This is deliberate, so we can use this component to evaluate the model over time for simulation.

Feedback in the model will trigger a computation that recurses back in time to a terminating condition during simulation. This is not efficient as results are recalculated for each simulation time step. Therefore, state is introduced to remember previously calculated results, forcing the time in the model to be causal and ordered. Keeping state is implemented by using continuations, i.e a component provides an output and an updated version of itself. Furthermore, components can provide a visualisation, either a plot or a figure of the current state of the component. These continuations and visualisations are combined, up to the final component representing the model, during composition.

Model transformations are used for division over domains during the co-design step and division within a domain during the partitioning step. The automated integration of multi-domain components facilitates domain independent definitions, thereby allowing their domain to be determined by the context and allowing fast and easy changes to the model. Within a domain, aggregate operations, such as element-wise or reduction operations, allow model transformations by exploiting the mathematical properties of a definition.

Overall, UniTi provides a framework for modelling CT, DT and DF components in a single model. In combination with mathematical definitions of model components, this allows a model-based design flow including support for model

transformations. Furthermore, different notions of time are represented in the model and integrated with UniTi, enabling exact simulation of time transformations and modelling execution time of dataflow processes.

CHAPTER 6

Case study

ABSTRACT – *In this chapter we will consolidate the work presented in the previous chapters with a case study on the design of a generic beamforming platform using a tiled reconfigurable architecture. The design flow and framework of UNITI is applied during the design process. First, a specification of the beamforming application is provided, which is executable for simulation and verification. Next, this specification is subdivided into sub-components representing the environment, the architecture and the application during the analogue/digital and hardware/software co-design step. As a result, components are modelled in the CT, DT and DF domain. These models are compared to Simulink and found to be more computationally efficient whilst also supporting exact time delays as experienced by the antenna signals. Thereafter, the partitioning step subdivides the beamformer onto a tiled architecture. During the design process the model becomes more specific. For the mapping and implementation step we will use the adaptive beamformer on a small architecture consisting of three reconfigurable processors. The case study is concluded with a discussion on the applicability and flexibility of UNITI for the design of a beamforming platform. UNITI is found to be very capable, yet the mapping and implementation steps are still completely manual and could greatly benefit if supported by UNITI.*

In this chapter the design flow and framework of UNITI is applied to a non-trivial case study. An embedded system is designed for the phased array beamforming application from chapter 2 based on a tiled reconfigurable architecture as presented in chapter 3. This is achieved using a *single* model which is refined into a more detailed model by the design steps presented in chapter 4 and supported by the formalisms for modelling domains and model transformations from chapter 5.

Parts of this chapter have been published in [KCR:3], [KCR:6], [KCR:9], [KCR:10], [KCR:11] and [KCR:14].

The phased array system consists of a processing part performing the beamforming operation and beamcontrol and a front-end for each antenna signal.The case study also includes a model of the environment to generate the signals received at the antennas of the phased array system, allowing us to execute the model and verify the correct operation of the beamforming and beamcontrol processing. In chapter 2 we have developed digital signal processing (DSP) algorithms to adaptively steer the main beam in order to track a moving source. Therefore the model of the environment generates source signals with a moving position so as to accurately simulate the antenna signals that the real system would receive. However, this means the signals generated from the environment must model the relevant properties of the environment *exactly*, otherwise we can not differentiate whether errors are caused by the model of the environment or caused by the adaptive beamcontrol algorithm. As we found in chapter 4, Simulink and other tools *do* introduce interpolation inaccuracies when modelling time delays such as experienced by the signals from a source to each of the antennas of a phased array receiver. This is especially relevant because for each antenna of the array, the delay of a signal is different, so a large array has many time delay components (which will even vary over time), causing a major problem in Simulink. It is possible to approximate the delay with a phase shift as a workaround, but that still introduces inaccuracies and is therefore only valid for narrowband signals, excluding wideband beamforming applications. UniTi, as we will see, is able to model the environment for a phased array system exactly.

We will develop the case study in three parts with increasing complexity:

- a simple beamformer with the path length between the transmitter of the source and each receiver of a planar array modelled by time delays and beamsteering based on a phase shift correction with a fixed steering angle, i.e. without a beamcontrol algorithm,
- an adaptive beamformer using a ULA and E-CMA for adaptive beamcontrol, thereby introducing QPSK modulated signals and feedback loops in the system,
- and a hierarchical beamformer with A-CMA as adaptive beamcontrol algorithm.

These are presented as such to gradually develop the model, each time focusing on a specific part of design; the model of the environment, the adaptive control algorithm and hierarchical beamforming respectively. During the design steps the model is refined and therefore becomes more specific; for the later stages of the design process we will use the E-CMA-based adaptive beamformer on the LEON SoC platform, a small tiled reconfigurable architecture consisting of three Montium reconfigurable processors and a NoC (see section 3.3.2).

In section 6.1 we will present a formal specification of the basic functional behaviour of the system. During the analogue/digital and hardware/software codesign step in section 6.2 the case is refined into a signal flow model and compared with a Simulink equivalent (in three parts). The processing is to be performed on multiple reconfigurable processors. The beamcontrol is expected to run on a sin-

gle processor, but the beamforming processing must be partitioned as presented in section 6.3. Next, we map the adaptive beamformer on a tiled reconfigurable architecture in section 6.4, and the implementation is presented in section 6.5. The UniTi model provides the input signals for the implementation, enabling verification without requiring actual transmitters, receivers and antenna front-ends. Finally, the results are presented in section 6.6.

6.1 Specification

We would like to design a beamforming system supported by a realistic simulation of the system and the environment. The final system should be suitable for multiple beamforming applications and therefore a tiled reconfigurable architecture is proposed for scalability and flexibility. In this section a formal specification of the functional behaviour is presented.

Simple beamformer A basic beamforming system is based on the Friis equation (section 2.2) and consists of the correction of the time delays caused by the path length differences between sources and receiving antenna elements. Below we give a mathematical specification for a beamforming system, assuming a single source, and an array of antennas.

Let s be the signal coming from this source, i.e. s is a function of time. Suppose that the *position* of antenna a_i is indicated by p_i, and that the DoA of signal s is indicated by d. Note that both p_i and d are vectors (with their origin at the centre of the array), though p_i is a vector $\langle x_i, y_i, z_i \rangle$ of cartesian co-ordinates whereas d is a vector $\langle r, \alpha, \gamma \rangle$ for range, azimuth and elevation.

Clearly the delay δ_i depends on the position p_i of antenna a_i and on the direction of arrival, and can be calculated as follows (c is the speed of light):

$$\delta_i = \frac{\ell(p_i, d)}{c}$$

where $\ell(p_i, d)$ is the *length* function which expresses the distance between antenna a_i and source s, defined as follows:

$$\ell\left((x_i, y_i, z_i), (r, \alpha, \gamma)\right) = \sqrt{(x_d - x_i)^2 + (y_d - y_i)^2 + (z_d - z_i)^2}$$

with

$$\begin{aligned} x_d &= r \cdot \sin(\alpha) \cdot \cos(\gamma) \\ y_d &= r \cdot \cos(\alpha) \cdot \cos(\gamma) \\ z_d &= r \cdot \sin(\gamma) \end{aligned}$$

Note that δ_i is a scalar value. However, as described in section 5.1.1 we consider the *delay* of a signal s with a value δ as a *signal transformation*, i.e, as a function with

signal s as argument and a changed signal as result. Remember that this delayed signal is defined as

$$delay_\delta(s) = t \mapsto s(t - \delta)$$

Thus, the sequence of values of the delayed signals at time t for the array of antennas $\langle a_1, a_2, \ldots \rangle$ is (s is the original signal):

$$\sigma(t) = \langle delay_{\delta_1}(s)(t), delay_{\delta_2}(s)(t), \ldots \rangle$$

The phase correction w_i for antenna a_i and steering direction $\langle \alpha_0, \gamma_0 \rangle$ is calculated as (λ is the wave length of the carrier):

$$w_i = e^{j \cdot \frac{2\pi}{\lambda} \cdot \Delta l_i}$$

where Δl_i is the difference in length between the origin and antenna a_i, projected in the steering direction, calculated as follows ($\bullet$ is the dotproduct of two vectors):

$$\Delta l_i = \langle x_i, y_i, z_i \rangle \bullet \langle 1, \alpha_0, \gamma_0 \rangle$$

Let $\vec{w}$ be the steering vector $\langle w_1, w_2, \ldots \rangle$ and let $\vec{w}^*$ be its complex conjugate. Then the *beamformer* bf applies a phase shift correction and is defined as the function (N is the number of antennas):

$$bf_{\vec{w},\sigma} = t \mapsto \frac{\vec{w}^* \bullet \sigma(t)}{N}$$

The above mathematical specification can be expressed directly as Haskell code as shown in chapter 5. This code is shown in listing 6.1. Most of the code is straightforward from the definitions. We only remark that `xs!i` select the `i`th element from list `xs`, `**` implements the dot product as defines in section 5.5 using `zipWith (*)` for the element-wise multiplication and `sum` for the reduction with addition. Furthermore, `spher2cart` implements a coordinate transformation, and although angles are represented as degrees for clarity the trigonometric functions expect radians. Finally, complex numbers are used, where `:+` is an infix operator expecting the real and imaginary part, and `cis` construct a complex number with magnitude 1 and its argument as angle. For simulation, a number of constants are defined that are used in the definitions, e.g. `ps` represents the antenna positions. Simulation is performed by evaluating the `model` for a list of simulation time steps `ts` using `map`, which applies the `model` to each time step one by one. Note that most definitions are globally defined and used in other definitions.

By executing this code we have verified that the output signal of the beamformer corresponds exactly to the signal of the source when steered in the correct direction. Figure 6.1 shows the resulting signal when mispointing, i.e. a 40 MHz sine source with amplitude 1 and initial phase 0° arriving from 11° azimuth (shown in light grey) while the 4-element ULA beamformer is steered to −3° azimuth. As expected the sine wave is attenuated because of mispointing and the initial phase has shifted because the distance to the centre of the array (100.3λ) is not a multiple of the carrier period (see section 2.4.2).

```
1  -----------------------------------------------------------------------
2  -- model
3  -----------------------------------------------------------------------
4
5  xs ! i     = xs !! (i-1)
6
7  hs ** xs   = sum (zipWith (*) hs xs)
8
9  delta i    = (ell p d)/c
10                             where
11                               p = ps!i
12
13 spher2cart (r,a,e) = (x,y,z)
14                             where
15                               x = r * sin(a) * cos(e)
16                               y = r * cos(a) * cos(e)
17                               z = r * sin(e)
18
19 ell (x,y,z) (r,a,e) = sqrt ((xd-x)^2 + (yd-y)^2 + (zd-z)^2)
20                             where
21                               (xd,yd,zd) = spher2cart (r,a,e)
22
23 sigma t = [ (delay (delta i d) s t) | i <- [1.. n] ]
24                             where
25                               d = doa t
26
27 w i = cis (2*pi/lambda * (dl i))
28
29 ws  = map w [1..n]
30
31 dl i = [x,y,z] ** [xd,yd,zd]
32                             where
33                               (x,y,z)    = ps!i
34                               (a,e)      = b
35                               (xd,yd,zd) = spher2cart (1,a,e)
36
37 bf ws sigma = \t -> ((map conjugate ws) ** (sigma t)) / (n :+ 0)
38
39 model = bf ws sigma
40
41 -----------------------------------------------------------------------
42 -- simulation
43 -----------------------------------------------------------------------
44
45 c        = 300*10^6
46 f_c      = 40*10^6
47 lambda   = c / f_c
48 d        = lambda/2
49
50 ps = [  (x-(n-1)/2)*d, 0, 0) | x <- [0..(n-1)] ]
51
52 b = (-3, 90)
53
54 d = (1000, 11, 90)
55
56 s = \t -> cis (2*pi*f_c*t)
57
58 stepsize  = 1 / (50 * f_c)
59 ts        = [0,stepsize..100e-9]
60 sim model = map model ts
```

Listing 6.1: Simple beamformer specification

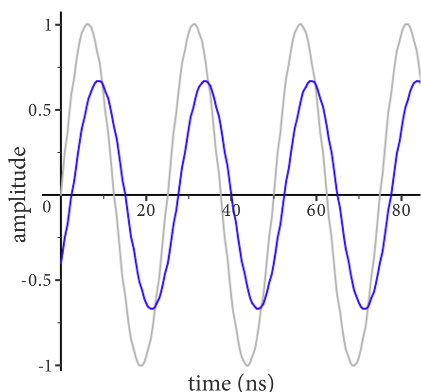


Figure 6.1: Beamformer input (in grey) and output for $N = 4$, $\vartheta = 11°$ and $\vartheta_0 = -3°$

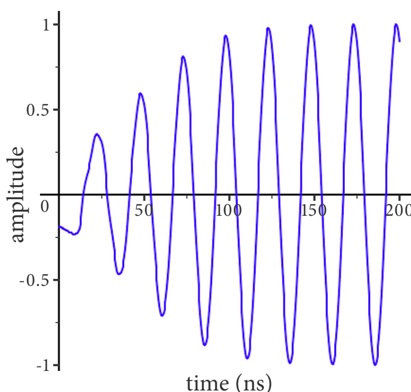


Figure 6.2: Correction by E-CMA for initial mispointing

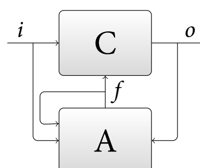


Figure 6.3: Adaptive algorithm

Adaptive beamcontrol In the above specification, the DoA d and steering vector $\vec{w}$ are fixed over time. For the cases with *adaptive* beamcontrol, the DoA is dynamic and depends on time. Thus, DoA d becomes a function of time t. Furthermore, the steering vector or steering angle is determined by the beamcontrol algorithm.

In order to define adaptive beamcontrol, we first give a general formulation of an adaptive algorithm (see figure 6.3). An adaptive algorithm consists of a component C which transforms an input signal i into an output signal o, given some steering or correction factor f. The correction factor f is then updated by the adaptive control component A based on the input i, the output o, and the (previous) factor f. Such an algorithm is specified by the following equations:

$$o = C(f, i)$$
$$f = A_f(i, o)$$

Note that in these equations all variables are functions over time, and that the results of C and A_f are also functions over time. In practice only a finite number of correction factors are calculated, i.e. the *function* f then becomes a *sequence* of values $\langle f_0, f_1, \ldots \rangle$, with f_0 a given initial value. The equations then transform into:

$$o(t) = C(f_t, i)(t)$$
$$f_{t+1} = A_{f_t}(i, o)(t)$$

Note that in these equations i and o are still functions over time.

To apply the above to adaptive beamforming, let σ be a vector of input signals for the beamforming component bf, y the output of bf, and $\vec{w}$ the steering vector (corresponding to the correction factor above). As above, σ and y are functions over time, whereas $\vec{w}$ is a steering vector at a specific moment in time. In our case the output y thus is defined as:

$$y = bf_{(\vec{w},\sigma)}$$

The E-CMA algorithm (see section 2.4.2) is defined as follows:

$$ecma_{\vec{w}}\left(\sigma, y\right) = t \mapsto \vec{w} \mathbin{\widehat{-}} \mu \cdot \varepsilon \mathbin{\tilde{\cdot}} \sigma(t)$$

where

$$\varepsilon = \frac{8 \cdot \left(|y(t)|^4 - |y(t)|^2\right) + j \cdot (\text{-}M \cdot \sin\left(M \cdot \angle\, y(t)\right))}{4 \cdot y(t)}$$

Note that $M = 4$ for QPSK modulated signals. Strictly speaking, the notations $\widehat{-}$ (for element-wise vector subtraction, see section 5.5) and $\tilde{\cdot}$ (for scalar-vector multiplication) are not necessary in a mathematical specification. However, anticipating the implementation we choose to make the parallelism in these operations explicit.

Now the updated steering vector $\vec{w'}$ at time t is

$$\vec{w'} = ecma_{\vec{w}}\left(\sigma, y\right)(t)$$

To simulate this, we have to calculate the results over a sequence $\mathbb{T}$ of time moments, with initial steering vector $\vec{w}_0$. That is, we have to calculate

$$results_{\vec{w}_0}(\mathbb{T})$$

with $results$ recursively defined as

$$results_{\vec{w}}\left(\mathbb{T}\right) = y(t) : results_{\vec{w'}}\left(\mathbb{T}'\right)$$

where

$$y = BF_{\vec{w},\sigma}$$
$$\vec{w'} = ecma_{\vec{w}}\left(\sigma, y\right)(t)$$

and t is the first element of the sequence $\mathbb{T}$ and $\mathbb{T}'$ the remaining elements of $\mathbb{T}$, and with ":" the list constructor operator.

Note that time is discretised here for the simulation time only, and not for the time delay as discussed in chapter 5, meaning that the calculation of the various time delayed signals in the beamformer is exact. In addition, the time step for simulation determines the update step for the adaptive beamcontrol algorithm, i.e. the time step for simulation is the same as the time step for updating the steering vector. We will uncouple these two time steps in the next section.

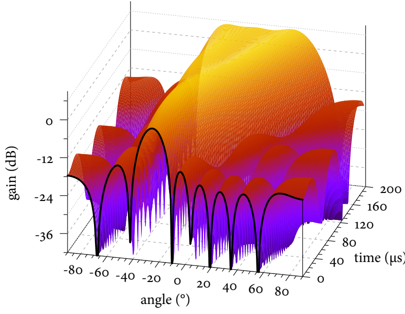


Figure 6.4: E-CMA radiation pattern for linear increasing DoA from −25° to 60°

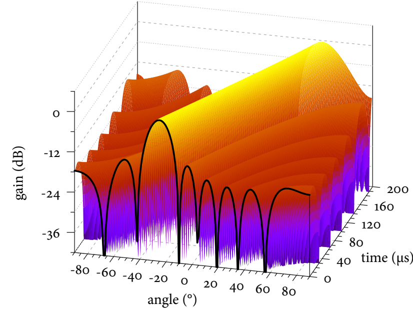


Figure 6.5: A-CMA radiation pattern for linear increasing DoA from −25° to 60°

As before, the above specification is immediately translated into Haskell and evaluated for simulation. The adaptive beamformer is simulated in a scenario in which the azimuth α increases over time by $4 \times 10^5\, t$, and initially starts at $-25°$. The range r is fixed to 1000 and the elevation γ to 90°. Thus the DoA $d(t)$ is:

$$d(t) = (1000, (\text{-}25 + 4 \times 10^5\, t)^\circ, 90^\circ)$$

Figure 6.2 shows the beamformer output for a simulation from 0 µs to 0.2 µs with an 8-element ULA and a sine wave source. The DoA of the source is −25° while the array is initially steered to 0°, causing a distorted signal which is quickly corrected by E-CMA. Figure 6.4 shows the resulting radiation pattern of the E-CMA algorithm over time for a simulation from 0 µs to 200 µs. In this time frame the DoA increases from −25° to about 60°. As there are no interferers E-CMA causes the beamwidth to widen as the angle increases. The E-CMA also increases the gain to compensate for the modulus decrease because of mispointing besides changing the phase. Note that the input signal is a unmodulated (analytic representation of a) sine wave resulting in a single modulation point.

Hierarchical beamcontrol For hierarchical beamcontrol the A-CMA algorithm is used, defined as follows (see section 2.4.3):

$$acma_\theta\,(\sigma, y) = t \mapsto \theta - \mu \cdot \varepsilon$$

where

$$\begin{aligned} \varepsilon &= \left(|y(t)|^2 - 1\right) \cdot (\sigma(t)^H\, \mathbf{B}'\, \sigma(t)) \\ \mathbf{B}'_{c,r} &= (c - r) \cdot \phi'_\theta \cdot e^{(c-r)\cdot\phi_\theta} \qquad \text{for all } c, r \in \{1 \cdots N\} \\ \phi_\theta &= j \cdot 2\pi d \cdot \sin\theta \\ \phi'_\theta &= j \cdot 2\pi d \cdot \cos\theta \end{aligned}$$

Here, $\mathbf{B}'$ is multiplied on the left with the hermitian of the vector $\sigma(t)$, and on the right with the vector $\sigma(t)$ itself, using a matrix multiplication.

A-CMA calculates a steering *angle* θ, while the beamformer expects a steering *vector*. The steering vector $\vec{w}$ can be calculated from θ using a LPT:

$$\vec{w}_i = e^{j\cdot\pi \sin\theta\cdot i} \qquad \text{forall } i \in \{1,\ldots,N\}$$

with an antenna spacing of $\lambda/2$.

Figure 6.5 shows the radiation pattern over time for the same scenario as above. The result looks much cleaner because there is no gain variation and the radiation pattern is fixed over the angle, since no gain taper is applied by A-CMA.

6.2 Co-design

During the co-design step, functionality of the initial specification is divided into sub-components representing the environment, the architecture (hardware) and the application (software). This is achieved by encapsulating functionality in components and connecting them using the sequential, parallel and feedback composition operators defined in chapter 5. Furthermore, domains are introduced, i.e. components are in the CT, DT or DF domain. In order to keep the complexity manageable, this is performed in three steps; first basic beamforming, then beam-control, followed by hierarchical beamforming. During each step a corresponding Simulink model is used for comparison and benchmarking. For this design step we will assume a single core architecture with a dedicated front-end per antenna.

6.2.1 *Simple beamformer*

The division of functionality over components is based on the system design presented in section 2.3.3. To validate the phased array receiver, the signals received at the antennas are generated, thereby modelling the environment. Therefore, a source, a transmitter and a channel are added. The model of the channel implements the delay from the source to the different receiver antennas.

The RF front-end is implemented in analogue hardware as the frequencies are too high to allow the use of only digital hardware. After down-conversion, the signals are digitised and filtered by the AP block in fixed digital hardware. Thus, between the RF frond-end and the AP for each antenna, an ADC is added. Beamforming is performed on the processor, represented as a DF process.

6.2.1.1 **Simulink model**

The Simulink model of the simple beamformer is shown in figure 6.6. The channel implements a variable time delay for each transmitter (Tx) and receiver (Rx) pair. For each transmitter the source signal is multiplied with a directional dependent gain in the direction of each receiver. For simplicity, the directional gain is 1 in all directions, i.e. omni-directional antennas are used. A multi-dimensional matrix of signals is used between the blocks; one dimension for the sources (each has a single transmitter), a second dimension is used for the signal to each receiver (besides

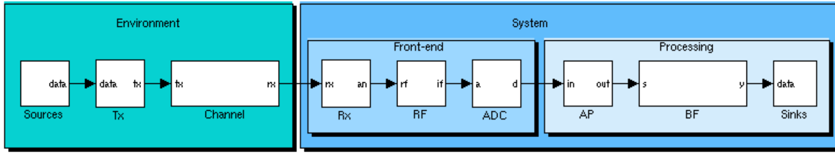


Figure 6.6: Simulink simple beamformer model

a dimension for time). At the receiver side the signals from all transmitters to that receiver are summed, thereby reducing the dimension of the matrix of signals. The RF and AP blocks only pass-through the signal because we use a baseband equivalent model, i.e. up and down conversion are replaced by a complex valued signal representation. The ADCs use a ZOH to sample the signal, no quantisation is used. The beamformer applies a complex multiplication for beamsteering with a fixed steering angle. At the beamformer a dimension for each beam is used, but for presentation purposes we will use only a single beam.

6.2.1.2 UniTi model

The UniTi version of the simple beamformer model is illustrated in figure 6.7. The figure also shows the used structural hierarchy. At the top level of the design the model consists of the environment followed by the beamforming system:

$$model = environment \triangleright system$$

Below we will explain how the environment and the system are formalised, and how the model is simulated over a sequence of time steps.

Environment First of all, the $environment$ models one or more sources together with their transmitters that send the signals. We will assume that a *source* generates the signal as it is sent by a transmitter. Suppose

$$sources = (src_1, src_2, \ldots)$$

that is

$$sources = src_1 \parallel src_2 \parallel \ldots$$

The fact that a source *generates* a signal formally means that each src_j is a function such that $src_j()$ is a *signal*. Each antenna receives the signal from each source through a *channel* for which we now model the time delay. A channel ch_{ji} from source j to antenna i in fact is a *signal transformation* which delays the signal s_j:

$$ch_{ji}(s_j) = delay_{\delta_{ji}}(s_j)$$

Since the DoA d_j of source j and the position p_i of antenna i are known, δ_{ji} can be calculated as on page 151.

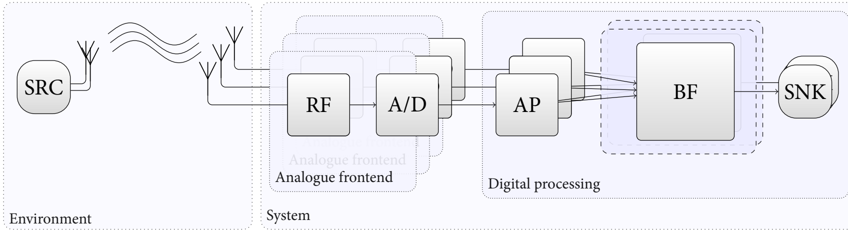


Figure 6.7: UniTi simple beamformer model

The *environment* now is the total of the sequential compositions for all j, i of src_j and ch_{ji}:

$$env_{ji} = src_j \triangleright ch_{ji}$$

Let

$$chs_i = ch_{1i} \parallel ch_{2i} \parallel \dots$$

then

$$sources \triangleright chs_i$$

delivers all signals for antenna i. Now let

$$channels = chs_1 \parallel chs_2 \parallel \dots \parallel chs_N$$

then the total environment is

$$environment = (\parallel^* sources) \triangleright channels$$

where $\parallel^*$ creates as many copies of *sources* (in parallel) as needed.

System The *system* consists of (i) the parallel composition of a *frontend* for each antenna, (ii) a part which *processes* the outputs from these antenna frontends and produces beams, and (iii) the parallel composition of a sink (*snk*) for each beam to plot the result, i.e.

$$system = (\parallel^* frontend) \triangleright processing \triangleright (\parallel^* snk)$$

Concerning (i), the frontend of each antenna, we remark that it consists of a receiver rx, an rf frontend and an adc, i.e.

$$frontend = rx \triangleright rf \triangleright adc$$

The receiver rx gets delayed signals from each source, i.e. rx is a function which adds a sequence of delayed input signals into a single output signal as defined in section 5.5:

$$rx = \bigoplus$$

Note that we consider rx here as part of the system under design, and that the signals from the environment are combined at the receiver antennas. Note also that

the rf and adc for each antenna are the same. Here we will assume that the signal transformation rf only passes through the signal, i.e. rf is the identity function, though for later refinements of the model, rf can be defined differently.

The adc (with sample period h) is defined as in section 5.1.2:

$$adc_h(s) = t \mapsto s(\lfloor t/h \rfloor \cdot h)$$

The frontend is identical for each antenna. That is possible because the influence of the position of the antenna is already accounted for in the delay of each input signal, as discussed above. To combine the frontends of all antennas into one component, we simply have to compose as many frontends in parallel as there are antennas.

Concerning (ii), the processing part consists of antenna processing (ap) and beamforming (bf), where ap is dealt with in the DT domain and beamforming in the DF domain. As with the rf frontend we assume that ap is the identity function, though it might be defined differently. There are as many ap components needed as there are antennas and their outputs are input for the beamforming operation.

The definition of bf differs from the definition in section 6.1 where bf was a signal in the CT domain, whereas now it is a function from input tokens to output tokens. Thus:

$$bf_{\vec{w}}(\vec{x}) = \frac{\vec{w}^* \bullet \vec{x}}{N}$$

In order to wrap the function bf in a DF component, we first have to apply the $\boxdot$ operator and then we have to initialise the internal state with an empty state using the $\Uparrow$ operator (see section 5.1.3).

The total processing chain now becomes:

$$processing = (\;\|^*\; ap) \;\triangleright\; ((\boxdot\, bf_{\vec{w}}) \Uparrow [\,])$$

where $\vec{w}$ is the same steering vector as defined in section 6.1.

The function bf calculates a single beam. Without going into details, we mention that in case more than one beam has to be formed using the same antenna signals, we defined a composition operator $\triangleright^*$ which duplicates the input signals to match the number of beamformers. Note that duplication of input signals is not the same as parallel composition of signal transformers.

Finally, concerning (iii), the snk components plot the signal from each beam as a side-effect and returns a vacuous output.

Simulation The model as derived above contains components in the CT domain ($sources$, $channels$, rx, rf, adc), in the DT domain (ap), and in the DF domain ($(\boxdot bf_{\vec{w}}) \Uparrow [\,]$). The sequential composition operator takes care that the various domains are integrated, e.g., by embedding the ap component defined in the DT domain in a CT domain component (see section 5.3).

Again, all definitions above can be straightforwardly reformulated in Haskell. Since the composition operators are also defined in Haskell, the whole model can be simulated by evaluating it as a single Haskell program.

In order to evaluate the model, it first has to be applied to the empty signal $()$. Since in this case there are many sources, a nested structure of $()$'s has to be provided. Still, we will denote this vacuous input by $()$. As on page 155 a simulation calculates the results for a sequence of time steps $\mathbb{T}$:

$$simulation = results_{model\,()}(\mathbb{T})$$

The sequence of results is defined as

$$results_f(\mathbb{T}) = y : results_{f'}(\mathbb{T}')$$

where

$$(y, f') = f(t)$$

as explained in section 5.4.3. As before, t is the first time step in $\mathbb{T}$, and $\mathbb{T}'$ consists of the remaining time steps in $\mathbb{T}$.

Radiation pattern As an example of the flexibility of the definitions, we reuse the components from the definitions above to generate radiation patterns. This is achieved by calculating the transfer function over all angles.

The radiation pattern is calculated by setting the (complex) source signal to $1\,e^{j\,0}$ and calculate the result over all source angles (α, γ) with a fixed steering angle (α_0, γ_0):

$$P(\alpha, \gamma) = \left| bf_{(\alpha_0, \gamma_0)}(\vec{x}) \right|$$

where

$$\begin{aligned} \vec{x} &= (src \;\triangleright\; channels \;\triangleright\; (\,\|^{*}\,(rx \;\triangleright\; adc)))\,()\,(0) \\ src\;() &= t \mapsto \left((\alpha, \gamma), e^{j \cdot 2\pi f_c t}\right) \end{aligned}$$

Note that the source *src* now is a function which yields the complex signal including its corresponding DoA (α, γ).

Figure 6.8 shows a radiation pattern of a 3 by 5 element array, with positions that are randomly shifted slightly from their original positions, steered to 80° elevation. This results in a flattened beam and somewhat chaotic side lobes. Figure 6.9 show the radiation pattern of a 6-element ULA located along the x-axis. Note the array is only directional in one dimension.

6.2.1.3 Comparison

The graphical block-diagram representation of Simulink is intuitive and has simple semantics. However, such a graphical representation is less flexible when the model is changed during development. A textual representation is easier in that respect, but keeping an overview of the model is more difficult. Therefore, a lot of structural hierarchy is used in the above UniTi model. Furthermore the composition operators and aggregate data structures increase flexibility. Although the definitions are cryptic, they are completely independent of the number of antenna

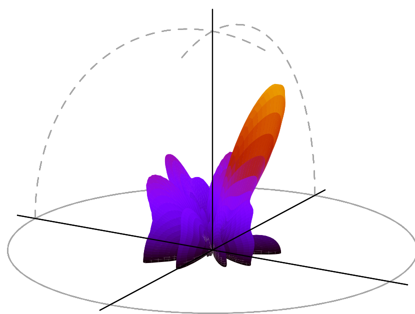


Figure 6.8: 3D radiation pattern for a 3×5 array steered to (110°,60°)

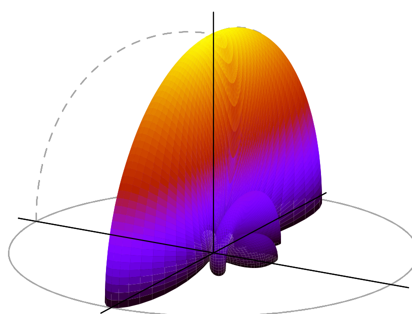


Figure 6.9: 3D radiation pattern for a 6-element ULA

elements and sources, while specifying such a model graphically would be quite laborious and inflexible for more than a few antennas and sources. For that reason multi-dimensional arrays are used as a data-structure in the Simulink model. The representation of the operations on the signals in Simulink is, however, not as explicit as the aggregate operations in the UniTi model, making it more difficult to understand the model. In addition, with the UniTi model the type checker ensures that the correct dimension of the aggregate structure is used (each dimension has a different type for identifying the signal), while in Simulink mistakes are easier.

Of major importance is that UniTi uses exact time delays, while Simulink models uses interpolation for simulation (see section 4.3). The UniTi model is therefore more accurate but also more efficient. This is especially relevant for this case study, as a time delay is used for each transmitter receiver pair.

As an example we simulate a 5×5 planar array with two sources at a 20° separation in azimuth angle. Thus 2×25 time delays are used to implement the channel between the sources and antenna elements. One source is a 1 kHz cosine and the other a 4 kHz cosine; their combined result is shown in light-grey in figure 6.10. A simulation is performed, both in Simulink and UniTi, from 0 s to 4 s with a (simulation) step size of 0.01 s. The result when steering the beam in the direction of the first source thereby suppressing the second source is shown in dark-grey. In the UniTi model this is exactly the result as expected, however, the same system in Simulink has an error in the range of the step size (after the start-up effect) as shown in figure 6.11. This is because the simulation step size also determines the granularity of the interpolation in Simulink, while in UniTi it only determines when a result is calculated for the plot as explained in chapter 5.

The execution time for both models was measured using a 2 GHz Core 2 Duo with 4 GB RAM. For Simulink R2010b the execution time was measured using the Simulink profiler, while for the UniTi model the execution time was retrieved from the operating system. The DoA was used as a parameter for the UniTi simulations to ensure the results were not cached by the interpreter. The results are shown

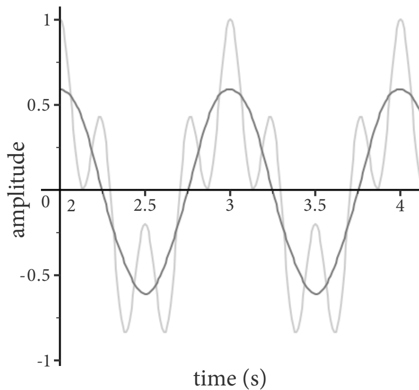


Figure 6.10: Beamformer result

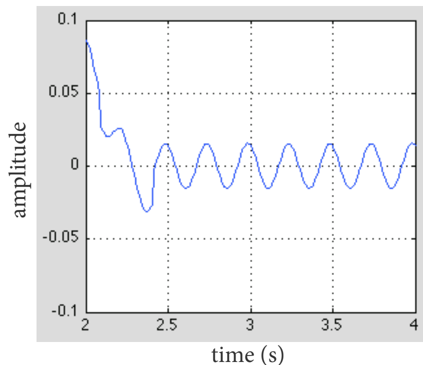


Figure 6.11: Simulink error

in table 6.1. The numbers are averaged over 10 runs after 2 warmup runs. When simulated in an interpreter, the UniTi simulation is slightly (1.06 times) faster, so even in the interpreter exact results are gained for a comparable performance. The compiled UniTi version is about 10 times faster while the compiled Simulink model is about 2.5 times faster, giving the UniTi implementation a 3.35 times speed-up compared with the Simulink implementation. For both, the execution time scales about linearly with the number of antennas.

Table 6.1: Simple beamformer execution times

	Interpreted	**Compiled**
Simulink	0.987 s	0.394 s
UniTi	0.927 s	0.118 s
Speed-up	1.06	3.35

6.2.2 *Adaptive beamformer*

For the adaptive beamformer, the simple beamformer is extended with E-CMA as adaptive beam-control algorithm. This introduces feedback in the model because of the control algorithm. Furthermore, E-CMA uses iterative updates of the steering vector and therefore has state.

E-CMA is a tracking algorithm, which needs the initial DoA of the source. By using reconfigurable hardware, the beamforming functionality can be replaced temporarily by DoA estimation. For the adaptive beamformer case we assume the initial positions of the sources are known.

The source signal is QPSK modulated, of which the fixed constellation points are exploited by the E-CMA algorithm to improve tracking of the source. As input data we use 2 bit symbols which are encoded as a constellation point by multiplying

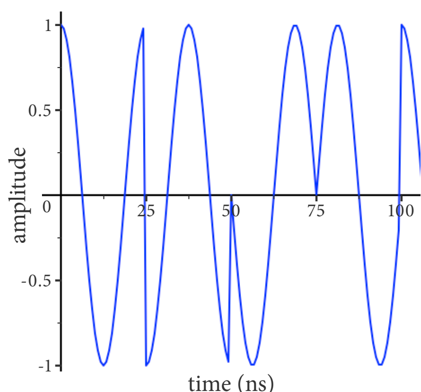


Figure 6.12: QPSK modulated signal

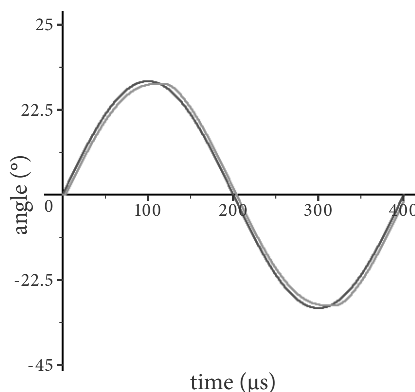


Figure 6.13: DoA of source (dark grey) and A-CMA steering angle (light gray)

with $\frac{\pi}{2}$ to get the phase of the carrier signal. Such a QPSK modulated signal (from the UniTi model) is shown in figure 6.12. These signals have a large bandwidth, because of the discontinuous changes of the signal. To limit the bandwidth typically (e.g. for the DVB-S application [28]) a pulse shaping filter is used at the transmitter to suppress high frequency components. At the receiver, a matched filter is used to suppress noise before demodulation. The same root-raised-cosine (RRC) filter is used for both the pulse shaping filter and the matched filter, and is implemented by a 25-tap FIR filter [14]. The signal is up-sampled three times because of the filter.

6.2.2.1 Simulink model

Figure 6.14 shows the system part of the Simulink model. The antenna signals after the AP and the output of the BF are used as input for E-CMA (BC) to compute the next steering vector. The current steering vector is input for the BF, so there is a delay for the steering vector breaking the feedback loop (this is implemented by letting BC store the last steering vector while computing the next one).

A variable time delay is used to implement the channel (as for the simple beamformer). Note that because the DoA of the source is changing, the delay is indeed varying as it depends on the DoA of the signal. The phase of the carrier changes at the symbol rate of 36 Msymbols/s or every 27.78 ns. The simulation step size must be smaller than this, otherwise the time delay block interpolates a signal with a phase that changes each sample. Including the three times up-sampling for the pulse shaping filter, we will use a 30 times smaller step size. The interpolation error is in the range of the step size, causing a relatively large 3 % amplitude error. However, as for the DVB-S application, the SNR is expected to be less than 16 dB. This amplitude error is acceptable (3 % amplitude error is about 30 dB SNR).

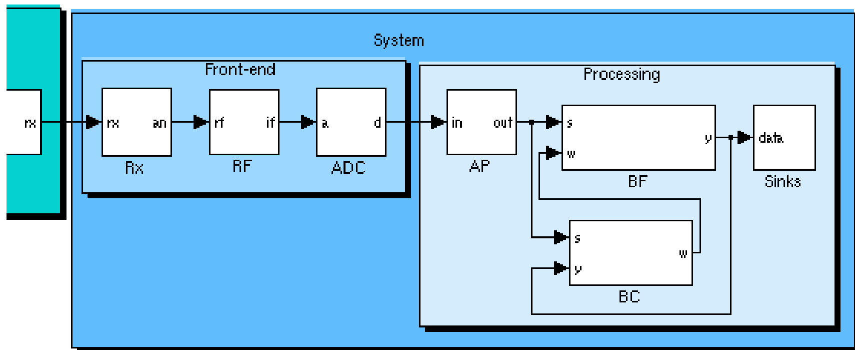


Figure 6.14: Simulink adaptive beamformer model

The pulse shaping filter reduces the bandwidth of the input signal. After filtering it can be considered a narrow-band signal and we can also approximate the channel with a complex multiplication by a phase-shift (section 2.2.2). This removes the need for up-sampling, reducing the number of simulation steps 10 times (from 30 times to 3 times, three times up-sampling is still needed for the filter). Furthermore, the approximation error is smaller; with a carrier frequency of 10 GHz and a bandwidth of 50 MHz the error is about $^{50}/_{10000}$ =0.5 %. The Simulink model with a phase shift approximation is therefore included for comparison.

6.2.2.2 UniTi model

We start with giving the definitions of a single src that replaces the src_i's in the simple beamformer model, and which will be explained below. The QPSK modulated source signal is defined in UniTi as:

$$\begin{aligned} src\,() &= rate_{dt}\,(input) \;\triangleright\; qpsk \;\triangleright\; rrc \;\triangleright\; dac \\ qpsk\,(x) &= e^{j \cdot x \frac{\pi}{2}} \\ rrc &= fir_{\vec{h}} \Uparrow \vec{0} \end{aligned}$$

where $\vec{h}$ consists of the RRC filter coefficients. The $input$ is a sequence of random 2 bit symbols, which are QPSK modulated by $qpsk$ and filtered by rrc, before being converted the analogue domain with dac.

As such, the input, QPSK modulation and FIR filter are defined in the DT domain, while the components of the environment are defined (as for the simple beamformer) in the CT domain. Therefore, the sample rate of the DT signals must be defined to connect those components, which is achieved by $rate_{dt}$ with dt the sample period. The output signal of $rate_{dt}\,(input)$ is a piecewise continuous function of time, causing the rest of the DT components to be lifted to CT components as explained in section 5.3. As a result the dac is essentially passing through the

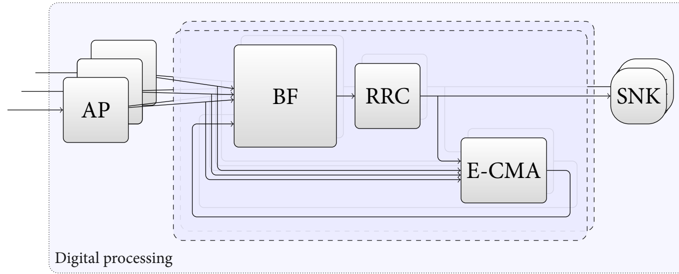


Figure 6.15: UniTi adaptive beamformer model

signal. For simplicity we have up-sampled the input symbols three times so that they have the correct rate for the pulse shaping filter[1].

The rest of the model (e.g. *model*, *environment*, *system*, etc.) is the same as for the simple beamformer, except for the processing component which is shown in figure 6.15. The signal flow diagram is the same as for the Simulink model, except that in the Simulink model the RRC filter block is included in the BF block.

In figure 6.15 the component consisting of bf, rrc and $ecma$ contains a feedback loop, preceded by ap. Thus, the structure of this component is

$$processing = (\ \|^{*}\ ap)\ \ \triangleright\ (\ \circlearrowright\ \ F)$$

with F of the form

$$\begin{aligned} F \ = \ & (\vec{x}, \vec{w}) \Rrightarrow (y, \vec{w}') \\ & \quad \text{where} \\ & \qquad (\vec{x}, \vec{w}) \rightarrowtail (((\boxdot\, bf) \Uparrow [\,])\ \ \triangleright\ rrc) \rightarrowtail y \\ & \qquad (y, \vec{x}) \rightarrowtail (ecma \Uparrow \vec{w}_0) \rightarrowtail \vec{w}' \end{aligned}$$

Here, the arrow $\Rrightarrow$ means that F in addition contains an internal state which is updated each time that F is evaluated. The definition of F can be read from figure 6.15 in a straightforward way: Between the two $\rightarrowtail$ arrows there is a component which has internal state. On the left hand side there is the input to the component, and on the right hand side there is the output of the component. During evaluation of the component, its internal state is updated. In Haskell this translates directly to the *arrow notation*[73].

The definition of $ecma$ is a discrete version of the definition used for the specification in section 6.1. In addition, state is used for the steering vector. Thus, three components in the UniTi model have state: the pulse shaping filter, the matched filter and E-CMA. In addition, there is one feedback loop for the beamcontrol algorithm.

[1] A sample rate conversion component is a little involved as it needs to save the last sample as state and repeat it n times as output, with n the rate conversion.

6.2.2.3 Comparison

For comparison, we will use a single source with random input symbols. For the array we will use a 32-element ULA with the antennas at $\lambda/2$, and the source has a DoA that changes in a sine wave motion from 0° to 30° azimuth and back, i.e. half a period of the sine wave. The symbol rate is 36 Msymbols/s, which is up-sampled three times to 108 MS/s. This results in a step size of about 10 ns for the Simulink model with a phase shift based channel and is also used as simulation step size for the UniTi model. The Simulink model with a time delay based channel has a step size of 1 ns.

The execution times of a simulation from 0 μs to 100 μs averaged over 5 simulations are shown in table 6.2. Note that the time delay Simulink model computes about 100 000 simulation steps and the other two about 10 000 simulation steps. The Simulink model with time delays takes 35 times longer than the model with phase shifts when interpreted and 5 times when compiled. This is expected as 10 times as many samples are computed. The UniTi model is about 3 times faster than the Simulink model with phase shifts (for the compiled versions), with the UniTi model using an exact time delay based implementation for the channel.

Table 6.2: Adaptive beamformer execution times

	Interpreted	**Compiled**
Simulink with time delays	826.326 s	38.686 s
Simulink with phase shifts	23.470 s	8.004 s
UniTi	18.926 s	2.951 s

6.2.3 Hierarchical beamformer

For the hierarchical beamformer, the adaptive beamformer is extended with two stage beamforming; the first stage is in the analogue domain and the second stage is in the digital domain. Furthermore, we have exchanged the beamcontrol algorithm for A-CMA (see section 2.4.3). A-CMA determines the DoA of the QPSK-modulated source signal (the same as used for the adaptive beamformer). The DoA is used as a steering angle for both the analogue and the digital stage. As A-CMA computes a steering angle, we will use a LPT as beamsteerer at both stages.

6.2.3.1 Simulink model

Figure 6.16 shows the system part of the Simulink model. We have added an analogue beamformer (A-BF) after the RF fontend and before the ADC. Both the analogue beamformer and the digital beamformer (D-BF) are phase shift based. Furthermore, we have added a beamsteerer based on a LPT below both beamforming stages (A-BS and D-BS). The beamcontrol block (BC) was changed to A-CMA and the steering angle (theta) is used as input to the beamsteerers. As such, the steering angle is converted to the analogue domain for the analogue stage using an DAC.

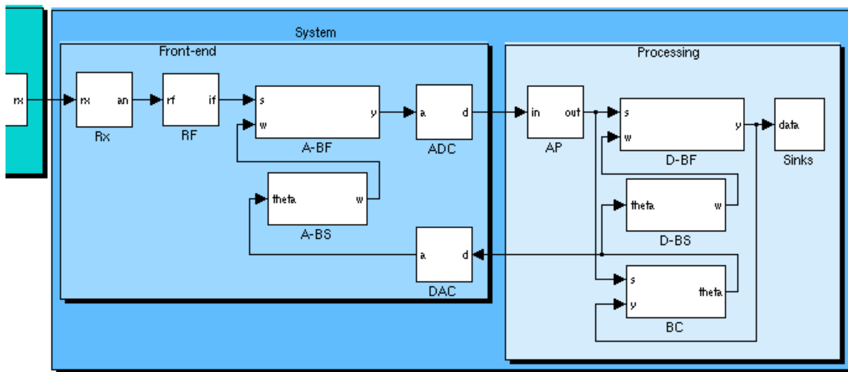


Figure 6.16: Simulink hierarchical beamformer model

The rest of the model is the same as for the adaptive beamformer, i.e. a QPSK modulated source signal and time delays for the channel are used.

6.2.3.2 UniTi model

The UniTi model of the analogue front-ends and the digital processing is shown in figure 6.17. As can be seen, the model is more complicated than before because of a double feedback loop from the A-CMA block. It is again the same as the Simulink model, except that some of the names are changed and the blocks are slightly rearranged in position to match with the definitions below. The model of the environment is the same as for the adaptive beamformer.

We will start with the definition of the digital processing part:

$$processing = (\ \|^{*}\ ap)\ \rhd\ (\ \circlearrowleft\ G)$$

with G of the form

$$G = (\vec{x}, \theta) \mapsto ((y, \theta'), \theta')$$

where

$$\begin{aligned} \theta &\rightarrowtail lpt \rightarrowtail \vec{w} \\ (\vec{x}, \vec{w}) &\rightarrowtail (((\boxdot\, bf_{\vec{w}}) \Uparrow [\,])\ \rhd\ rrc) \rightarrowtail y \\ (\vec{x}, y) &\rightarrowtail (acma \Uparrow \theta_0) \rightarrowtail \theta' \end{aligned}$$

The *processing* component is comparable with the E-CMA version of the adaptive beamformer. The differences, besides the beamcontrol algorithm, are that a LPT is used to determine the steering vector $\vec{w}$ from the steering angle θ, and that the steering vector from *acma* is duplicated; the outer one is for the feedback loop to the digital beamformer and the other one is an extra output of the *processing* component.

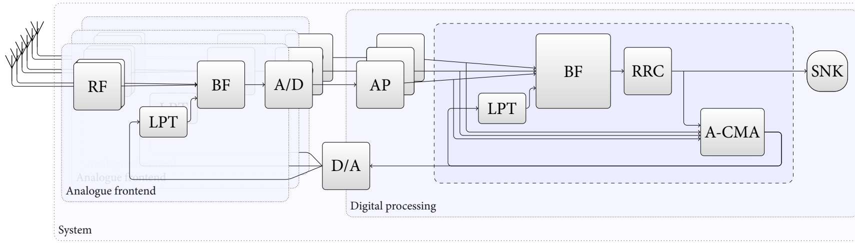


Figure 6.17: UniTi hierarchical beamformer model

The definition of the system follows as:

$$system = \circlearrowleft \; ((\; \|^{*} \; frontend) \; \triangleright \; processing) \; \triangleright \; (\; \|^{*} \; snk)$$

The *processing* component now outputs a beamformed signal as well as a steering angle from A-CMA. This steering angle is fed back to the analogue beamformer as extra input to each *frontend* (in a hierarchical beamforming the first stage consists of multiple sub-arrays, see section 2.3.4).

It is still ongoing research how to combine a feedback loop in the CT domain with state, as discussed in section 5.4.3.3. Without state, the feedback loop must be re-computed from initial situation at time $t = 0$ until the simulation time t for each simulation step. To provide a fairer comparison we have therefore moved the "analogue" beamforming stage to behind the ADC, thereby allowing state. Each frontend is therefore defined as:

$$frontend = ((\; \|^{*} \; (rx \; \triangleright \; rf \; \triangleright \; adc)) \; \| \; lpt) \; \triangleright \; bf$$

The sequence of components is similar to the frontend as defined for the simple beamformer. However, now there is a parallel composition of rx, rf and adc for each antenna from the sub-array, and this composition is in parallel with an LPT (lpt) and followed by a beamformer (bf).

6.2.3.3 Comparison

A simulation is performed with 4-elements for the first stage and 8-elements for the second stage. Therefore, the total number of antennas is 32, as it was for the adaptive beamformer. Also the same simulation scenarios is used, i.e. the source has a DoA that changes in a sine wave motion from 0° to 30° azimuth and back.

The output of the A-CMA algorithm (the steering angle) of the UniTi model is plotted over time in figure 6.13. The DoA of the source is shown in dark grey and the steering angle from A-CMA is shown in light grey. As can be seen, A-CMA correctly follows the DoA but slightly lags behind.

The execution times of a simulation from 0 µs to 100 µs averaged over 5 simulations are shown in table 6.3. Note that A-CMA is a more complex algorithm than E-CMA. As a consequence execution times are longer for all simulations, especially for the interpreted Simulink simulation with time delays and the UniTi simulations. UniTi has also become slower than the Simulink simulation with phase shifts. We expect this is the case because the UniTi model uses a naive implementation of matrix multiplication, while Simulink is highly optimised for matrix operations. Nevertheless, UniTi still offers a model using exact time delays at a reasonable increase in execution time (only 4 % for the compiled versions), and is much faster than the Simulink model using time delays (about 46 times for the interpreted version and 6 times for the compiled version).

Table 6.3: Hierarchical beamformer execution times

	Interpreted	**Compiled**
Simulink with time delays	2455.014 s	78.502 s
Simulink with phase shifts	29.142 s	11.954 s
UniTi	53.390 s	12.425 s

6.3 Partitioning

In chapter 3 we found that beamforming is the most computationally intensive part of the system and that it therefore must be partitioned over multiple cores. Figure 6.18 illustrates this transformation. Beamcontrol processing is performed far less often (about 1×10^6 times less often for the DVB-S case, i.e. 50 MS/s versus 50 Hz array dynamics), so we expect that beamcontrol processing is performed on a single tile as there is enough time for computation and communication (for low cost algorithms such as E-CMA and A-CMA).

In section 2.3.4 we discussed hierarchical beamforming; beamforming is performed in multiple stages while the beamsteer correction is distributed over the stages. This approach is thus very suitable to partition the beamforming operation.

In section 5.5 we defined exactly such a model transformation, a divide-and-conquer approach which distributes a distributive operation over a reduction operation. In case of PS based beamforming, beamforming is defined as the dot-product of the antenna signals ($\vec{x}$) with a correction vector $\vec{w}$ (see above). Thus, the distributive operation is a complex multiplication (with the steering vector weight) and the reduction operation is a sum.

A nice property of this model transformation is that each part in the partitioning performs the same functionality. The transformation on the beamformer is then defined as:

$$distribute_n \ (bf\,(\vec{w}, \vec{x}))$$

with the definition of *distribute* from section 5.5. Note that this definition is independent of the number of antenna elements. All antenna signals and steering

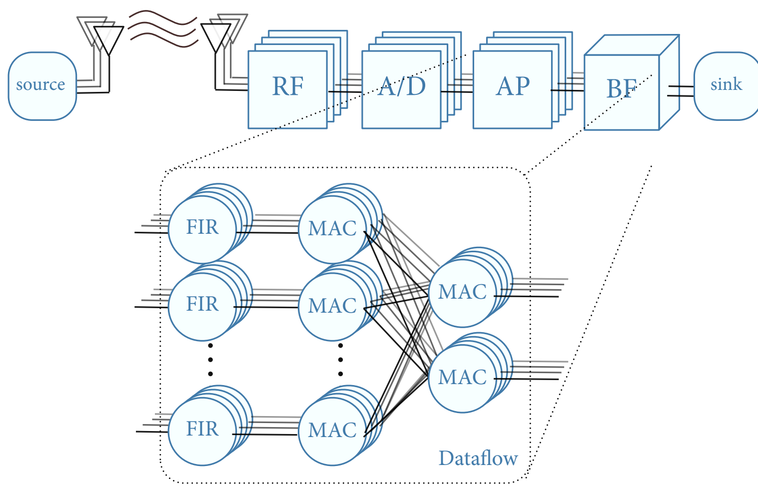


Figure 6.18: Partitioning

weights are split into parts of size n resulting in m sub-vectors. These m parts are beamformed with the first element of the sub-vector as reference, so the steering vector weights are normalised to the first element. The results of these beamformers are then recursively split into parts and beamformed in turn, with as weight vector the weights of the first elements of the sub-vectors. As here bf is a DF component, the result is a number of bf components for beamforming the sub-vectors and the later stages.

6.3.1 *Granularity*

As the partitioning of the beamformer is now parameterised over the number of elements (n) to beamform per part, the next step is to determine a suitable n. This is achieved by assigning costs to computation and communication for the target architecture. The cost can for example represent the price, the energy efficiency or maintainability, i.e. some combination of non-functional requirements, and is very dependent on the application and the specific situation. As an example, we will base the cost on resource usage in terms of number of clock cycles for computation and number of connections for communication; multiplication has a cost of 10 and addition has a cost of 1, the communication cost is equal to the number of inputs and outputs of the component.

Single tile Suppose a beamforming system with 64 antenna signals as input and a single beam as output is partitioned. The simplest architecture consists of a single tile, i.e. no partitioning is performed. For a single tile architecture, the computation cost is $63 * 10 + 63 * 1 = 693$ and a communication cost of $64 + 1 = 65$.

Table 6.4: Partitioning cost

	Single tile	Fully partitioned	Constrained
Computation cost	693	693	693
Communication cost	65	189	105
Tiles	1	64	21
Computation cost per tile	703	11	33
Communication cost per tile	65	3	5

Fully partitioned A single tiled architecture is not very scalable and is probably not feasible, so we want to distribute the beamforming over multiple tiles. The other extreme is a fully partitioned beamformer, i.e. two antenna elements are beamformed per part. As for each part the weight vector is normalised, one of the weights becomes 1 removing the need for a multiplier for that weight. Each 2-input beamformer thus consists of a single multiplication and addition. Each tile has a computation cost of $10+1$ and a communication cost of $2+1$, totalling $63*11 = 693$ and $63*3 = 189$.

Constrained A more realistic example has constraints for the partitioning set by the architecture, i.e. a tile has a maximum of computation and communication resources. Assume we constrain each tile to a computation capacity of 40 and a communication capacity of 6, this would allow for beamforming of four inputs with a computation cost of $3*11 = 33$ and a communication cost of $4+1 = 5$. The function *distribute* with $n = 4$ and 64 inputs then results in 21 tiles, 16 for the first stage, 4 for the second stage and 1 for the final stage, totalling $21*33 = 693$ and $21*5 = 105$.

Evaluation The three different partitionings are summarised in table 6.4. As expected, the computation cost stays the same, but the communication cost increases with a smaller granularity (larger number of parts). Of course, a smaller granularity lowers the computation and communication cost per tile as it is one of the reasons for partitioning (the other being parallelisation).

Using a model transformation for partitioning allows us to quickly evaluate different granularities for partitioning. Moreover, this is possible anytime during development. Evaluating even these three options in Simulink is very cumbersome as it requires one to draw each tile of the solutions, because they are not easily captured in block-diagrams. With the distribution function, we can transform the solution from one with a single tile (with $n = 64$), to one with many tiles (with $n = 2$) or anything in between and for any number of antenna inputs. This transformation is simply not possible in Simulink.

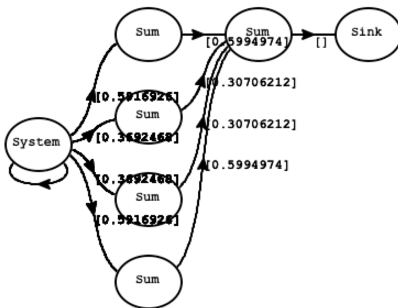


Figure 6.19: 16 antenna beamformer

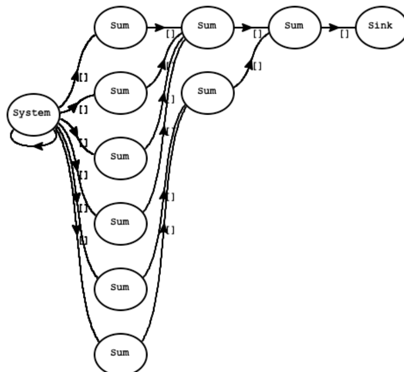


Figure 6.20: 23 antenna beamformer

6.3.2 E-CMA on a tiled architecture

Returning to our case study, consider the adaptive beamformer with the E-CMA beamcontroller is to be partitioned for a tiled architecture. The beamformer is intended for the DVB-S application, so the data rate is 50 MS/s.

Assuming the tiled architecture presented in section 3.3.3, which consists of tiles which can process 200 M ops and a 400 MB/s NoC, we find that we can beamform 4 antenna signals per tile. For 16 antennas this results in a hierarchical beamformer with 5 (DF) processes (shown in figure 6.19) and thus 5 tiles. For 23 antennas the beamformer consists of 9 processes, shown in figure 6.20, which for example matches a 3×3 grid of tiles well, leaving some processing capacity for the beamcontrol calculations. Both of these partitionings are generated automatically using the *distribute* transformation with n the partitioning granularity (the number of elements per part), only the number of the antennas are changed. We found in section 2.3.1 that 256 antennas is a feasible array for the DVB-S application, requiring 85 tiles on this architecture.

For the case study, we experiment with an existing realised platform, the LEON SoC platform presented in section 3.3.2. The LEON SoC platform consists of three Montium processors and a NoC. Each Montium has 5 ALUs and runs at 15.84 MHz on the FPGA prototype realisation, however, we expect an ASIC realisation to run at about 200 MHz. Each ALU has three levels; the first level can compute four additions, subtractions or logic operations, the second level can compute a MAC operation, and the third can perform a butterfly operation. See appendix B for an overview of the Montium architecture. The Montium can perform one complex multiplication per cycle, so 4 antenna signals can be beamformed per processor (tile). With three tiles and 4 inputs per tile the maximum array size is 8; the first stage is performed on two Montiums (4 elements each) and the second stage on the third Montium (leaving some computation capacity for e.g. beamcontrol processing). This is by far not sufficient for the DVB-S application.

6.4 Mapping

During the mapping phase the parts of the partitioned application are assigned to tiles on the tiled architecture. We have partitioned the beamforming to use the maximum computation and communication capacity possible in order to reduce the data rate as soon as possible. The mapping is therefore relatively straightforward; as each part is partitioned to use the maximum resource capacity of a tile, each part is assigned to its own tile. For more dynamic automated run-time mapping in case of multiple changing applications on a tiled architecture see [45, 97].

For the mapping and implementation step, we will continue with the adaptive beamformer case for the DVB-S application on the LEON SoC platform. Besides the beamformer and beamcontroller, a RRC matched filter for QPSK modulated signals is included as the cost function of E-CMA is based on the QPSK modulation points. The input data for the beamformer is generated by the UniTi model.

The LEON SoC can only perform beamforming for an 8-element array, as there are only three Montiums available and we also need to compute the second stage and E-CMA on the third Montium. In addition, the matched filter must also be computed. With only four clock cycles per sample, this must also be performed on dedicated tiles, which are not available on the LEON SoC. Therefore, the beamforming operation and the filtering are time division multiplexed, i.e. the operations are alternated during execution. This further reduces the input data rate of the beamforming application supported on the platform. Nevertheless, we will use this platform for mapping and implementation as proof of concept and to evaluate an actual implementation on prototype hardware.

6.4.1 *Assignment of kernels*

The processing of the adaptive beamformer is partitioned to processing kernels and encapsulated as DF processes. Next, these processes are assigned to tiles.

Beamformer The beamformer was partitioned into three DF processes in the previous section. The two processes of the first stage are mapped on two dedicated Montiums, requiring 4 clock cycles for the 4 inputs each. The second stage beamforms only the two results from the first stage, requiring 1 clock cycle. This is mapped to the third Montium.

Baseband processing The baseband processing (matched filter) uses a separate filter for the real and imaginary parts of the complex signal resulting from the beamformer. Each filter is implemented as a process on a separate Montium performing a 25-taps RRC FIR filter. The filter is executed after the beamformer for each beamformed sample. An N-taps FIR filter can be mapped on the Montium in $N/5$ clock cycles [44]. Hence, each of the two filters can be executed by a Montium in $25/5 = 5$ clock cycles.

Montium 1 | Beamforming | Matched filter | $\vec{w}$ update
Montium 2 | BF | Extended CMA (E-CMA)
Montium 3 | Beamforming | Matched filter | $\vec{w}$ update
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Figure 6.21: Scheduling of the processing blocks on 3 Montiums

Beamcontrol E-CMA is used for beamcontrol processing and includes the beam-steerer. E-CMA is more complex than the beamforming operation, but as we have already mentioned, it is computed less often. We have verified that the update rate of E-CMA can be decreased a few hundred times in relation to the sample rate before symbol errors start to occur and the algorithm becomes unstable. Therefore we will use an update rate of $\rho = \frac{1}{250} = 0.004$ of the antenna sample rate. As it is computed much less often, we can map the beamcontrol processing to a single tile.

6.4.2 Scheduling

The scheduling of the beamformer and the matched filter is shown in light grey in figure 6.21. For each signal vector from the antennas the processing takes 10 clock cycles on the Montium. After executing the beamforming operation 250 times E-CMA is executed. The execution of E-CMA is illustrated in dark grey in figure 6.21. As we will find out below, the implementation of E-CMA requires 21 clock cycles, while updating the steering vector requires another 4 clock cycles ($\vec{w}_{1\ldots4}$ is calculated by one Montium while the other calculates $\vec{w}_{5\ldots8}$). Since the Montiums communicate via the NoC which is clocked at a 3 times higher frequency, the output sent from one Montium to the next is already available at the next Montium clock cycle. Therefore, communication does not introduce latency in the schedule.

The schedule shows the utilisation of the Montiums is not optimal. Although the schedule could be improved, the main problem is a dependence of the matched filter on the beamformer in combination with a small number of tiles, and the limited amount of computations in the second stage. Pipelining the beamformer and the matched filter would, in the case of consecutive beamforming operations, reduce the number of free slots from 11 to 8 (the matched filter is moved 1 slot to the left in figure 6.21). E-CMA can be pipelined; it is started after the first clock cycle of the matched filter. This is possible because the last 5 taps of the FIR filter are calculated first, such that the filter output can be obtained after one clock cycle already. In the remaining 4 cycles, the other 20 filter taps are processed. However, as E-CMA is only executed once every 250 samples and as the rest of E-CMA can not be parallelised further (we will find below that an iterative algorithm is used), this has a limited effect.

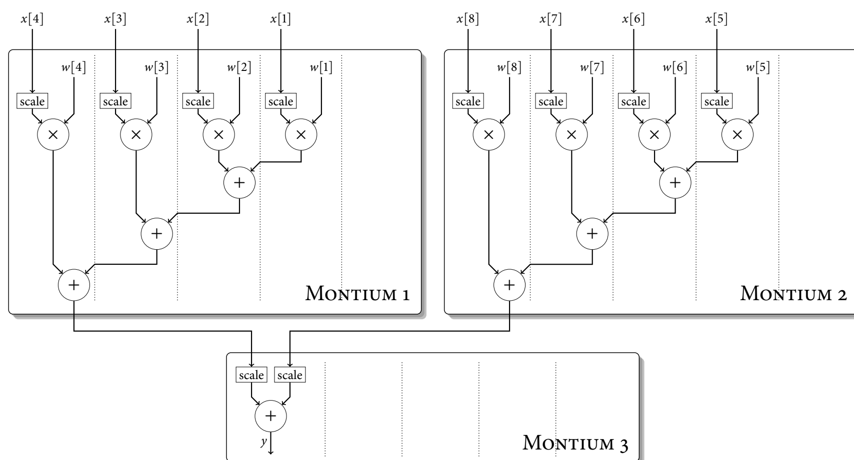


Figure 6.22: Mapping of the beamforming operation performed by 3 Montiums

6.5 Implementation

We have implemented the adaptive beamformer on a FPGA prototype realisation of the LEON SoC platform (section 3.3.2). Because the Montium processors on the LEON SoC platform only operate at 15.84 MHz, the sample rate of the antenna signals is lowered to 1.5 MS/s (complex).

Ideally code generation would take the definitions of the functionality of each DF process to generate an implementation for each of the cores of the tiled architecture. With code generation as a model transformation (i.e. the compiler is the model transformation), this would give us the means to verify that the implementation is correct and to perform design space exploration with different implementations (i.e. for different kind of processors on a heterogeneous platform). Unfortunately such a model transformation is not developed and developing it is complex and might not even be possible for e.g. the Montium, so instead we will provide a manually generated implementation of the beamforming, matched filter and beam-control operations [95].

6.5.1 *Beamformer*

The beamforming operation requires one complex multiplication and one addition per beam per sample per antenna. A complex multiplication on the Montium is implemented using four of the five MAC units at level 2 of the ALUs. Level 3 of the ALUs is used for the addition. The final mapping is shown in figure 6.22. Scaling of the input signals is used at each stage so that the dynamic range at the output is the same as at the inputs.

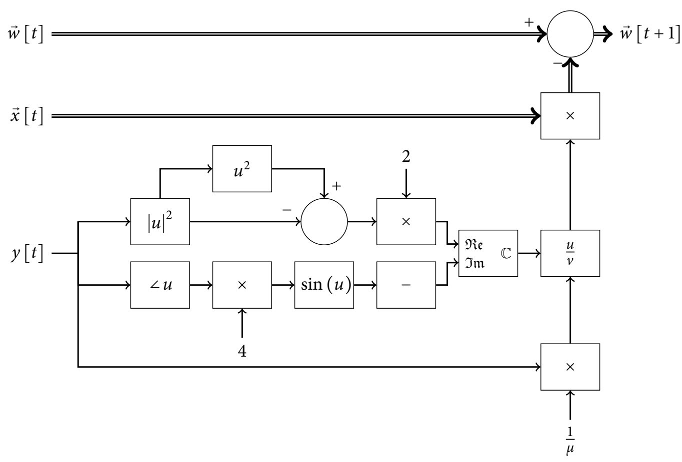


FIGURE 6.23: Block diagram of E-CMA

6.5.2 *Baseband processing*

The matched filter is implemented using two 25-taps FIR filters for the real and imaginary parts of the beamformer output. A FIR filter tap maps directly on the level 2 MAC unit. Therefore with 5 ALUs 5 filter taps are calculated in one clock cycle, which is performed 5 times for the matched filter.

6.5.3 *Beamcontrol*

The beamcontrol algorithm is more difficult to implement on the MONTIUM. We give a summary of the implementation presented in [95, 104]. The formula for E-CMA is repeated here (with discrete inputs x):

$$\vec{w}[t+1] = \vec{w}[t] \; - \mu \cdot \frac{2 \cdot \left(|y[t]|^4 - |y[t]|^2\right) - j \cdot (\sin(4 \cdot \angle y[t]))}{y[t]} \cdot \vec{x}[t]$$

A block-diagram representation is shown in figure 6.23. Note that the complicated part of the equation only consists of scalar operations.

We are using the 1.15 fixed-point arithmetic modus of the MONTIUM for the implementation of E-CMA, meaning values are in the range of $[0\dots1\rangle$ and need to be scaled at various places in the algorithm. Scaling is implemented efficiently using the level 1 shifters of the ALUs, limiting the scaling to powers of two, however. As the QPSK signal has a magnitude around 1 which can become larger than 1 because of noise, the input signals are scaled by a factor 1/2 (otherwise magnitudes larger than 1 are saturated causing problems for E-CMA as the gain can not converge to 1). The results of both $|u|^2$ and u^2 are scaled by a factor 2 to include the input scaling into the E-CMA algorithm. As the angle ranges from -π to π, it is scaled

by π for both the arc-tangent used for $\angle u$ as the $\sin(u)$ computation. This maps the angle from $[-\pi \ldots \pi\rangle$ to $[-1 \ldots 1\rangle$ matching 1.15 fixed-point values and thereby allowing overflowing of the angle values. Furthermore, analysis of the weights of the steering vector $\vec{w}]$ show that their value can go up to about 6. Therefore, the normalisation factor $\frac{1}{N}$ of the beamforming operation is applied to all the weights of $\vec{w}$ to keep their magnitude smaller than 1 (giving the same results as applying the factor to the beamformer output).

Most operations can be implemented on the Montium ALUs directly, except for the coordinate transform operations ($|u|$ and $\angle u$, for which we use the coordinate rotation digital computer (CORDIC) algorithm), the sine computation, and the complex division (for which we use lookup tables). For completeness, these implementations are presented in appendix B. Of those operations, the sine computation has a limited accuracy of 10 bit because it uses a lookup table (LUT). For E-CMA with QPSK modulation, this is accurate enough as the constellation points are 45° apart. For a larger number of phase constellation points the accuracy may no longer be acceptable and a larger memory or a CORDIC implementation must be used. Furthermore, the complex division requires a scale factor so that the division does not saturate for the Montium's fixed-point representation. Therefore, the multiplication with μ (0.05) and the scaling of the weight vector (1/8) are used as a scale factor. For this scale factor, the complex division saturates if $|\vec{v}| < 0.08$ (see appendix B). As for E-CMA the denominator $\vec{v}$ equals y and since E-CMA is used to steer $|y|$ to 0.5, the probability of a lookup of one of these saturated values is very low.

The total number of clock cycles required is 21: 16 for the CORDIC algorithm, 2 for the complex multiplication, and 3 for the rest of the operations. The scalar result is sent to the other 2 Montiums, used by each to update half of the steering vector. The update consists of a complex multiplication and addition using 1 clock cycle per weight and implemented using level 2 of the ALUs.

6.6 Results

A system design for a generic beamforming platform was developed using the UniTi design flow. Starting with a simple beamformer, which was extended with adaptive beamforming and hierarchical beamforming, the design was developed from specification to partitioning. The adaptive beamformer was further developed all the way to implementation on a tiled architecture. The specification was divided into sub-components during the co-design phase, after which it was compared with Simulink. Next, the beamformer was partitioned over multiple tiles. During these model-based design steps a single model was developed and refined using the domains and model transformations provided by UniTi. Next, a manual mapping and implementation of the adaptive beamformer on the LEON SoC platform was presented. In this section we will evaluate UniTi and present the results of adaptive beamforming on the LEON SoC platform.

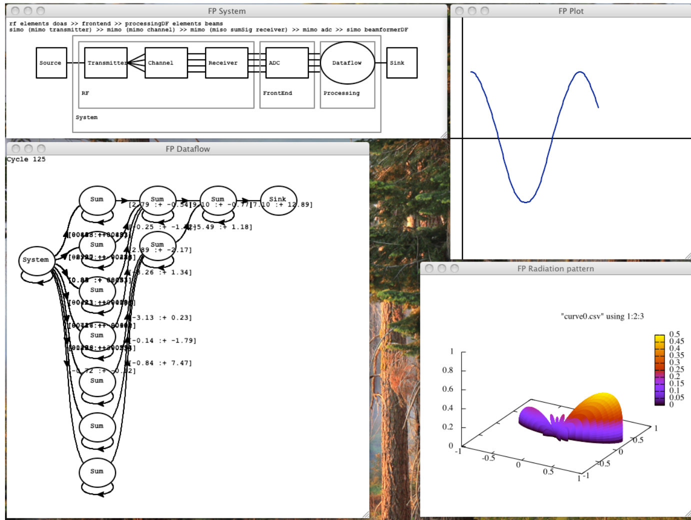


FIGURE 6.24: Framework

6.6.1 UniTi

The main evaluation criteria for UniTi are the effectiveness and usefulness of the approach. As such criteria are difficult to quantify objectively, we will present the applicability and flexibility of UniTi for the phased array beamforming case study, in order to provide an indication.

6.6.1.1 Applicability

Figure 6.24 shows a screenshot of the complete framework during the simulation of the beamforming case study with a 5×5 array, a 30° steering direction and two sources, one of which is filtered away. It shows the results of a simulation of the CT, DT and DF domains in a single model, including structural hierarchy in the system overview and the processes during execution of the DF model. The model is executed for simulation, allowing step by step evaluation of the behaviour of the system. Additionally a radiation pattern shows the current steering direction of the beamcontroller.

Performance The model shown in figure 6.24 is the same model as used for the simple beamformer during the co-design step. We have already shown that the UniTi model is at least as computationally efficient as an equivalent Simulink model (and 3 times faster for the compiled version), while providing exact simulation of the environment. To increase the accuracy of the Simulink model, the time step must be reduced making it even less efficient as the model is evaluated more often than the UniTi model. As a result, UniTi is much more computationally efficient for the adaptive and hierarchical beamformer than a Simulink model us-

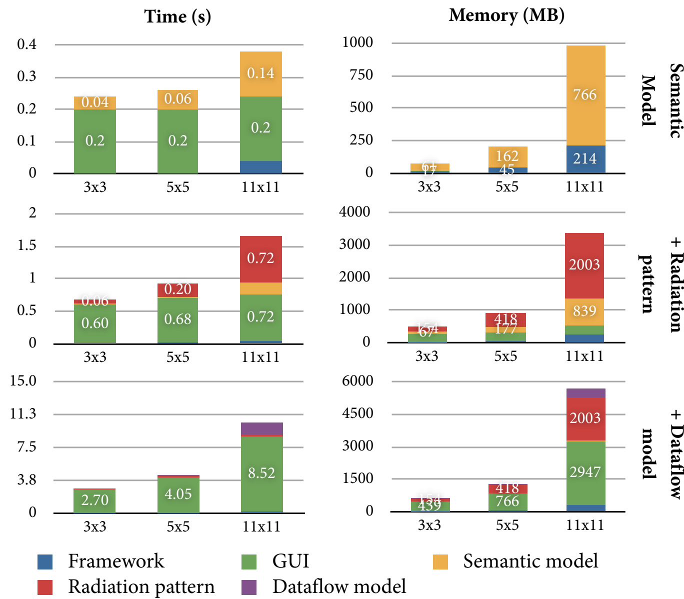


Figure 6.25: Profiling results (time (s) and memory (MB)) for the simple beamformer case study on a 2 GHz Core 2 Duo with 4 GB RAM.

ing time delays. It is also more efficient than a Simulink model implemented with phase shifts and using the same time step for the adaptive beamformer case. For the hierarchical beamformer, the execution time of UniTi is about twice as long and the compiled version takes about 5 % longer than the Simulink model with phase shifts. However, in these cases UniTi still has the advantage of supporting exact time delays.

Profiling results for an increasing number of antennas (3×3, 5×5 and 11×11 arrays) are shown in figure 6.25. The top two graphs show the results of only the CT and DT version of the model (with the beamformer in the DT domain). The next two graphs include the radiation pattern, which is computationally expensive. The final two also include the dataflow model (by lifting the beamformer function to a DF component), including the visualisation of the DF processes. We can see that the memory requirements grow faster with more antennas than the processing requirements. This limits the simulation to a few hundred antennas for a 2GHz Core2 Duo with 4GB RAM. Further, the radiation pattern calculation becomes dominating with larger arrays. Note that the instantiation of the wxWidgets toolkit for the graphical user interface (GUI) has a relatively large but fixed processing overhead compared to the model. With the dataflow model the GUI becomes dominating and is also dependent on the array size. This is because of the redrawing of the Bezier curves and the channel contents of the dataflow model during simulation, which uses a naive implementation and is therefore the first candidate for optimisation. As the framework was developed as a proof of concept, we expect there is ample room for improvement in efficiency.

Table 6.5: Code size [lines]

	Framework	Case
Support	811	
GUI	411	
System model	141	438
Radiation pattern	10	46
Dataflow model	1143	135
Total	2516	619

Designer productivity Designers can efficiently and easily extend a model with more detail by just a few lines of code, as we have seen for the three beamforming cases during co-design. For example, we exchange the simple sine source from the simple beamformer with a QPSK modulated source in the DT domain or we add an analogue beamformer for the hierarchical beamformer. Much of the design flow is automated by the framework, i.e. multi-domain composition, communication and synchronisation for a DF model and model transformations. Table 6.5 shows the code size in lines. We see that the framework is 2500 lines, half of which is of the dataflow support. The case is about 600 lines, with the majority in the system model as this implements the functionality of the system and is re-used for the dataflow model. A large part of the code (the framework) is thus re-useable, providing the glue-logic. The additional code needed because of the design framework is very little, about 5 %.

It is difficult to estimate and compare the development time of the case study, as it was used to develop to the framework. The graphical representation of Simulink is more intuitive when developing the initial model. However, with equal knowledge of the tools, we expect the UniTi approach to be more productive because the higher abstraction level of the implementation improves flexibility (see below) making adjustments easier. Furthermore, changes are checked by the type system and transformations are defined to be correctness preserving.

6.6.1.2 Flexibility

The presented design flow of UniTi and the guidelines for specifying the algorithm using aggregate operations aim at increasing the flexibility. For example, the number of sources or antenna elements and their positions can be changed without the need to change the model, and higher-order model transformations are used for automated partitioning. This enables us to quickly evaluate design alternatives.

Automation Composition, simulation and multi-domain integration are automatically provided by the framework. Implementing the functionality is of course manual. Design decisions for dividing functionality over domains and specifying the algorithm so it can be partitioned effectively are also the designer's responsibility. Still, lifting functions to operate on multiple elements, and partitioning using data and control parallelism with such aggregate structures is automated.

Scalability Specifying the algorithm at a higher abstraction level makes it independent from the number of elements and enables automated model transformations, thereby improving the scalability of the design. The framework itself (for our case) scales linearly in performance with the number of antennas as shown in figure 6.25.

6.6.2 *Adaptive beamforming on the LEON SoC platform*

As explained, model transformations for automated mapping and code generation are not integrated or available for UniTi (yet). Therefore the adaptive beamformer was mapped and implemented on the LEON SoC manually.

6.6.2.1 Applicability

For simulation and verification, a comparable synthetic scenario is used as in section 2.4.2. A ULA is mounted on a moving vehicle, driving towards the source at 72 km/h. When driving, the vehicle is moving to the left and right with respects to the source in a sine wave motion with an amplitude of 30°, i.e. 60° peak-to-peak, and at 100 Hz. A QPSK modulated source is used with random data and signals are generated for each antenna by the UniTi model. These antenna signals are beamformed and tracked by E-CMA.

The results are verified by analysing the E-CMA cost function over time (see section 2.4.2), by plotting the constellation points of the the output signal and by comparing the demodulated output data with the input data. Figure 6.26 shows the E-CMA cost function J_{E-CMA} (section 2.4.2). The costs vary between 0.08 and 0.2, corresponding to a maximum amplitude error of $\sqrt{0.20}\approx 0.45$ and a maximum phase error of $\arcsin\left(\sqrt{0.20}\right)\approx 27°$. In practice the error will consist of both an amplitude and phase error and each individual error will therefore be smaller. Nevertheless, 27° is well within the 45° separation of the constellation points. As shown in figure 6.27, the constellation points are still clearly distinguishable. Indeed, after demodulation, the output data is equal to the input data for the simulation times we used.

When compared to the simulation in section 2.4.2, the limited 16 bit wordwidth of the Montium increases the cost and spreading of the constellation points significantly. However, the results are still good enough for robust and proper tracking of DVB-S signals with rather extreme worst-case vehicle dynamics.

Performance The 8-element beamformer with a matched filter and E-CMA were mapped on three Montiums. Beamforming requires 5 clock cycles on the Montiums, utilising 9 of the 15 slots (with one slot being a single clock cycle on one Montium, see figure 6.21). The low utilisation is caused by the limited work for the third Montium, which will improve for larger arrays. The matched filter requires 5 clock cycles, utilising 10 of the 15 slots. E-CMA requires 21 clock cycles for computing the correction factor (the scalar part) and another 4 clock cycles on two Montiums for updating the steering vector, utilising 29 of the 75 slots. E-

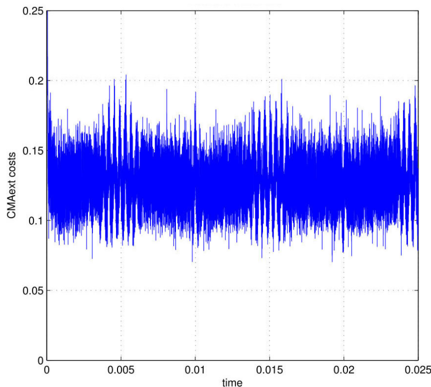


Figure 6.26: E-CMA cost function over time

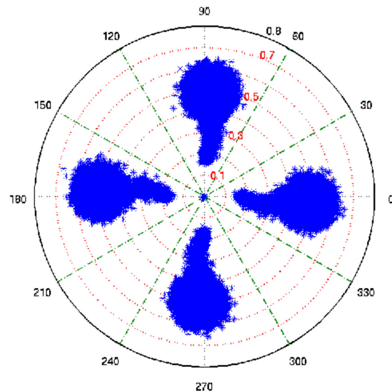


Figure 6.27: Constellation diagram of the adaptive beamformer output

CMA is partly pipelined with the matched filter improving utilisation with 8 slots. Improving utilisation further is of low priority, however, as E-CMA is only executed once every 250 samples or 2500 clock cycles. For both the beamformer and the matched filter, meaning most of the time, only level 2 of the Montium ALUs is used. So utilisation of the Montium itself could also be better; for those operations a simpler tile could be used. On the other hand, the beamcontrol algorithm has a better utilisation of the Montium ALUs and requires the additional flexibility of the Montium, which pleads for a more heterogeneous tiled architecture.

The memory requirements for the Montium are 1024 16 bit words for each the sine and complex division LUTs, and 16 16 bit words for the arc-tangent LUT. This is about 20 % of the available memory.

Designer productivity The manual implementation for the Montium is relatively complex because of the Montium's many functional units, the signal paths between the units, and the dependencies between them. It is therefore relatively time-consuming and error-prone. Being able to generate input signals for the Montiums with UniTi and to verify the results against it, is helpful for the implementation step. The UniTi model does not use fixed point data, therefore, it will be useful to implement support for fixed-point data in UniTi.

6.6.2.2 Flexibility

The LEON SoC platform consists of reconfigurable processors and a reconfigurable NoC, so it is very flexible. This flexibility is intended to be used to switch between different scenarios such as between searching using a spatial reference algorithm (section 2.4.1) and tracking using a blind reference algorithm, the latter of which we have implemented.

Automation Unfortunately, both the mapping and implementation are a manual process for the LEON SoC. The Montium and the NoC do automate the management of data streams after configuration, i.e the Montium is stalled automatically until new data arrives [104].

Scalability Because of the applied partitioning of the beamformer, the implementation scales well with the number of antennas; additional Montium tiles need to be added, but their functionality remains the same. The matched filter operates on the output of the beamformer and has a fixed complexity, so it will become smaller compared to the beamformer for increasing number of antennas, and the current mapping will suffice. The beamcontrol algorithm is more complex, but the correction factor is a scalar operation with a fixed complexity and its result is then distributed over the tiles to update the steering vector. As the same input data is used for beamforming and for updating the steering vector, this nicely exploits locality-of-reference and scales well with the number of antennas. Scaling is only limited by the need to distribute the correction factor from a single point to the rest of the tiled architecture. However, as E-CMA has limited complexity and is not executed often, it can easily be duplicated for a better distribution.

6.7 Conclusion

In this chapter we have applied the UniTi design flow and framework presented in chapters 4 and 5 to the design of a generic beamforming platform as presented in chapter 2 using a tiled architecture as presented in chapter 3. This case study is presented in three parts; first a simple beamformer is developed which focuses on modelling the environment. Next, the design is extended into an adaptive beamformer based on E-CMA for the beamcontrol algorithm, and includes feedback and state. Finally, the design is extended to a hierarchical beamformer with multi-stage beamforming in the analogue and digital domain and A-CMA for beamcontrol.

In the first step the specification of the case study is presented. The definitions are very similar to the mathematical equation from chapter 2, however, at the same time their implementation in Haskell is straightforward. As a consequence, the specification is executable. This is applied to verify the specification with plots of the beamformer output and the radiation patterns expected from the E-CMA and A-CMA algorithms.

The next step is co-design; the equations from the specification are used to define components representing the environment, the architecture and the application. The composition operators are used to combine components and for structural hierarchy. Furthermore, the environment and the analogue front-ends are represented in the CT domain, while the AP is represented in the DT domain and the beamforming and beamcontrol in the DF domain. The model is gradually developed by adding QPSK modulated source signals with an RRC filter with state, and E-CMA for adaptive beamcontrol using feedback, followed by A-CMA and

a second feedback loop. The UniTi models during this design step are compared with equivalent Simulink models. For the same simulation step sizes, the execution time is at least about the same or up to 3 times less in UniTi in the cases presented. However, even with the same execution time, we always gain accuracy with UniTi; the result of the UniTi simulations are exact, whereas Simulink introduces inaccuracies. For example, in the DVB-S case, reducing the simulation step size in Simulink with a factor 10 gives an acceptable accuracy of 3 % (30 dB SNR). In UniTi the accuracy is limited by the machine precision of about 1×10^{-13} % (300 dB SNR). With a 10 times smaller step size in Simulink, UniTi is up to forty times faster.

During the partitioning step, different partitionings of the beamformer are explored using a model transformation that exploits the associativity of addition and the distributivity of multiplication. As the beamcontrol algorithm is only executed once every few hundred samples it is not partitioned.

The mapping step assigns the processing components of the adaptive beamformer to tiles on a tiled architecture with three reconfigurable processors that is realised on an FPGA. Because of the small number of tiles, only an 8-element beamformer can be mapped, yet it verifies the scalability of the beamforming application on a tiled architecture with an implementation.

The implementation requires four clock cycles for beamforming and five clock cycles for baseband processing on two of the processors. The third processor performs a final addition and once every 1000 clock cycles the beamcontrol algorithm is executed, taking 24 clock cycles. Since only four clock cycles are available per sample, the implementation must be further partitioned for real-time operation.

The results verify that the UniTi design flow and framework can be successfully applied up to the mapping and implementation step for the design of a generic beamforming platform. The performance of the framework limits simulations to a few hundred antennas because of memory requirements, but the approach is flexible and expected to be more productive. The mapping and implementation are limited by the small number of tiles on available tiled architectures. Furthermore, the implementation takes a lot of effort. On the other hand, the implementation does provide scalability and is able to successfully execute an adaptive beamforming application.

Chapter 7

Conclusions

In this thesis we set out to advance the design of embedded systems using a model-based design approach. Looking at the trends, such an approach is considered crucial to deal with the increasing complexity of designing embedded systems. In particular we have considered a larger application on a many-core architecture, leading us to touch upon many of the trends such as requiring the modelling of multiple-domains, the inclusion of time in the model, and the need for an adaptive and flexible system that is also efficient.

We conclude that the design of future embedded systems requires support to deal with their complexity by dividing the problem into subproblems, yet at the same time requires support to integrate the various aspects of those subproblems. This integration is needed to ensure correct operations of the final design as well as to improve collaboration and interaction between different parts of the design.

Model-based design provides such an approach by modelling multiple domains in a single model and by using model transformations. We specifically considered the combination of the CT and DT domains with the DF domain. The CT domain is used to model the environment of a system and the analogue hardware, the DT domain is used to model the digital hardware, and the DF domain is used to model the software. We have found few tools supporting all these domains. Even fewer support model transformations or mathematical definitions of model components to assist model transformations. Furthermore, we have found no tools supporting the exact simulation of models with time transformations, such as time delays, or supporting separate notions of time such as simulation time, approximation time and local time. These features are important to accurately model the environment.

The UniTi design flow and modelling and simulation framework does support these aspects. UniTi provides a unified perspective on time, signal and components in multi-domain models, consisting of CT, DT and DF components. In all domains components represent signal transformations, yet the representation of signals differ. CT signals are represented by functions of time in order to support

time transformations. This is possible because components can change the time reference of the CT signal before the function is applied to a time, thereby enabling exact simulation of models containing such time transformations. DT signals are represented as values, i.e from the perspective of the DT component the input is a single value. Although this value can change over time, the DT component is not able to influence this time nor should it be able to. DF signals are represented as a list of tokens representing an update to an input channel of a dataflow process. This differs from the standard representation of dataflow models, however, this is required to unify the DF domain with the signal representation of a time-varying quantity as in the CT and DT domains. As a consequence, DF components take care of managing the contents of channels and the firing of processes. In a UniTi model, DF components are embedded into DT components and DT components are embedded into CT components for integrated multi-domain modelling. Furthermore, components are defined using mathematical definitions. Mathematical definitions facilitate model transformation by exploiting mathematical properties such as distributivity and associativity, as well as preserving correctness.

We have identified several notions of time in such models: the simulation time (of the model), the sample time (e.g. of an ADC), the approximation time (e.g. of an integration), the execution time (e.g. of a dataflow process), and the local time (e.g. for a time delay). In UniTi these notions are separated, allowing the local time of an input CT signal to be different from the local time of an output CT signal to implement e.g. a time delay. These notions of time do not have to match with the simulation time, e.g. the simulation time is automatically changed to the latest sample time, locally at the ADC component. Components that deal with changes over time, such as integration or differentiation, in the general case need a solver. In all current tools, this solver is global and uses a global approximation time step. In UniTi a solver is locally applied for the component, enabling the designer to choose a specific solver and its approximation time step, for each component individually.

Embedded systems interact with their environment, and many embedded systems perform signal processing on streaming data from the environment. As an example application, we have discussed the application domain of phased array beamforming applications. Beamforming is a relatively large application, in that it is not able to run on a single processor. Phased array systems are typically used in applications such as radar and radio astronomy. High costs have withheld their use for consumer applications, such as satellite reception and wireless communications. Therefore we have presented a generic beamforming platform that could enable such systems for consumer applications by economies of scale. However, between beamforming applications (satellite reception, radar, radio astronomy, and wireless communications) there are large differences, especially in the required array size. Thus, a generic platform must be modular, scalable and flexible to support multiple applications. In addition, the beamforming application must be partitioned for such a modular platform. A hierarchical beamformer is used to perform multi-stage beamforming, and hybrid beamforming is used to perform the first stages in the analogue domain for further cost savings. A beamforming system must also be able to search and track signals-of-interest in a dynamic environment. Many search

and track algorithms are costly in terms of processing and as such not very suitable for a low-cost platform with limited resources. Therefore, we have presented E-CMA as a low-cost tracking algorithm for PSK modulated signals. However, E-CMA is not suitable for hierarchical beamforming because it computes a steering vector for a single stage. Therefore A-CMA is presented that provides a steering angle, which can be used for all stages, at the cost of a quadratic dependence on the number of antennas instead of linear for E-CMA.

A generic beamforming platform requires scalability and flexibility, making a tiled reconfigurable architecture a good fit, for the tiles are modular and reconfiguration enables efficient reuse of the hardware. Beamforming is explored on a number of such architectures, which confirm that tiles provide scalability and reconfigurability provides flexibility. Yet, programming such architectures is not easy: applications must be partitioned, the parts can not have shared state, and the communication must be explicit. In addition, the used reconfigurable processor requires a lot of effort to program, and there are only a few clock cycles per input sample for the beamforming application, requiring a relatively large amount of communication per computation. To improve usability, dataflow models are used to represent an application for a tiled architecture, as it can represent the parts of the application as processes, and the communication between them as channels, making the communication explicit. Furthermore, the use of a dataflow model takes care of synchronisation, which is very convenient in beamforming applications which have many data streams.

A design flow has been presented, to accompany the UniTi framework, based on model transformations. The first step is the co-design step for the division of a specification to a representation of the environment, the application and the architecture. Analogue/digital co-design is used to determine which components of the environment and the architecture to model in the CT domain and which in the DT domain. Hardware/software co-design is used to determine which components of the application to model in the DT domain in hardware and which to model in the DF domain in software. The next step is the partitioning step for the division of the application on a tiled architecture. As such, the application is parallelised. Mapping and code generation provide the final implementation.

The UniTi framework supports this design flow by allowing a single unified model in the CT, DT and DF domain. Model components can be domain independent, and can thus be moved between domains. Parallelisation is supported by (mathematical) model definitions using aggregate operations, such as element-wise or reduction operations. The mathematical definitions do not unnecessarily restrict the dependencies between computations, and the aggregate operation encourages parallel definitions.

The UniTi design flow and modelling and simulation framework presented in this thesis is further explored with a case study concerning the design of a generic beamforming platform. First, a formal specification is presented, which is executed for verification. Next, the specification is refined into a multi-domain model with components representing the environment, the architecture and the application. This model is compared to an equivalent model in Simulink and found to be at least

as fast and up to 30 times faster, while providing exact simulations of time delays. Thereafter, the beamforming operation is partitioned to a hierarchical beamformer using a model transformation, and for which different granularities are explored. The mapping and implementation have been performed without UniTi support, yet it verifies the scalability of a hierarchical beamformer on a tiled architecture.

Overall, UniTi is successfully applied to the design of an embedded system. As such, we have taken a few steps forward by providing a functional design flow and framework with support for multiple domains and model transformations.

7.1 Research questions

Following the conclusions we will now address the research questions presented in chapter 1 directly:

- *What is a suitable design flow for embedded systems based on a divide-and-conquer approach?*

  A model-based design flow supported by model transformations for the co-design, partitioning and code generation steps is suitable for the design of complex embedded systems. Such a design flow requires support for modelling the environment, the architecture and the application of an embedded system in a unified approach. Such a design flow follows from the increasing interaction of embedded systems with their environment, the need for a tiled architecture to support scalability and the use of a divide-and-conquer approach to manage complexity.
- *What is required from a modelling and simulation framework to support this design flow?*

  A single model is required for the CT, DT and DF domains. The integration of these domains needs a unified perspective on time, signals and components, with support for sequential, parallel and feedback composition, as well as support for different notions of time. To support model transformations, support is needed for mathematical model definitions and applications that are defined using aggregate operations.
- *Are tiled reconfigurable architectures suitable for large high-performance applications?*

  The tiles provide modularity and scalability, but also require the applications to be partitioned in independent parts with explicit communication. The reconfigurability provides flexibility, yet programming a reconfigurable system takes a lot of effort. Thus, the use of a tiled reconfigurable architecture requires additional effort over a single core fully programmable solution (which is not feasible for such large applications), but less than a fully dedicated implementation. Furthermore, the use of dataflow models improves usability by representing a partitioned application with explicit communication, providing synchronisation, and providing analysis of the model. Finally, a design flow and modelling and simulation framework supporting such architectures is needed, as provided in this thesis.

7.2 Discussion

This thesis covers a broad range of subjects and in addition presents the design of an embedded system all the way from specification to implementation. It also combines several research areas; systems engineering, signal processing applications, computer architectures and functional programming.

This has the advantage that we have touched upon most aspects in designing embedded systems and are able to provide contributions at the boundaries of research areas. Yet, we have not been able to discuss all topics in-depth. In particular, the representation of the architecture in the model, including structural aspects and cost aspects, has been limited. Furthermore, we have found that the signal processing model, distributed concurrent applications, and mapping of applications onto a tiled architecture are all based on dataflow, while the processor tiles are still based on control flow. We would have gladly discussed dataflow processors as a reconfigurable processor for streaming application, matching very well with tiled architectures, as we have explored in [KCR:5, KCR: 16] and [114] for example.

Finally, we have presented novel perspectives on modelling time, exact simulation of time transformations, and local solvers, as well as representing and integrating signals and components in the CT, DT and DF domains. The full consequences of these choices in more than a single case study will have to be evaluated. For example; are there models that can not be represented using UniTi, and what is the numerical accuracy and stability when using multiple feedback loops and local solvers?

7.3 Outlook

The design of embedded systems using model-based design and supported by a framework like UniTi seems well positioned [13, 42, 48, 59, 68]. Several of its advantages have been presented in this thesis. Of course, there are many opportunities for future work.

Some of the work has already been mentioned during the discussion. For example, hardware aspects of the architecture, such as hardware cost, resource costs, timing, structure, etc. were intended to be included as meta-data in the models. Furthermore, the use of dataflow processors as tiles would be interesting to further explore.

For beamforming applications, the use of hierarchical hybrid arrays raises interesting questions concerning the use of different antenna configurations for subarrays, and the combination of time delay based beamforming stages with phase shift based beamforming stages.

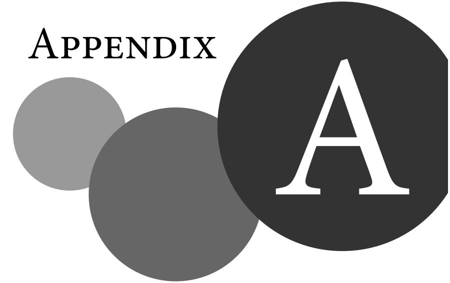
Concerning UniTi, there remains a lot to be done. The problem that occurs when feedback in the CT domain is combined with state for more efficient simulations is an important hurdle to be solved. Furthermore, we expect to be able to integrate additional domains, such as finite state machines or timed automata, because a CT signal can be changed to a (completely) different function depending

on the time, thereby representing a state change. Analysis of dataflow models is currently not supported, but should be easy to add, as we can already extract a visualisation of the model and can thus also extract different formats as required by existing analysis tools.

Usability of the UniTi tool would be improved if a graphical view of the models, as well as a design environment, complement the textual representations. This is because graphical tools are intuitive to use and common in systems design tools. However, the textual representation remains important, for example for the initial specification or for complex components.

Finally, the mapping and implementation of the design flow should be better supported by the framework. There are promising mapping tools available [45, 97] which could be integrated, or at least supported similarly to the support for dataflow analysis tools. In addition, in the NEST project [67] there is ongoing work enabling the execution of Haskell code on a 32 core architecture, thereby greatly simplifying the code generation part. There is also ongoing research to generate hardware from definitions similar to the UniTi definitions [6], using a hardware description language (VHDL) as intermediate. Supporting hardware generation using this should be relatively straightforward, skipping the mapping and code generation steps altogether.

Appendix A



Dataflow

Dataflow refers to the flow of data as contrasted to the flow of control as used in (sequential) stored-program models such as a von Neumann machine. Originally dataflow was introduced as an execution model for dataflow machines; execution is performed by *actors* on the availability of abstract data elements called *tokens* thereby reducing the task of explicit memory management [64, 108]. A directed graph representation indicates the dependencies between the actors, where the nodes of the graph represent actors and the edges represent queues of tokens. Execution of an actor is also called *firing* of a node. A node can only fire if it is *enabled* as determined by the enabling or firing rule. The firing rule specifies how many tokens are required and must therefore be available for each input of the node.

Dataflow was later introduced as a useful model for signal processing known as synchronous dataflow (SDF) [57] (also called MRDF). In this model, a node represents functions or *computations* and edges represent *signal paths*.

The third approach sees dataflow as a special case of a Kahn PN and is known as a dataflow PN. In a dataflow PN, nodes represent *processes* and edges represent unbounded FIFO *channels*, where processes are concurrent continuously executing functions and *consume* tokens from channels and *produce* tokens into channels. As such, dataflow processes can be seen as mapping sequences of inputs to sequences of outputs or functions on streams [55].

A.1 Terminology

Note that in each approach above the terminology is slightly different. Nodes represent actors, functions or processes, where an actor *performs* a computation, a function *is* a computation and a process is the action of *running* a computation. Similarly, edges represent availability of data, signal paths or channels, where channels represent both the data as the signal path. However, these terminologies are

often used interchangeably. Lee and Matsikoudis [55] argue and show that dataflow PNs are a generalisation as they can describe SDF and dataflow execution as dataflow PNs.

A.2 Dataflow model

A dataflow model or dataflow process network is a graph of nodes (processes) connected by edges (channels); data tokens are processed (computed) inside nodes and sent (communicated) from one node to another through the edges. Thus, nodes represent computation and edges represent communication. Processes consume and produce tokens by reading from and writing to channels, where tokens are atomic data elements. As processes are independent, they may not influence each other besides the explicit input and outputs, i.e. dataflow processes must be side-effect-free. Channels are unbounded FIFO token containers used for interaction between processes. Channels are of unbounded capacity, but buffers between processes are modelled by two channels in opposite directions; one models the data to be communicated and the so-called back-edge models empty space in the buffer. Only a single process is allowed to read from and write to a channel.

A process may consume and produce several tokens at a time; when there are not enough tokens available on the input edges of a node, that node will not execute (fire). The condition that enables firing is called the firing rule. Note that executions can overlap: if enough tokens are available to fire, the process directly executes even if the process is already executing.

The number of tokens consumed and produced per firing (the rates) can be variable. A SRDF graph always consumes and produces a single token, a MRDF graph has a fixed token rate at each edge. In a CSDF graph, the token rates cycle through a number of phases with fixed token rates (possibly zero) at each phase. VPDF graphs have a limited form of data-dependent token rates [115], where the token rate is determined by a parameter from an input channel. Finally, DDF graphs have fully data-dependent token rates.

A.3 Dataflow analysis

As execution depends on availability of tokens, cycles in the graph can introduce deadlock (where the processes are waiting on each other). In general (for DDF) it is undecidable if a graph will deadlock [64]. In addition, if more tokens are produced than consumed for a channel tokens accumulate requiring infinite buffers. A back-edge limits the buffer size required, but how much buffer space is needed? Too much space leaves part of the buffer unused, but too little space results in deadlock. For restricted dataflow models such as SDF, where the number of tokens consumed and produced is not data-dependent and thus unpredictable, deadlock freedom can be proven and the minimum buffer sizes required for this can be computed [115]. Furthermore, by assigning an execution time to each firing of a process, the latency and throughput of the model can be calculated.

Dataflow models have no notion of time, only ordering. For metrics such as throughput and latency to make sense, and allow them to be determined by the analysis techniques, processes in the dataflow model are annotated with execution time. Consumption and production of tokens is assumed instantaneous in the model; the time that consumption and production takes in the "real world" is absorbed by the execution time. In practice, data enters the application via one or more *source* processes (e.g., an ADC that samples data at fixed time intervals) and leaves the application via one or more *sink* processes (e.g., a DAC). The data rates for these sources and sinks are fixed and therefore, they determine the application's performance constraints.

For the analysis to be valid, the computations clustered as a single process must be side-effect free and the (worst-case) execution times must be conservative (real execution times should not be larger than the worst-case estimate) [16]. Dataflow processes and models are monotonic (order-preserving), causal (depend only on previous and current inputs) and deterministic (same output results for the same inputs, independent of the firing order) [57].

A.4 Dataflow execution

There are several execution models for the DF domain (e.g. concurrent processes, compilation of dataflow graphs, tagged token model) [57]. The most common is to implement dataflow processes as concurrent processes with static scheduling and implement the firing rules as a sequence of "read", "execute" and "write" phases, although there is no clear winner [56, 57, 64, 82, 115].

For the non-data-dependent dataflow models (SRDF, MRDF and CSDF), a static execution schedule can be determined at design time. This eliminates the need for a scheduler when executing on a single processing resource. All dataflow models have self-timed execution. Therefore there is no need for global control of the execution.

A.5 Properties

Dataflow models have a number of useful properties. Firstly, tokens in channels must remain ordered and no tokens can be lost.

Secondly, in dataflow different data rates are decoupled. A process can only execute if all required input tokens are available, otherwise it will block. Therefore, it is not important when and in what order [1] the tokens arrive; as soon as the last input required for firing is available the process executes thereby consuming the inputs. If after firing enough tokens are available to enable the process once again, it can execute straight away concurrently in the model. To limit the number of concurrent executions of a process self-edges are used, i.e. channels that loop back to the process itself. As the self-edge becomes an extra enabling condition, the next

[1] This refers to the order of arrival of tokens over the (empty) input channels, not to the order of tokens in a channel

execution of a process can only start after a previous execution has finished thereby producing one or more tokens on the self-edge.

Thirdly, because tokens are only consumed when all required tokens are available to the process, this ensures synchronisation of the processes' inputs. As said, dataflow assumes unbounded FIFO channels, but buffers are modelled with back-edges representing buffer space. A process writing to this buffer can only execute if a "space" token is available. This ensures no data is overwritten and lost. Assume process 1 produces into a buffer and process 2 consumes from it. If the execution of process 2 takes longer than that of process 1, tokens from process 1 accumulate in the buffer until it is full, causing process 1 to wait until process 2 produces new "space" tokens. This is called *back-pressure* and ensures synchronisation of computations.

Finally, any process that is enabled can execute, independent from other processes. Therefore, processes that are waiting for input do not prevent other processes from executing, making dataflow latency tolerant.

Appendix B

The Montium

The Montium is an example of a coarse-grained reconfigurable processor [43] developed by Recore Systems [81]. It is optimised for signal processing operations. Several core operations, called *kernels*, for signal processing applications have been implemented on the Montium; signal processing operations such as FIR filters and FFTs [44] and a number of baseband processing and wireless communication kernels such as CDMA and OFDM receivers, Viterbi and Turbo decoders [79] and Reed-Solomon decoding [KCR:1].

We will first present an overview of the processor landscape in order to position coarse-grained reconfigurable processors. Then we will discuss the Montium architecture in detail. Finally, we will present the implementations of three relatively complex (compared to the capabilities of the Montium) operations: a coordinate transformation for determining the magnitude and phase of a complex number using CORDIC, computation of the sine function using a LUT, and computing a complex division.

B.1 Processor landscape

Processing hardware can be divided into five groups with increasing efficiency, but decreasing flexibility [43]: general purpose processors (GPPs), application-domain optimised processors, coarse-grained reconfigurable processors, fine-grained reconfigurable processors, and ASICs.

A GPP is designed for general use and therefore offers the most flexibility, but has limited parallelism. Since operations are done sequentially, high clock speeds are needed to give good performance, resulting in lower energy efficiency. Furthermore, they implement ALUs that can compute a large variety of different operations, improving flexibility but requiring more control and energy overhead.

A digital signal processor (DSP) or a graphics processing unit (GPU) can be seen as a GPP optimised for an application domain, signal processing and graph-

ics respectively. Therefore they provide higher performance and energy efficiency for those domains, while still providing much of the flexibility of a GPP. DSPs have support for complex numbers, saturated computations, MACs operations and FFT butterfly operations for example, all of which are useful for the beamforming application. Another large advantage of GPPs, DSPs and GPUs is the support for higher level programming languages and tools.

Coarse-grained reconfigurable hardware is designed for word level algorithms. These are the same algorithms as the DSP is intended for, but instead of running a program, the hardware is configured to perform a certain task. This implies that signal paths are relatively stable improving energy efficiency. As expected, its flexibility comes from the ability to reconfigure the hardware for a particular algorithm within the hardware's application domain. Because the functional blocks are larger compared to fine-grained reconfigurable hardware, the overhead is less and this increases its power efficiency.

Fine-grained reconfigurable hardware, such as FPGAs, uses look-up tables to implement functionality and an extensive configurable interconnect between them. Configurability is therefore at the bit-level. Since the design of a fine-grained reconfigurable hardware device is very regular, it can be highly optimised for performance.However, the user is essentially specifying hardware, that is synthesised to configurations, therefore requiring more effort [104]. Furthermore, reconfiguration times are in the milli-second to second range [104].

If designed properly, an ASIC is the most efficient. It is efficient because the hardware is designed specifically for certain functionality and is not changeable. Therefore, the flexibility of an ASIC is limited by design.

B.2 The Montium processor

The Montium processor is an example of a coarse-grained reconfigurable processor intended for signal processing operations. As such, it nicely balances (energy) efficiency versus flexibility for this application domain [44].

The Montium is shown in figure B.1 and consists of three parts; the processing part array (PPA), the (instruction) decoders and the sequencer. Furthermore, it is connected to a communication and configuration unit (CCU) [104]. Its template based design allows for customisation of architectural properties. The default design has a data path width of 16 bit, a targeted clock frequency of 100 MHz for 90 nm technology, 5 parallel ALUs and 10 local memories of 1024 words. Its silicon area is approximately 2 mm^2 and its power consumption is approximately 550 μW/MHz.

Sequencer The sequencer stores and controls a sequence of instructions, i.e. a program or kernel. A program counter is used for the program flow and is directly connected to a static random-access memory (SRAM) containing the program to select the next instruction to be executed. Hence, an instruction can be fetched immediately from local memory and is not affected by a typical memory hierarchy

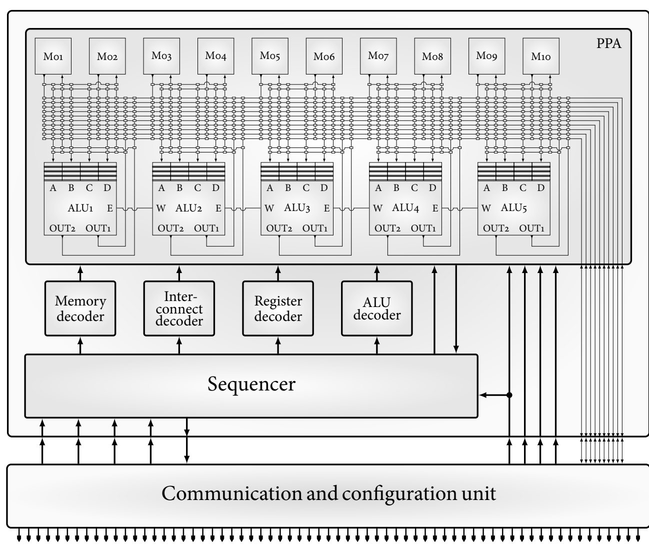


Figure B.1: Montium

with caches in conventional architectures, in which access latency is unpredictable. As there is no interaction between the data path and the instruction program, the kernel execution is fully deterministic.

Instruction decoders Instructions from the sequencer are decoded by the instruction decoders. A separate decoder is used for different parts of the processors; the memories, the interconnect, the registers and the ALUs. Each decoder contains a subset of all possible control signal combinations for that part. Fields in the sequencer instruction select an entry in each decoder, thereby severely limiting the needed control signals from instructions. Furthermore, decoder entries can be shared between instructions. Together this enables efficient storage of kernels and better energy efficiency as the number of changing control signals is reduced.

Processing part array Processing is performed using 5 processing parts. The instruction decoders decompress instructions into control signals for the processing parts. Each processing part contains an ALU, a register bank (4 deep for each of the 4 inputs), and 2 local memories. They are connected with a large crossbar consisting of 10 global buses that provides a high bandwidth to 10 memory units. Each ALU can be connected to 2 of the memories via a local interconnect or to the 8 other memories via the global buses. In addition, each ALU can receive an intermediate value from its right neighbour ALU via an east-west connection. Using these 5 inputs, multiple operations can be executed simultaneously and from each ALU at most 3 results can be generated (one to the west output and two to the bottom outputs, which are connected to the interconnect).

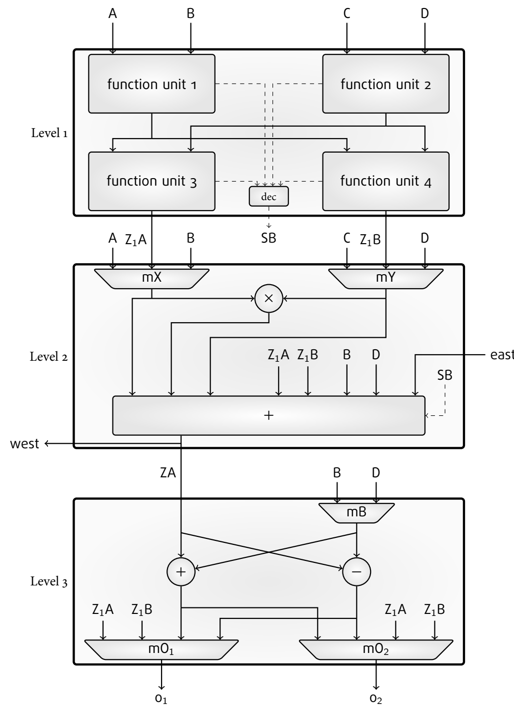


Figure B.2: Structure of one Montium ALU

Each memory has an address generation unit (AGU) to generate common addressing patterns such as linear addressing, stride-by-*n* (i.e., the address is incremented by *n* after each read or write operation), bit reversing (an output reordering technique that is typically used for FFT algorithms) and modulo counting (e.g. for creating circular buffers). This relieves the ALUs from the calculation of memory addresses making its instructions much more regular and enabling the address calculations to be performed in parallel with the data processing operations.

Figure B.2 shows the internal structure of one ALU. Each ALU consists of three levels. The first level contains four function units capable of logical functions and basic arithmetic. The two topmost function units are connected to four register banks providing inputs. The lower two function units are connected to the output of the units above. Each function unit generates status flags to indicate the occurrence of overflow, a negative result or whether its result equals zero. These status flags may be used by the sequencer, for example for conditional jumps. The second level contains a multiplier followed by an adder/subtracter for MAC operations. Either the outputs of the first level or the register bank can be used as input for the multiplier. The results or the same inputs are used as the left operand of the adder. In addition, the right operand for the adder can be statically or dynamically (depending on the value of the status bit SB) selected from inputs B, D, Z_1A and Z_1B.

Also note the east input that can be selected for the adder as well as the west output of the result. The third level contains a butterfly unit for FFT operations.

The ALU outputs are selected from the outputs of each level. Almost all arithmetic operations in the ALUs can be executed in either integer modus (i.e., they operate on the 16 rightmost bits) or in 1.15 fixed point modus (i.e. the leftmost bit is used as sign bit whereas the other bits contain the fixed point fraction). In order to avoid overflow, the intermediate values can be saturated. Since the ALU is not pipelined, the entire operation from the register file inputs to the ALU outputs can be done within one clock cycle. Each ALU, memory or entire processing part can be turned off when not used, saving energy.

Communication and configuration unit The CCU is responsible for control, configuration, memory initialisation, synchronisation as well as interfacing with the NoC. The CCU provides a configuration interface for temporally halting and reconfiguring the Montium and starting a kernel. Furthermore it provides two mechanisms for communication: block mode and streaming mode communication. In the block mode, the input samples are stored in the memories, and retrieved again when execution has finished, by means of a direct memory access (DMA) transfer by the CCU. In streaming mode, a program configured in the Montium can generate a read request for reading data from any input NoC connection or generate a write request for writing data to any output NoC connection. While waiting for a read or write request the Montium can continue processing, overlapping computation and communication.

B.3 Kernels implemented on the Montium

In this section, we will present three implementations of kernels [104] which are used for the case study; a coordinate transformation, sine computation and complex division.

Coordinate transformation A coordinate transformation from Cartesian coordinates $\vec{u} = (x, y)$ to polar coordinates (r, θ) is defined as:

$$r = |\vec{u}| = \sqrt{x^2 + y^2}$$
$$\theta = \angle \vec{u} = \arctan \frac{y}{x}$$

However, the square-root and arc-tangent are computationally expensive operations which are not supported in hardware on the Montium. These operations can be implemented using LUTs. However, full 16 bit accuracy requires $2^{16} = 64$k memory locations, while the Montium only has $10 \cdot 2^{10} = 10$k local memory available. A more efficient approach is the CORDIC algorithm [110], which iteratively approximates the magnitude and phase using only shift and add operations. Using CORDIC, or generalisations to the algorithm [112], also other operations, such as the sine and division operation among others, can be approximated iteratively. However, we will use different implementations for those operations.

The CORDIC equations are:

$$\begin{aligned} x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right) \end{aligned} \qquad \text{where} \qquad \begin{aligned} d_i &= +1 \quad \text{if } y_i < 0 \\ d_i &= -1 \quad \text{otherwise} \end{aligned}$$

In the limit this converges to:

$$\begin{aligned} x_n &= A_n \sqrt{x_0^2 + y_0^2} \\ y_n &= 0 \\ z_n &= z_0 + \arctan\left(\frac{y_0}{x_0}\right) \\ A_n &= \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \end{aligned} \qquad \text{such that} \qquad \begin{aligned} r &= \frac{x_n}{A_n} \\ \theta &= z_n \end{aligned}$$

Each x_{i+1}, y_{i+1} and z_{i+1} equation is implemented on a separate ALU. The shift operation (2^{-i}) and the optional negation (d_i) based on y_i are implemented using level 1 of the ALUs. The addition or subtraction is implemented using level 3. This implementation is shown in figure B.3. Using this implementation, one CORDIC iteration can be computed per clock cycle.

The part of the equation that is multiplied by d_i, i.e. $y_i \cdot 2^{-1}$ for the calculation of x_i, is always negated. Then based on d_i either the negated or non-negated value is chosen. The decision variable d_i is generated as a status bit of one of the functional units, which indicates if the value it contains (y_i in this case) is negative or not. In case of x_{i+1} this is a by-product of the shift-operation, otherwise y_i is passed to the function unit explicitly. As mentioned, the arc-tangent operation is not available. Therefore it is provided provided by a LUT. However, with CORDIC only a limited number of arc-tangents need to be stored in memory, one for each iteration. As each iteration adds one bit of accuracy to the result and the Montium uses 16 -bit, a maximum of 16 iterations and thus 16 memory locations are required. However, due to the limited word-width in combination with the bit-shift operation, the smallest error is already reached after 14 iterations [104]. Note that the CORDIC equations are only valid for angles between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. For larger angles an initial rotation is performed with a similar set of equations [104], thus requiring one additional iteration. In addition, the gain A_n of the CORDIC algorithm must be corrected with a multiplication with a constant requiring another clock cycle, giving a total of 16 clock cycles again.

Sine computation The CORDIC algorithm can also be used to calculate the sine function [110], requiring another 16 clock cycles for 14 iterations of CORDIC. An implementation as a LUT requires only 2 clock cycles; one for setting the memory address and one for retrieving the value. However, with 10 bit memories, the accuracy of the sine function is limited to 10 bit.

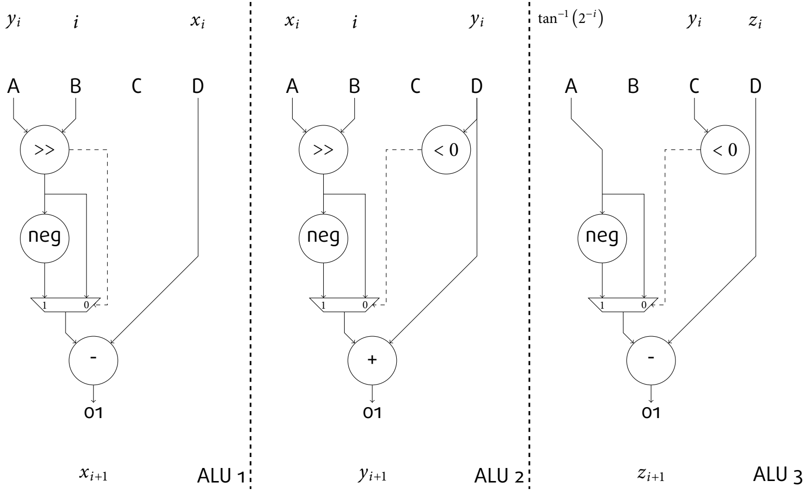


Figure B.3: Implementation of the CORDIC algorithm

Complex division Again the CORDIC algorithm can be used for division [110], at the cost of 16 clock cycles. For a complex division another generalisation is available [8], which uses complex valued decision variables and additional logarithmic LUTs. We estimate it to take 2 clock cycles per iteration and 4.5 times as much memory [8] to implement.

Using the multipliers of the Montium, a more direct implementation is chosen. Complex division is defined as:

$$\frac{\vec{u}}{\vec{v}} = \frac{a + jb}{c + jd} = \frac{ac + bd + j}{c^2 + d^2} + j\frac{bc - ad}{c^2 + d^2} = \frac{ac + bd}{|\vec{v}|^2} + j\frac{bc - ad}{|\vec{v}|^2}$$

The computation of the two nominator parts requires four multiplications, an addition and a subtraction. The nominator is implemented using level 2 of 4 ALUs and the east-west connections. The computation of the denominator $|\vec{v}|^2$ requires two multiplications and an addition. The two divisions are implemented by replacing them with multiplication and using a LUT for the multiplication factor $\frac{1}{|v^2|}$.

Using 1.15 fixed-point values, however, means that with $|\vec{v}|^2$ in the range of $[0 \ldots 1\rangle$, $\frac{1}{|\vec{v}|^2}$ is in the range of $\langle 1 \ldots \infty \rangle$, which cannot be represented in a 1.15 fixed point notation. Therefore, a scale factor μ must be included in the LUT, i.e. the LUT implements $\frac{\mu}{|\vec{v}|^2}$. For example, for $\mu = 1/160$ the range of $\frac{\mu}{|\vec{v}|^2}$ is $\langle 0.00625 \ldots \infty \rangle$. For this range, values of $|\vec{v}|^2 \leq 1/160 \implies |\vec{v}| < \sqrt{1/160}$ (≈ 0.08) are saturated to 1.

The LUT operation takes two clock cycles, but in the first clock cycle the denominator is already computed, which is then multiplied with the result from the LUT in the second clock cycle, giving a total of two clock cycles for the complex division.

Acronyms

1D 1-dimensional
2D 2-dimensional
3D 3-dimensional

A-CMA angular CMA
ADC analogue-to-digital converter
AGU address generation unit
AHB advanced high-performance bus
ALU arithmetic logic unit
AP antenna processing
ASIC application specific integrated circuit

BB baseband
BC beamcontrol
BF beamforming
BS beamsteering

CCU communication and configuration unit
CMA constant modulus algorithm
CORDIC coordinate rotation digital computer
CSDF cyclo-static dataflow
CT continuous-time

DAC digital-to-analogue converter
DDF dynamic dataflow
DE discrete event
DF dataflow
DMA direct memory access
DoA direction of arrival
DSP digital signal processing
DSP digital signal processor
DT discrete-time
DVB-S digital video broadcast for satellite

E-CMA extended CMA
EDSL embedded domain specific language
ESPRIT estimation of signal parameters by rotational invariance techniques

FFT fast Fourier transform
FIFO first-in first-out
FIR finite impulse response
FPGA field-programmable gate array

FRP	functional reactive programming
GPP	general purpose processor
GPU	graphics processing unit
GUI	graphical user interface
HDL	hardware description language
HPBW	half-power beamwidth
IC	integrated circuit
IF	intermediate frequency
INBW	inter-null beamwidth
LO	local oscillator
LOFAR	low frequency array
LPT	linear phase taper
LUT	lookup table
MAC	multiply-accumulate
MBiS	multiple boards in a system
MCoB	multiple chips on a board
MIMO	multiple-input multiple-output
ML	maximum likelihood
MPSoC	multiprocessor system-on-chip
MRDF	multi-rate dataflow
MUSIC	multiple signal classification
NoC	network-on-chip
ODE	ordinary differential equation
ops	operations per second
PN	process network
PPA	processing part array
PS	phase shift
PSK	phase-shift keying
QAM	quadrature amplitude modulation
QPSK	quadrature phase-shift keying
RF	radio frequency
RRC	root-raised-cosine
SDF	synchronous dataflow
SDR	software-defined radio
SNR	signal-to-noise ratio
SoC	system-on-chip
SR	synchronous/reactive
SRAM	static random-access memory
SRDF	single-rate dataflow
TD	time delay
ULA	uniform linear array
VLIW	very long instruction word
VPDF	variable-rate phased dataflow
ZOH	zero-order-hold

Bibliography

[1] Haskell 98 Language and Libraries: The Revised Report. *Journal of Functional Programming*, 13(1), January 2003.

[2] Aeroflex Gaisler. The LEON2 processor. URL `http://www.gaisler.com`.

[3] Ben Allen and Mohammad Ghavami. *Adaptive Array Systems, Fundamentals and Applications*. Wiley, May 2005. ISBN 978-0-470-86189-9.

[4] Apple-CORE consortium. Architecture Paradigms and Programming Languages for Efficient programming of multiple CORES (Apple-Core) project. URL `http://www.apple-core.info/`.

[5] Krste Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, December 2006.

[6] Christiaan Baaij, Mathijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *Digital System Design: Architectures, Methods and Tools (DSD 2010), 13th Euromicro Conference on*, pages 714–721, September 2010.

[7] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978. DOI 10.1145/359576.359579.

[8] Jean-Claude Bajard, Sylvanus Kla, and Jean-Michel Muller. BKM: a new hardware algorithm for complex elementary functions. *Computers, IEEE Transactions on*, 43 (8):955–963, August 1994.

[9] Arya Behzad. Radio Design for MIMO Systems with an Emphasis on IEEE 802.11n. In *Solid-State Circuits Conference (ISSCC 2007), IEEE International*, 2007. Tutorial.

[10] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of computer programming*, 19(2): 87–152, November 1992. DOI 10.1016/0167-6423(92)90005-V.

[11] Tjerk Bijlsma. *Automatic parallelization of nested loop programs (for non-manifest real-time stream processing applications)*. PhD thesis, University of Twente, July 2011. ISBN 978-90-365-3173-3. DOI 10.3990/1.9789036531733.

[12] Tjerk Bijlsma, Marco J. G. Bekooij, Piere G. Jansen, and Gerard J. M. Smit. Communication between Nested Loop Programs via Circular Buffers in an Embedded Multiprocessor System. In *Software & Compilers for Embedded Systems (SCOPES 2008), 11th International Workshop on*, pages 33–42. http://eprints.eemcs.utwente.nl/11970/, March 2008. ISBN not assigned.

[13] Benjamin S. Blanchard and Wolter J. Fabrycky. *Systems Engineering and Analysis*. Prentice Hall, 3rd edition, 1998. ISBN 978-0-1313-5047-2.

[14] Koen C. H. Blom. *DVB-S signal tracking techniques for mobile phased arrays.* Master's thesis, University of Twente, December 2009.

[15] Gerard Bos. *Radio astronomy signal processing on a tiled reconfigurable architecture*. Master's thesis, University of Twente, July 2010.

[16] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2nd edition, October 2004. ISBN 978-0-3872-3137-2.

[17] Luca P. Carloni, Roberto Passerone, Alessandro Pinto, and Alberto L. Sangiovanni-Vincentelli. Languages and Tools for Hybrid Systems Design. *Foundations and Trends in Electronic Design Automation*, 1(1/2):1–193, June 2006. DOI 10.1561/1000000001.

[18] Paul Caspi and Marc Pouzet. Lucid Synchrone: une extension fonctionnelle de Lustre. *Journées Francophones des Langages Applicatifs (JFLA)*, February 1999.

[19] Antony Courtney and Conal Elliott. Genuinely Functional User Interfaces. In *ACM SIGPLAN Haskell Workshop (HW'2001)*, pages 41–69, September 2001.

[20] CRISP consortium. Cutting Edge Reconfigurable ICs for Stream Processing (CRISP) project. URL `http://www.crisp-project.eu/`.

[21] Marco de Vos. LOFAR: the first of a new generation of radio telescopes. In *Acoustics, Speech, and Signal Processing (ICASSP'05), IEEE International Conference on*, pages 865–868, March 2005. DOI 10.1109/ICASSP.2005.1416441.

[22] Marco de Vos, André W. Gunst, and Ronald Nijboer. The LOFAR Telescope: System architecture and signal processing. *Proceedings of the IEEE*, 97(8):1431–1437, August 2009. DOI 10.1109/JPROC.2009.2020509.

[23] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, July 2005. DOI 10.1145/1070838.1070856.

[24] Johan Eker et al. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003. DOI 10.1109/JPROC.2002.805829.

[25] Conal Elliott. Functional Implementations of Continuous Modeled Animation. In *Principles of Declarative Programming (PLILP'98/ALP'98), 10th International Symposium on*, pages 284–299. Springer, July 1998.

[26] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Functional programming (ICFP '97), 2nd ACM SIGPLAN international conference on*, pages 263–273. ACM, August 1997. ISBN 0-89791-918-1. DOI 10.1145/258948.258973.

[27] Cagkan Erbas, Andy D. Pimentel, Mark Thompson, and Simon Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP Journal on Embedded Systems*, 2007(1), January 2007. DOI 10.1155/2007/82123.

[28] ETSI. Digital video broadcasting (DVB); framing structure, channel coding and modulation for the 11/12 GHz satellite services. Technical Report ETSI EN 300 421 (V1.1.2), August 1997.

[29] ETSI. Digital Video Broadcasting (DVB): Second generation framing structure, channel coding and modulation system for Broadcasting. Technical Report ETSI EN 302 307 V1.2.1, August 2009.

[30] Thomas Huining Feng and Edward A. Lee. Scalable Models Using Model Transformation. In *Model Based Architecting and Construction of Embedded Systems (ACESMB), 1st International Workshop on*. EECS Department, University of California, Berkeley, September 2008.

[31] John G. F. Francis. The QR transformation. *The Computer Journal*, 4(3):265, 1961.

[32] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming*. Springer, 1998. DOI 10.1007/BFb0054087.

[33] Lal Chand Godara. *Smart antennas*. CRC Press, January 2004. ISBN 978-0-8493-1206-9.

[34] Dominique N. Godard. Self-recovering equalization and carrier tracking in two-dimensional data communication systems. *Communications, IEEE Transactions on*, 28(11):1867–1875, November 1980. DOI 10.1109/TCOM.1980.1094608.

[35] Christoph Grimm, Martin Barnasconi, Alain Vachoux, and Karsten Einwich. An Introduction to Modeling Embedded Analog/Mixed-Signal Systems using SystemC AMS Extensions. Technical report, June 2008.

[36] André W. Gunst and Gideon W. Kant. Signal Transport and Processing at the LOFAR Remote Stations. In *28th Union Radio-Scientifique Internationale General Assembly (URSI 2005)*, October 2005.

[37] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9): 1305–1320, September 1991. DOI 10.1109/5.97300.

[38] Robert C. Hansen. *Phased Array Antennas*. Wiley, January 1998. ISBN 978-0-4715-3076-3.

[39] Andreas Hansson. *A composable and predictable on-chip interconnect*. PhD thesis, Technische Universiteit Eindhoven, June 2009. ISBN 978-90–386-1871-5.

[40] Simon Haykin. *An introduction to analog and digital communications*. Wiley, 1989. ISBN 978-0-471-85978-8.

[41] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, September 2006. ISBN 978-0-12-370490-0.

[42] Thomas A. Henzinger and Joseph Sifakis. The Embedded Systems Design Challenge. In *Formal Methods (FM 2006), 14th International Symposium on*, pages 1–15. Springer, August 2006.

[43] Paul M. Heysters. *Coarse-Grained Reconfigurable Processors - Flexibility meets Efficiency*. PhD thesis, University of Twente, September 2004. ISBN 978-0-139-42716-3.

[44] Paul M. Heysters and Gerard J. M. Smit. Mapping of DSP Algorithms on the Montium Architecture. In *Parallel and Distributed Processing Symposium (RAW 2003), 17th IEEE International*. http://eprints.eemcs.utwente.nl/1516/, April 2003. DOI 10.1109/IPDPS.2003.1213333.

[45] Philip K. F. Hölzenspies. *On run-time exploitation of concurrency*. PhD thesis, Univsity of Twente, April 2010. ISBN 978-90-365-3021-7. DOI 10.3990/1.9789036530217.

[46] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003. ISBN 978-3-540-40132-2. DOI 10.1007/978-3-540-44833-4_6.

[47] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007. ISBN 978-0-5216-9269-4.

[48] Axel Jantsch. *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*. Morgan Kaufmann, June 2003. ISBN 978-1-55860-925-9.

[49] Eric A. M. Klumperink, Bram Nauta, André B. J. Kokkeler, and Gerard J. M. Smit. CMOS Beamforming Techniques STW project proposal. Technical report, 2006.

[50] Marco J. Kruijswijk. *Hierarchical wideband beamforming using fixed weights*. Master's thesis, University of Twente, June 2011.

[51] Timo I. Laakso, Vesa Välimäki, Matti Karjalainen, and Unto K. Laine. Splitting the unit delay - Tools for fractional delay filter design. *IEEE Signal Processing Magazine*, 13(1):30–60, January 1996.

[52] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991. DOI 10.1109/5.97301.

[53] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *Object Oriented Real-Time Distributed Computing (ISORC 2008), 11th IEEE International Symposium on*, pages 363–369. IEEE, May 2008. DOI 10.1109/ISORC.2008.25.

[54] Edward A. Lee. Computing Needs Time. *Communications of the ACM*, 52(5):70–79, May 2009. DOI 10.1145/1506409.1506426.

[55] Edward A. Lee and Eleftherios Matsikoudis. The Semantics of Dataflow with Firing. In *From Semantics to Computer Science*. Cambridge University Press, September 2009. ISBN 978-0-5215-1825-3.

[56] Edward A. Lee and David G. Messerchmitt. Synchronous Data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. DOI 10.1109/PROC.1987.13876.

[57] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. DOI 10.1109/5.381846.

[58] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, December 1998. DOI 10.1109/43.736561.

[59] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. 2011. ISBN 978-0-557-70857-4.

[60] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Embedded Software (EMSOFT'07), 7th ACM & IEEE International Conference on*, pages 114–123, October 2007. DOI 10.1145/1289927.1289949.

[61] MapleSoft. MapleSim. URL `http://www.maplesoft.com/products/maplesim/`.

[62] Conor McBride and Ross Paterson. Functional Pearl: Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. DOI 10.1017/S0956796807006326.

[63] Wolfgang Mueller, Alberto Rosti, Sara Bocchio, Elvinia Riccobene, Patrizia Scandurra, Wim Dehaene, and Yves Vanderperren. UML for ESL design: basic principles, tools, and applications. In *Computer-Aided Design (ICCAD 2006), IEEE/ACM International Conference on*, pages 73–80. ACM, November 2006. DOI 10.1109/ICCAD.2006.320068.

[64] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, December 1999. DOI 10.1016/S0167-8191(99)00070-8.

[65] National Instruments. NI LabVIEW - Improving the Productivity of Engineers and Scientists. URL `http://www.ni.com/labview/`.

[66] Paul A. Nelson and Stephen J. Elliott. *Active Control of Sound*. Academic Press, June 1993. ISBN 978-0-125-15426-0.

[67] NEST consortium. Netherlands Streaming (NEST) project. URL `http://www.nest-consortium.nl/`.

[68] Gabriela Nicolescu and Pieter J. Mosterman. *Model-Based Design for Embedded Systems*. CRC Press, November 2009. ISBN 978-1-4200-6784-2.

[69] Hristo Nikolov et al. Daedalus: toward composable multimedia MP-SoC design. In *Design Automation Conference (DAC'08), 45th annual*, pages 574–579. ACM, June 2008. ISBN 978-1-60558-115-6. DOI 10.1145/1391469.1391615.

[70] Object Management Group, Inc. (OMG). OMG Systems Modeling Language (OMG SysML). Technical Report Version 1.1, November 2008.

[71] OMG Architecture Board ORMSC. Model Driven Architecture (MDA). Technical Report ormsc/2001-07-01, July 2001.

[72] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, November 2008. ISBN 978-0-5965-1498-3.

[73] Ross Paterson. Arrows and computation. In *The Fun of Programming*, pages 201–222. Palgrave Macmillan, March 2003. ISBN 978-1-4039-0772-1.

[74] Arogyaswami J. Paulraj, Richard H. Roy, and Thomas Kailath. A subspace rotation approach to signal parameter estimation. *Proceedings of the IEEE*, 74(7):1044–1046, July 1986. DOI 10.1109/PROC.1986.13583.

[75] John Peterson, Gregory D. Hager, and Paul Hudak. A language for declarative robotic programming. In *Robotics and Automation, IEEE International Conference on*, pages 1144–1151. IEEE, May 1999. DOI 10.1109/ROBOT.1999.772516.

[76] Rik Portengen. *Phased array antenna processing on reconfigurable hardware*. Master's thesis, University of Twente, December 2007.

[77] John G. Proakis and Dimitris K. Manolakis. *Digital Signal Processing*. Prentice Hall, 4th edition, April 2006. ISBN 978-0-1318-7374-1.

[78] Gerard K. Rauwerda. *Multi-Standard Adaptive Wireless Communication Receivers - Adaptive Applications Mapped on Heterogeneous Dynamically Reconfigurable Hardware*. PhD thesis, University of Twente, January 2008. ISBN 978-90-365-2607-4. DOI 10.3990/1.9789036526074.

[79] Gerard K. Rauwerda, Paul M. Heysters, and Gerard J. M. Smit. Towards Software Defined Radios using Coarse - Grained Reconfigurable Hardware. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):3–13, January 2008. DOI 10.1109/TVLSI.2007.912075. ISSN 1063-8210.

[80] Behzad Razavi. *RF microelectronics*. Prentice Hall, November 1997. ISBN 978-0-1388-7571-5.

[81] Recore Systems. The Montium processor. URL `http://www.recoresystems.com`.

[82] Hideki John Reekie. *Realtime Signal Processing: Dataflow, Visual, and Functional Programming*. PhD thesis, University of Technology Sydney, September 1995.

[83] Kenneth C. Rovers. *Front-end research for a low-cost spectrum analyser*. Master's thesis, University of Twente, June 2006.

[84] Richard H. Roy, Arogyaswami J. Paulraj, and Thomas Kailath. ESPRIT - A subspace rotation approach to estimation of parameters of cisoids in noise. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 34(5):1340–1342, October 1986. DOI 10.1109/TASSP.1986.1164935.

[85] Michael Rübsamen and Alex B. Gershman. Direction-of-Arrival Estimation for Nonuniform Sensor Arrays: From Manifold Separation to Fourier Domain MUSIC Methods. *Signal Processing, IEEE Transactions on*, 57(2):588–599, February 2009. DOI 10.1109/TSP.2008.2008560.

[86] Ingo Sander. *System modeling and design refinement in ForSyDe*. PhD thesis, KTH Royal Institute of Technology, April 2003.

[87] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(1):17–32, January 2004. DOI 10.1109/TCAD.2003.819898.

[88] Ralph O. Schmidt. Multiple Emitter Location and Signal Parameter Estimation. *Antennas and Propagation, IEEE Transactions on*, 34(3):276–280, March 1986. DOI 10.1109/TAP.1986.1143830.

[89] Dana Scott and Christopher Strachey. Toward a mathematical semantics for programming languages. In *Computers and Automata, Symposium on*, pages 19–46. Symposium on Computers and Automata, April 1971.

[90] Merrill I. Skolnik. *Introduction to Radar Systems*. McGraw-Hill, 3rd edition, December 2000. ISBN 978-0-0704-4533-8.

[91] Gerard J. M. Smit, André B. J. Kokkeler, Pascal T. Wolkotte, Philip K. F. Hölzenspies, Marcel D. van de Burgwal, and Paul M. Heysters. The Chameleon Architecture for Streaming DSP Applications. *EURASIP Journal on Embedded Systems*, 2007:78082, January 2007. DOI 10.1155/2007/78082.

[92] Gerard J. M. Smit, André B. J. Kokkeler, Pascal T. Wolkotte, and Marcel D. van de Burgwal. Multi-core Architectures and Streaming Applications. In *System Level Interconnect Prediction (SLIP 2008), 10th International Workshop on*, pages 35–42, April 2008. DOI 10.1145/1353610.1353618.

[93] Samir S. Soliman and Mandyam D. Srinath. *Continuous and Discrete Signals and Systems*. Prentice Hall, 2nd edition, January 1998. ISBN 978-0-135-1847-3.

[94] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprette. System design using Khan process networks: the Compaan/Laura approach. In *Design, Automation & Test in Europe Conference & Exhibition (DATE 2004)*, pages 340–345, February 2004. DOI 10.1109/DATE.2004.1268870.

[95] Fasil C. Taddesse. *Implementation of adaptive beamforming on a multiprocessor system on chip*. Master's thesis, University of Twente, September 2010.

[96] Walid Taha, Paul Brauner, Robert Cartwright, Verónica Gaspes, Aaron Ames, and Alexandre Chapoutot. A Core Language for Executable Models of Cyber Physical Systems. *ACM SIGBED Review*, 8(2):39–43, June 2011. DOI 10.1145/2000367.2000376.

[97] Timon D. ter Braak, Philip K. F. Hölzenspies, Jan Kuper, Johann L. Hurink, and Gerard J. M. Smit. Run-time spatial resource management for real-time applications on heterogeneous MPSoCs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 357–362. European Design and Automation Association, March 2010. ISBN 978-3-9810801-6-2.

[98] Timon D. ter Braak, Hermen A. Toersche, André B. J. Kokkeler, and Gerard J. M. Smit. Adaptive resource allocation for streaming applications. In *Embedded Computer Systems (SAMOS 2011), International Conference on*, pages 388–395, July 2011. DOI 10.1109/SAMOS.2011.6045489.

[99] The MathWorks. MATLAB and Simulink for Technical Computing. URL `http://www.mathworks.com/`.

[100] John R. Treichler and Brian G. Agee. A New Approach to Multipath Correction of Constant Modulus Signals. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 31(2):459–472, April 1983. DOI 10.1109/TASSP.1983.1164062.

[101] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998. DOI 10.1017/S0956796897002967.

[102] Alain Vachoux, Christoph Grimm, and Karsten Einwich. SystemC-AMS Requirements, Design Objectives and Rationale. In *Design, Automation and Test in Europe Conference and Exhibition (DATE 2003)*, pages 388–393. IEEE, December 2003. ISBN 0-7695-1870-2. DOI 10.1109/DATE.2003.1253639.

[103] Vesa Välimäki and Timo I. Laakso. Principles of fractional delay filters. In *Acoustics, Speech, and Signal Processing (ICASSP'00), IEEE International Conference on*, pages 3870–3873. IEEE, June 2000. DOI 10.1109/ICASSP.2000.860248.

[104] Marcel D. van de Burgwal. *Interfacing networks-on-chip : hardware meeting software*. PhD thesis, University of Twente, October 2010. ISBN 978-90-365-3067-5. DOI 10.3990/1.9789036530675.

[105] Harry L. van Trees. *Optimum array processing*, volume Detection, estimation and modulation theory. Wiley, March 2002. ISBN 978-0-4710-9390-9.

[106] Frank E. van Vliet. Trends in Wideband Phased-Array Front-Ends. In *European Radar Conference (EuRAD 2007)*, October 2007. ISBN 978-2-87487-004-0. DOI 10.1109/EURAD.2007.4404960.

[107] Sriram Vangal et al. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Solid-State Circuits Conference (ISSCC 2007), IEEE International*, pages 98–99,589, February 2007. DOI 10.1109/ISSCC.2007.373606.

[108] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4): 365–396, December 1986. DOI 10.1145/27633.28055.

[109] Hubregt J. Visser. *Array and Phased Array Antenna Basics*. Wiley, September 2005. ISBN 978-0-470-87117-1.

[110] Jack E. Volder. The CORDIC Trigonometric Computing Technique. *Electronic Computers, IRE Transactions on*, 8(3):330–334, September 1959. DOI 10.1109/TEC.1959.5222693.

[111] Jasper D. Vrielink. *Phased Array Processing: Direction of Arrival Estimation on Reconfigurable Hardware*. Master's thesis, University of Twente, January 2009.

[112] John S. Walther. A unified algorithm for elementary functions. In *Spring Joint Computer Conference (AFIPS'71)*, pages 379–385, May 1971. DOI 10.1145/1478786.1478840.

[113] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Functional programming (ICFP'01), 6th ACM SIGPLAN International Conference on*, pages 146–156. ACM, September 2001. DOI 10.1145/507635.507654.

[114] Rinse Wester. *A dataflow architecture for beamforming operations*. Master's thesis, December 2010.

[115] Maarten H. Wiggers. *Aperiodic multiprocessor scheduling for real-time stream processing applications*. PhD thesis, University of Twente, June 2009. ISBN 978-90-365-2850-4. DOI 10.3990/1.9789036528504.

[116] Pascal T. Wolkotte. *Exploration within the Network-on-Chip Paradigm*. PhD thesis, University of Twente, January 2009. ISBN 978-90-365-2757-6. DOI 10.3990/1.9789036527576.

[117] Pascal T. Wolkotte, Gerard J. M. Smit, Gerard K. Rauwerda, and Lodewijk T. Smit. An Energy-Efficient Reconfigurable Circuit Switched Network-on-Chip. In *Parallel and Distributed Processing Symposium (RAW 2005), 19th IEEE International*. IEEE Computer Society, April 2005. ISBN 978-0-7695-2312-9. DOI 10.1109/IPDPS.2005.95.

[118] Zhengyuan Xu. New cost function for blind estimation of M-PSK signals. In *Wireless Communications and Networking Conference (WCNC 2000), IEEE*, pages 1501–1505, September 2000. DOI 10.1109/WCNC.2000.904857.

[119] Haiyang Zheng. *Operational Semantics of Hybrid Systems*. PhD thesis, University of California Berkeley, May 2007. ISBN 978-0-549-17252-9.

[120] Ilan Ziskind and Mati Wax. Maximum likelihood localization of multiple sources by alternating projection. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 36(10):1553–1560, October 1988. DOI 10.1109/29.7543.

List of Publications

Refereed

[KCR:1] Arjan C. Dam, Michel G. J. Lammertink, Kenneth C. Rovers, Johan Slagman, Arno M. Wellink, Gerard K. Rauwerda, and Gerard J. M. Smit. Hardware / Software Co-design Applied to Reed-Solomon Decoding for the DMB Standard. In *Digital System Design: Architectures, Methods and Tools (DSD 2006), 9th EUROMICRO Conference on*, pages 447–455. IEEE Computer Society, August 2006. ISBN 978-0-7695-2609-8. DOI 10.1109/DSD.2006.59.

[KCR:2] Mark S. Oude Alink, André B. J. Kokkeler, Eric A. M. Klumperink, Kenneth C. Rovers, Gerard J. M. Smit, and Bram Nauta. Spurious-Free Dynamic Range of a Uniform Quantizer. *Circuits and Systems Part II: Express Briefs, IEEE Transactions on*, 56(6):434–438, June 2009. ISSN 1549-7747. DOI 10.1109/TCSII.2009.2020929.

[KCR:3] Kenneth C. Rovers, Marcel D. van de Burgwal, Jan Kuper, and Gerard J. M. Smit. Towards effective modeling and programming multi-core tiled reconfigurable architectures. In *Engineering of Reconfigurable Systems & Algorithms (ERSA '09), International Conference on*, pages 167–173. CSREA, July 2009. ISBN 978-1-60132-101-5.

[KCR:4] Koen C. H. Blom, Marcel D. van de Burgwal, Kenneth C. Rovers, André B. J. Kokkeler, and Gerard J. M. Smit. DVB-S Signal Tracking Techniques for Mobile Phased Arrays. In *Vehicular Technology Conference Fall (VTC 2010-Fall), IEEE 72nd*, pages 1–5. IEEE, September 2010. ISBN 978-1-4244-3573-9. DOI 10.1109/VETECF.2010.5594146.

[KCR:5] Anja Niedermeier, Rinse Wester, Kenneth C. Rovers, Christiaan Baaij, Jan Kuper, and Gerard J. M. Smit. Designing a dataflow processor using CλaSH. In *NORCHIP 2010*, pages 1–4. IEEE, November 2010. ISBN 978-1-4244-8971-8. DOI 10.1109/NORCHIP.2010.5669445.

[KCR:6] Marcel D. van de Burgwal, Kenneth C. Rovers, Koen C. H. Blom, André B. J. Kokkeler, and Gerard J. M. Smit. Adaptive Beamforming Using the Reconfigurable MONTIUM TP. In *Digital System Design: Architectures, Methods and Tools (DSD 2010), 13th Euromicro Conference on*, pages 301–308. IEEE Computer Society, September 2010. ISBN 978-1-4244-7839-2. DOI 10.1109/DSD.2010.13.

[KCR:7] Koen C. H. Blom, Marcel D. van de Burgwal, Kenneth C. Rovers, André B. J. Kokkeler, and Gerard J. M. Smit. Angular CMA: A modified Constant Modulus Algorithm providing steering angle updates. In *Wireless and Mobile Communications (ICWMC 2011), 7th International Conference on*, pages 42–47. IARIA, June 2011. ISBN 978-1-61208-140-3.

[KCR:8] Kenneth C. Rovers, Jan Kuper, and Gerard J. M. Smit. The problem with time in mixed continuous/discrete time modelling. *ACM SIGBED Review*, 8(2):27–30, June 2011. ISSN 1551-3688. DOI 10.1145/2000367.2000373.

[KCR:9] Kenneth C. Rovers, Jan Kuper, Marcel D. van de Burgwal, André B. J. Kokkeler, and Gerard J. M. Smit. Mixed continuous / discrete time modelling with exact time adjustments. In *Wireless Communications and Mobile Computing Conference (CyPhy'11), 7th International*, pages 1111–1116. IEEE, July 2011. ISBN 978-1-4244-9539-9. DOI 10.1109/IWCMC.2011.5982696.

[KCR:10] Kenneth C. Rovers, Marcel D. van de Burgwal, Jan Kuper, André B. J. Kokkeler, and Gerard J. M. Smit. Multi-domain transformational design flow for embedded systems. In *Embedded Computer Systems (SAMOS 2011), International Conference on*, pages 93–101. IEEE Computer Society, July 2011. ISBN 978-1-4577-0802-2. DOI 10.1109/SAMOS.2011.6045449.

[KCR:11] Marcel D. van de Burgwal, Kenneth C. Rovers, Koen C. H. Blom, André B. J. Kokkeler, and Gerard J. M. Smit. Mobile satellite reception with a virtual satellite dish based on a reconfigurable multi-processor architecture. *Microprocessors and Microsystems*, pages 1–29, 2011. ISSN 0141-9331. DOI 10.1016/j.micpro.2011.08.005.

Non-refereed

[KCR:12] Kenneth C. Rovers, Marcel D. van de Burgwal, André B. J. Kokkeler, and Gerard J. M. Smit. Rationale for and design of a generic tiled hierarchical phased array beamforming architecture. In *Circuits, Systems and Signal Processing (ProRISC 2007), 18th Annual Workshop on*, pages 160–168. STW Technology Foundation, November 2007.

[KCR:13] Marcel D. van de Burgwal, Kenneth C. Rovers, André B. J. Kokkeler, Gerard J. M. Smit, S Kasra Garakoui, Michiel C M Soer, Eric A. M. Klumperink, and Bram Nauta. CMOS Beamforming Techniques project overview. In *Scientific ICT Research Event Netherlands (SIREN 2007)*. Informatica Platform Nederland, October 2007. URL `http://www.ictonderzoek.net/3/assets/File/posters/2007_23/2007_23.pdf`.

[KCR:14] Kenneth C. Rovers, Jan Kuper, and Gerard J. M. Smit. Semantic programming model-based design - Defining a hierarchical tiled multi-processor architecture. In *Circuits, Systems and Signal Processing (ProRISC 2008), 19th Annual Workshop on*, pages 83–88. STW Technology Foundation, November 2008.

[KCR:15] Kenneth C. Rovers, Marcel D. van de Burgwal, André B. J. Kokkeler, Jan Kuper, and Gerard J. M. Smit. Phased Array Beamforming Processing - Semantic & Dataflow Model Based Design. In *Scientific ICT Research Event Netherlands (SIREN 2009)*. Informatica Platform Nederland, November 2009. URL `http://www.ictonderzoek.net/3/assets/File/posters/2009_44/2009_44.pdf`.

[KCR:16] Kenneth C. Rovers, Marcel D. van de Burgwal, Jan Kuper, André B. J. Kokkeler, and Gerard J. M. Smit. On reconfigurable tiled multi-core programming - Processing cores evaluation. In *Circuits, Systems and Signal Processing (ProRISC 2009), 20th Annual Workshop on*, pages 507–514. STW Technology Foundation, November 2009.

Kenneth C. Rovers received his M.Sc. degree in electrical engineering and his M.Sc. degree in computer science in 2006 from the University of Twente, the Netherlands. For the last five years he has been working towards his Ph.D. degree in the Computer Architecture for Embedded Systems (CAES) group at the same university. His master's thesis was on the system design of RF front-ends. The work presented in this thesis is in the area of model-based design of embedded systems, focusing on the modelling of multiple domains, accurate inclusion of time, mathematical definitions, and model transformations, with a beamforming application as an example. His research interests include system level design, functional programming, reconfigurable tiled architectures, and dataflow processors.

ISBN 978-90-365-3294-5