Design Space Decomposition for Cognitive and Software Defined Radios

Almohanad S. Fayez

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical Engineering

Charles W. Bostian, Chair
Scott F. Midkiff
Cameron D. Patterson
William T. Baumann
Michael R. Taaffe

May 29, 2013
Blacksburg, Virginia

Keywords: Software Defined Radio, Cognitive Radio, Models of Computation, CSP, SDF, GNU Radio, OCCAM
Copyright 2013, Almohanad S. Fayez

Design Space Decomposition for Cognitive and Software Defined Radios

Almohanad S. Fayez

(ABSTRACT)

Software Defined Radios (SDRs) lend themselves to flexibility and extensibility because they depend on software to implement radio functionality. Cognitive Engines (CEs) introduce intelligence to radio by monitoring radio performance through a set of *meters* and configuring the underlying radio design by modifying its *knobs*. In Cognitive Radio (CR) applications, CEs intelligently monitor radio performance and reconfigure them to meet it application and RF channel needs. While the issue of introducing computational knobs and meters is mentioned in literature, there has been little work on the practical issues involved in introducing such computational radio controls.

This dissertation decomposes the radio definition to reactive models for the CE domain and real-time, or dataflow models, for the SDR domain. By allowing such design space decomposition, CEs are able to define implementation independent radio graphs and rely on a model transformation layer to transform reactive radio models to real-time radio models for implementation. The definition of knobs and meters in the CE domain is based on properties of the dataflow models used in implementing SDRs. A framework for developing this work is presented, and proof of concept radio applications are discussed to demonstrate how CEs can gain insight into computational aspects of their radio implementation during their reconfiguration decision process.

Grant Information

This project was supported in the past by Award No. 2009-SQ-B9-K011 awarded by the National Institute of Justice, Office of Justice Programs, US Department of Justice. The opinions, findings, and conclusions or recommendations expressed are those of the author and do not necessarily reflect the views of the Department of Justice.

I would also like to acknowledge the Saudi Arabian Cultural Mission (SACM) for providing me with a graduate fellowship during the course of my studies.

Dedication

To my wife Sarah and son Hitham, thanks for giving me the strength to keep pushing on and I am excited about spending more time with both of you. Thanks for being patient. My parents, Samir and Awatif, thanks for believing in me all these years and helping me in every way you can.

Acknowledgments

I would like to thank Dr. Charles Bostian for being a great friend and advisor and for believing in me when I needed it the most.

Shereef, thanks for all the great conversations we have had and all the great insight you have given me all these years TMV! Alex sometimes you drive me absolutely nuts too! Nick you should keep being random for as long as you can ... and thanks to both of you for bailing me out from that Python problem from a long time ago. Thanks to all the CWT family Judy, Sujit, Qinqin, Ying, Terry, and Gladstone thanks for being great friends and co-workers.

Randy Marchany, thanks for convincing me to talk to Dr. Bostian when I needed to find my path in graduate school.

I would also like to thanks my committee members: Dr. Cameron Patterson thanks spending the time to listen and answer my many questions for all these years. Dr. Scott Midkiff by being my undergraduate research advisor you helped pave my way into graduate school. Dr. Baumann working with you and Dr. Nathaniel Davis on developing ENGE 1104 was one the most eye opening experiences in my academic career, thanks for believing in sophomores. Dr. Michael Taaffe, thanks for introducing me to the beautiful world stochastics and probability.

All photographs and illustrations in the dissertation have been taken and created by the author.

Contents

Grant Information iii

Dedication iv

Acknowledgments v

List of Figures x

List of Tables xiii

Acronyms xvi

1 Overview 1

1.1 Introduction . . . 1

1.2 Motivation . . . 5

1.3 Problem Statement . . . 6

1.4 Goals . . . 6

1.5 Contributions . . . 8

1.6 Outline . . . 9

2 Literature Review 10

2.1 Introduction . . . 10

2.2 Radio Representation . . . 10

2.3 Radio Design Tools . . . 11

2.4 Custom Processors . . . 12
2.5 Models-of-Computation . . . 13
2.6 Conclusion . . . 14

3 Models of Computation **15**
3.1 Dataflow . . . 15
3.1.1 Kahn Process Networks . . . 16
3.1.2 Synchronous Dataflow . . . 16
3.1.3 Boolean Controlled Dataflow and Integer Controlled Dataflow . . . 20
3.1.4 Cyclo-Static Dataflow and Cyclo-Dynamic Dataflow . . . 22
3.1.5 Parametrized Synchronous Dataflow . . . 23
3.2 Process Algebra . . . 24
3.2.1 Communicating Sequential Processes . . . 24
3.2.2 Alternative Process Algebra . . . 28
3.3 Summary . . . 28

4 Digital Communication **29**
4.1 Protocol Layering . . . 29
4.1.1 OSI Model . . . 29
4.2 Binary Phase Shift Keying . . . 30
4.3 Differential Binary Phase Shift Keying . . . 31
4.4 Pulse Shaping . . . 31
4.5 Summary . . . 32

5 Tools **34**
5.1 Software . . . 34
5.1.1 GNU Radio . . . 34
5.1.2 Occam . . . 34
5.1.3 Ptolemy . . . 35
5.2 Hardware . . . 36

5.2.1 Universal Software Radio Peripheral 37
5.3 Summary 38

6 Framework **39**
6.1 Overview 39
6.2 Model Interpreter 41
6.2.1 Language 41
6.2.2 Model Definitions 41
6.2.3 Interpretation 43
6.3 Model Generator 43
6.4 Core Framework 44
6.4.1 First Stage 45
6.4.2 Second Stage 46
6.4.3 Customized GNU Radio 55
6.5 Interface 56
6.6 MIP Solver 56
6.7 Data Visualization 57
6.8 Perspective 57

7 Example Application Use **59**
7.1 Testing 59
7.1.1 Reactive Models 59
7.1.2 DBPSK Radio 60
7.1.3 SDF Graphs 60
7.2 Performance Measurement Runs 63
7.2.1 Knob 63
7.2.2 Meters 64
7.3 Architecture 65
7.3.1 DBPSK Transmitter 65

7.3.2 DBPSK Receiver 66
7.3.3 DBPSK Link 66
7.4 Reactive Models 67
7.4.1 Processes and Meta-Data 67
7.4.2 DBPSK Transmitter 68
7.4.3 DBPSK Receiver 69
7.4.4 DBPSK Link 70
7.5 Real-Time (SDF) Model 72
7.5.1 DBPSK Transmitter 72
7.5.2 DBPSK Receiver 78
7.5.3 DBPSK Link 85
7.6 Concluding Remarks 92

8 Conclusion and Recommendations for Future Work 93
8.1 Summary 93
8.2 Conclusion 94
8.3 Contribution 94
8.4 Future Work 95

Bibliography 96

List of Figures

1.1 Implementation Domains of CE and SDR. 2
1.2 Car Example Domain Interplay. 3
1.3 Car Example Domain Interplay. 3
1.4 Proposed Domain Definition for CE and SDR. 4
1.5 Computational Knobs and Meters Usage. 8

3.1 Dataflow Graph. 16
3.2 Synchronous Dataflow Graph. 17
3.3 SDF Graph with Deadlock. 20
3.4 SDF Graph with no Deadlock. 20
3.5 Inconsistent SDF Graph. 21
3.6 Consistent SDF Graph. 21
3.7 BDF Graph Example. 22
3.8 SDF Graph Control Example. 22
3.9 CSDF Graph Example. 23

4.1 Raised-Cosine Pulse Shaping Normalized Power Spectral Density. 33

5.1 GNU Radio Framework. 35
5.2 Ptolemy Framework. 36
5.3 Software Defined Radio Platform. 37

6.1 Framework Software Architecture. 40
6.2 Model Generation Domain Mapping Process. 44

6.3 Sequence of Events for CE/Framework Interaction 45
6.4 First Stage Analysis 46
6.5 Second Stage 47
6.6 Flowgraph Perspective After the First Stage 48
6.7 GNU Radio Expanded Flowgraph 49
6.8 Setting Flowgraph Relative Rates Starting at Data Source Block 51
6.9 Flowgraph Relative Rate that Must be Fixed 52
6.10 Final Flowgraph with Correct Relative Rates 52
6.11 First Iteration of Setting Relative Rate Starting at Block 1. 53
6.12 Second Iteration of Setting Relative Rate Starting at Block 3. 53
6.13 Flowgraph Relative Rate Mismatch. 54
6.14 Flowgraph Relative Rate After Algorithm is Done. 54
6.15 Modified Start Up Sequence of Flowgraphs in GNU Radio. 56
6.16 SDF Graph Used to Calculate Repetition Vector. 57

7.1 DBPSK Transmitter Constellation. 61
7.2 Inconsistent Graph. 61
7.3 Consistent Graph. 62
7.4 Buffer Scaling. 63
7.5 DBPSK Transmitter 65
7.6 DBPSK Receiver 66
7.7 DBPSK Link Over an Additive White Gaussian Noise Channel 66
7.8 DBPSK Transmitter Reactive Model 68
7.9 DBPSK Receiver Reactive Model 69
7.10 DBPSK Link Reactive Model 70
7.11 DBPSK Transmitter SDF Model 72
7.12 Memory Utilization for DBPSK Transmitter 75
7.13 Latency for DBPSK Transmitter 76
7.14 Throughput for DBPSK Transmitter 77

7.15 Reconfiguration for DBPSK Transmitter 78
7.16 DBPSK Receiver SDF Model 78
7.17 Memory Utilization for DBPSK Receiver 82
7.18 Latency for DBPSK Receiver 83
7.19 Throughput for DBPSK Receiver 84
7.20 Reconfiguration for DBPSK Receiver 85
7.21 DBPSK Link SDF Model 85
7.22 Memory Utilization for DBPSK Link 89
7.23 Latency for DBPSK Link 90
7.24 Throughput for DBPSK Link 91
7.25 Reconfiguration for DBPSK Link 92

List of Tables

4.1 Differential Coding 31

7.1 Channel Connections for DBPSK Transmitter Reactive Model. 69

7.2 Channel Connections for DBPSK Receiver Reactive Model. 70

7.3 Channel Connections for DBPSK Link Reactive Model. 71

Acronyms

ADC Analog-to-Digital Converter

AWGN Additive White Gaussian Noise

BDF Boolean-Controlled Dataflow

BPSK Binary Phase-Shift Keying

CCS Calculus of Communicating Systems

CDDF Cyclo-Dynamic Dataflow

CE Cognitive Engine

CORBA Common Object Request Broker Architecture

CR Cognitive Radio

CSDF Cyclo-Static Dataflow

CSP Communicating Sequential Processes

DAC Digital-to-Analog Converter

DART Dataflow for Run-Time

DBPSK Differential Binary Phase-Shift Keying

DIF Dataflow Interchange Format

DSP Digital Signal Processor

FIFO First In First Out

FPGA Field-Programmable Gate Array

FRS Family Radio Service

FSM Finite State Machine

GLPK GNU Linear Programming Kit

GPP General Purpose Processor

GPU Graphic Processor Unit

IDF Integer-Controlled Dataflow

IDF Integer

ISI Intersymbol Interference

ISO International Organization for Standardization

JTRS Joint-Tactical Radio System Program

KPN Kahn Process Networks

LTE Long Term Evolution

LWDF Lightweight Dataflow

MIMO Multiple-Input and Multiple-Output

MIP Mixed Integer Programming

MIP Mixed Integer Program

MISO Multiple-Input and Single-Output

MoC Models of Computation

NRZ Nonreturn-to-Zero

OSI Open System Interconnection

PAPS Periodic Admissable Parallel Schedule

PASS Periodic Admissable Sequential Schedule

PLL Phase-Locked Loop

PSDF Parametrized Synchronous Dataflow

RKRL Radio Knowledge Representation Language

RRC Root-Raised-Cosine Filter

SCA Software Communications Architecture

SDF Synchronous Dataflow

SDF Synchronous Dataflow

SDR Software Defined Radio

SIMO Single-Input and Multiple-Output

SISO Single-Input and Single-Output

STS Single-Thread-Scheduler

SWIG Simplified Wrapper and Interface Generator

TPB Thread-Per-Block

UAV Unmanned Aerial Vehicle

USRP Universal Software Radio Peripheral

WDL Waveform Description Language

Chapter 1

Overview

1.1 Introduction

The cultural reliance on wireless communication has transformed from purely voice services to data based services. As demand increases for data-based applications and Internet use, the wireless industry is having constantly to provide higher bandwidth and capacity for its users. Advances in computing technology has made it is possible to perform radio functions classically implemented in hardware to be done in software. The concept of Software Defined Radio (SDR) allows communication systems to be implemented partially or fully in software [1]. Such radio implementations lend themselves to flexibility, where software modifies the underlying radio operation. A Cognitive Engine (CE) can be integrated with an SDR to enhance its performance by adding a layer of cognition to monitor and adapt radio performance. Through this adaptation process, a radio is able to learn from its environment about suitable configurations. This ability to integrate model-based reasoning with SDR results in a Cognitive Radio (CR) [1].

Radio performance can be reconfigured by modifying a set of *knobs*, writable parameters like filter taps, gain, ... etc, and the performance can be monitored by observing a set of *meters*, readable parameters like throughput [2] [3]. By relying on software to control a radio's knobs and meters, the overall radio definition becomes more flexible. However the design and analysis is complicated by incorporating computational considerations into what is classically a hardware only system. As a result, the skill-sets and expertise necessary to build these radios is more diverse, requiring multi-domain expertise ranging from software to RF development.

SDR applications vary from handheld radios to base stations; therefore it is important to acknowledge the various computational requirements spanning these application. However, in the development of CR radios, the ability is lacking to expose additional computational knobs and meters, thus allowing the CE to explore alternative radio solutions by implement-

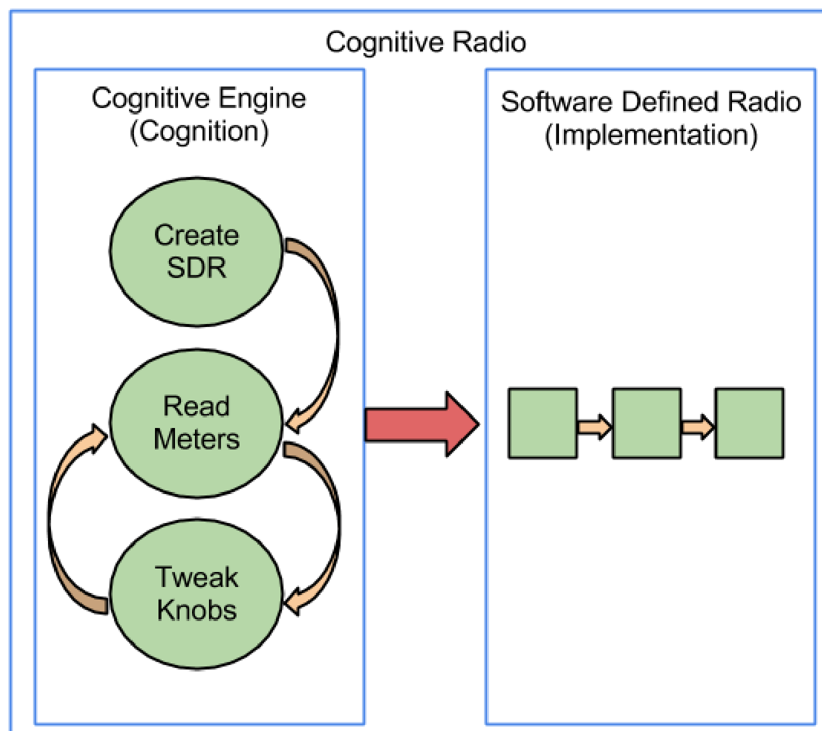

Figure 1.1: Implementation Domains of CE and SDR.

ing designs varying in memory, latency, and reconfiguration time requirements. In order to represent such knobs and meters, it is important to model the computational domains of both CEs and SDRs. Figure 1.1 represents the current understanding of how the CE and SDR domains interact to develop CRs. A CE behaves like a reactive outlook on a radio implementation; it represents how the individual components are stitched and configured. At the same time, the SDR represents a real-time perspective of the same radio based on the flow of data in the system.

A reactive representation provides a view of how the system is interacting with its environment [4]. As an example of a reactive model, consider a car and a driver. When the car approaches a red traffic light, the driver must stop it. The notion of stopping is a reactive perspective of what the driver needs to do. It does not define the mechanical motions needed to stop the car nor does it represents the timing necessary to stop before crossing the light; see Figure 1.2. The physical act of stopping the car has a real-time component; it requires the coordination of mechanical motions which will allow the driver to stop the car in a timely fashion. The reactive system outlook, wanting to stop the car, and the real-time system implementation, the physical act of stopping the car, are both different outlooks, or models, of the same system. See Figure 1.3, where the real-time process of stopping a car is implied by the reactive model of a stopped car.

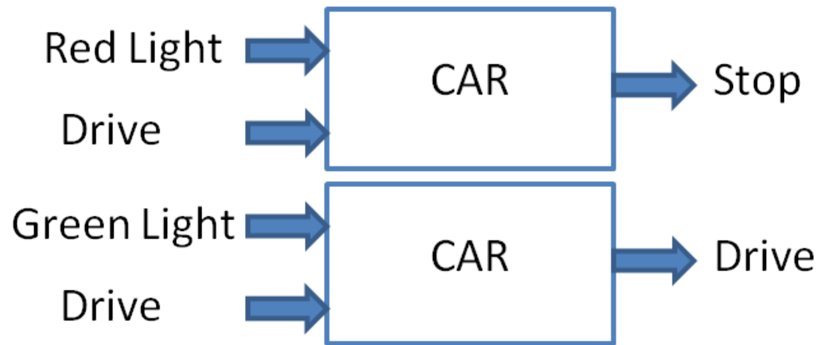

Figure 1.2: Car Example Domain Interplay.

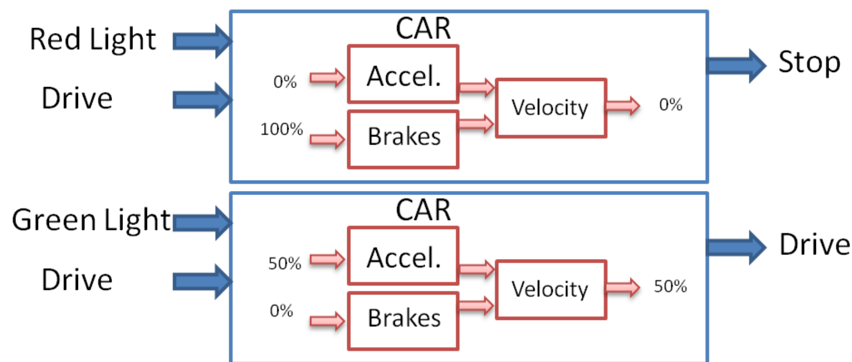

Figure 1.3: Car Example Domain Interplay.

Relating this to CR, in the reactive model, the CE does not have a notion of timing or specific implementation details, while in the real-time model the SDR contains the implementation specific information which is coupled with timing dependence information. Please note that the reference to an SDR as a real-time system is based on the signal processing perspective of sampling rates associated with the radio implementation relating to input/out rates. This is in contrast to the traditional definition of real-time where tasks are supposed to execute before the elapse of a pre-defined *deadline* [5]. While scheduling signal processing tasks and being able to define execution deadlines for them is important for SDR implementation, it is not directly addressed in the work discussed in this document.

A modified CR system specification is shown in Figure 1.4. In a CR, a CE represents the layer of intelligence generating and modifying radio configurations and does not need to be aware of the detailed implementation of the SDR flowgraphs being generated. However, it should be aware and configure parameters such as the symbol rate for digital modulations, sampling rates, interpolation factors, and decimation factors since such parameters affect the overall radio design and the over-the-air radio characteristics.

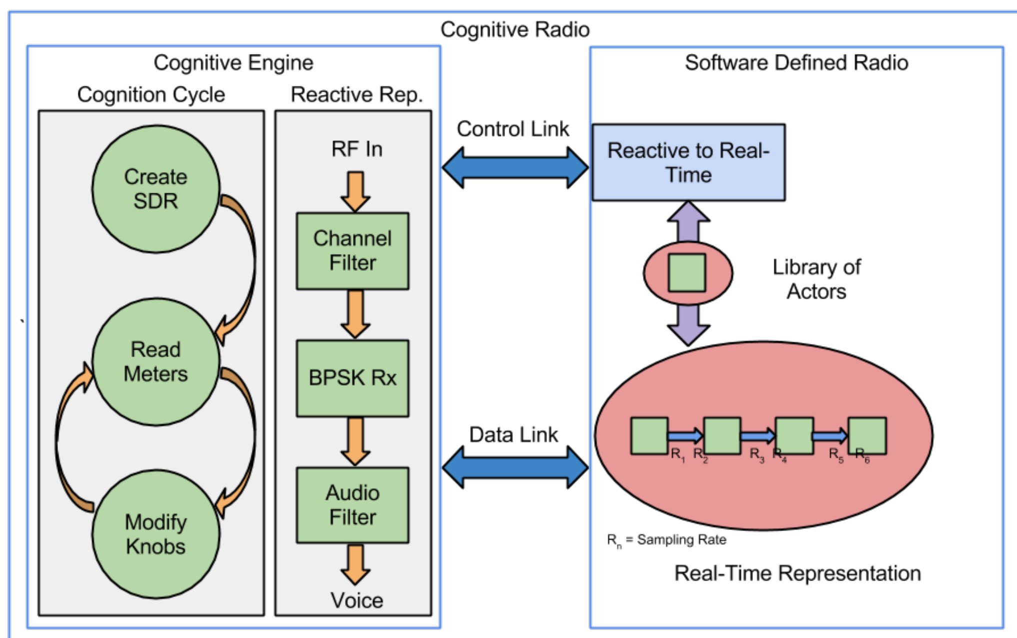

Figure 1.4: Proposed Domain Definition for CE and SDR.

In the work described here, a control layer between the CE and the SDR reads the reactive system model and reinterprets it to a real-time system model by using the *meta-data*, parameters from the reactive model allowing the real-time model generation, *e.g.*, sampling rates. A CE is able to pass desired implementation goals, such as a range of memory utilization, throughput, and reconfiguration times, and the SDR domain will attempt to meet these goals. The SDR domain is able to evaluate the generated radio, by observing the meters, and measure the associated performance metrics and return them to the CE. Essentially, the CE requests a radio configuration using a reactive model and the SDR evaluates the quality of the synthesized radio using the real-time model. This type of analysis allows CEs to explore implementations with various memory utilization, throughput, and runtime reconfiguration time values while abstracting the details of the SDR implementation.

This work models and implements both a CE and SDR system in appropriate models of computation. It explains the implementation of a framework allowing CR designers to specify reactive radio models and synthesize real-time SDR implementations from these models.

1.2 Motivation

CEs should be able to control and monitor radio performance through a system of knobs and meters. The ability to create computational knobs and meters leads to additional degrees of freedom available for CEs to explore in requesting alternative radio implementations that are able to adapt to changing application and RF environmental needs. By allowing CEs to monitor and control computational aspects of their radio implementations, we make them able to better understand and assess the computational capabilities of their underlying hardware. Knowledge of computational limits and configuration parameters allows CEs to explore radio solutions which can account for computational hardware limitations in addition to the RF environment and application needs.

Observing platform computational performance becomes even more relevant as SDR software tools are able to run on a wider variety of hardware platforms, from x86 processors to more resource restricted embedded processors. Example of such software tools are GNU Radio [6] running on embedded processor [7], OSSIE [8] running on an Field-Programmable Gate Array (FPGA) platform as discussed in[9] and the IRIS [10] running on an FPGA platform as discussed in [11].

Since CR applications are able to run using various hardware platforms, it is important for CEs to be aware of the resource capabilities and limitations of the computational hardware on which they are running. An example application scenario is one where a desktop based CR application was migrated to an embedded platform [12]. While the migration process requires manual software customization to make use of the embedded processor, the ability of having computational knobs and meters could help in further refining system performance.

There are also situations where CR are integrated with other application, *e.g.*, Unmanned Aerial Vehicles (UAVs) as in [13]. By having a CR be part of this larger system, the CE needs to allocate resources both to manage UAV flight and resources to manage radio implementation. Therefore, CEs must allocate computational throughput, latency, and memory utilization to distribute resources between running the flight management system and the radio implementation software. The use of computational knobs and meters helps CEs meet such objectives. With the incorporation of video and voice radio applications, computing resources needs to be managed between multimedia and radio processing. With CR technology being integrated into other areas such as medical sensing and smart grids [14], CRs will need to manage application computational needs.

Also by allowing CEs to have a reactive perspective of radio implementation and allowing the SDR to be concerned with the implementation details in a real-time domain, it becomes possible to have the computational knobs and meters as the only form of interaction between both domains. This outlook abstracts the implementation details for a CE allowing it to focus on observing and reconfiguring underlying radio implementations instead of being coupled to and aware of the implementation level details.

1.3 Problem Statement

How can SDRs provide CEs with meaningful computational knobs and meters providing insight into the radio implementation?

The problem explored in this dissertation addresses the development of reactive radio flowgraph models for CEs and their translation of radio models into real-time models for SDR implementation. In addition, through this model transformation process and how it addresses SDR models can provide computational knobs which allow CEs to influence the physical implementation and be able to monitor the radio reconfiguration results through meters which provide perspective via knobs on the computational quality of the radio implementation.

While there is previous work exploring the application of models of computation to SDR, the concept of computational knobs and meters has not been greatly explored. This is discussed in more detail in the next chapter. More specifically such knobs and meters should allow meaningful exploration of computational aspects of CR implementation. In addition, consideration of the ability to map from reactive radio models to real-time, or dataflow, based models is a unique aspect of this dissertation.

1.4 Goals

The first goal of this dissertation is to develop a framework which allows CEs to define reactive radio flowgraphs using the Communicating Sequential Processes (CSP) model and translate the radio definitions to Synchronous Dataflow (SDF) based SDR implementations. The framework generates SDR simulations for the Ptolemy software and generates GNU Radio models used for implementing radios running over-the-air. A first generation Universal Software Radio Peripheral (USRP) [15] serves as a programmable front-end.

An important aspect of this work is the introduction of a new knob and multiple new meters which allow cognitive engines to monitor the actual computational performance of an SDR while configuring it for chosen performance. The following is the list of the introduced knobs and meters:

- **Knob**:
  - *Memory Buffer Scaling:* The framework developed here allows the definition of minimum buffer sizes needed to relay data between the radio flowgraphs components. This knob allows CEs to define a scalar multiple buffer sizes derived from the calculated minimum sizes. Increasing buffer sizes can affect the throughput and latency of the radio implementation.
- **Meters**:

- *Memory utilization:* The actual memory utilization that the SDR is able to achieve.
- *Latency:* The computational latency of the flowgraph based on the SDR implementation.
- *Throughput:* The computational throughput achieved by the SDR implementation.
- *Graph consistency:* Indicates whether the relative data rates between block ports will allow fixed buffer sizes which can accommodate the memory needs of the radio flowgraph during execution.
- *Radio Reconfiguration Time:* Reconfiguration in the work presented entails tearing down and reconstructing a radio graph. This is not reconfiguring the radio without disrupting execution as is done in *runtime* reconfiguration. The semantics of the models used in this work do not support runtime reconfiguration. While the models do not support runtime reconfiguration, they do provide valuable, properties including deadlock detection, detecting sampling rate mismatch, and being able to calculate memory requirements for radio flowgraph execution. Since CEs can maintain state information they can provide state information to an SDR while it is being torn down and rebuilt.

CEs are also able to define desired ranges of these meters (*i.e.*, memory utilization, reconfiguration time, latency, and throughput) and have the framework explore buffer scaling factors necessary to realize solutions within these ranges for future use. By using such framework, CEs are able to explore various implementations of radio flowgraphs and save the configurations necessary to recreate them in the future. This allows CEs to use the framework during their *training* or *learning* processes, basically when they are developing suitable solutions from various base scenarios, so they can rely on these implementations in future situations when they are needed.

Looking at Figure 1.5, we can see how CEs can configure the buffer scaling knob and be able to observe its effect on the meters. The graphs in the figure represent expected meter trends and the highlighted regions in the graph represent the target meter ranges for the radio application as determined by the CE. Throughput can increase by increasing the buffer sizes since the flowgraph is able to process more data at each execution; however, there will be a limit to the amount of throughput improvement possible. The latency increases as the buffer scaling increases since data will take longer to traverse from the input to the output of a flowgraph because larger buffer sizes implies longer processing. The total memory utilization of a flowgraph will naturally increase as the buffer scaling increases since more data needs to be saved in the buffers. The reconfiguration time is not directly dependent on the buffer scaling since it does not affect other aspects of flowgraph allocation, *e.g.*, parsing Occam files, generating GNU Radio models, parsing GNU Radio flowgraphs, allocating block details, amongst many other steps. Therefore the reconfiguration time meter provides an

estimate of how much time it takes to reconfigure the current graph and CEs can use this knowledge to determine whether it is worth reconfiguring a flowgraph at the expense of the measured reconfiguration times.

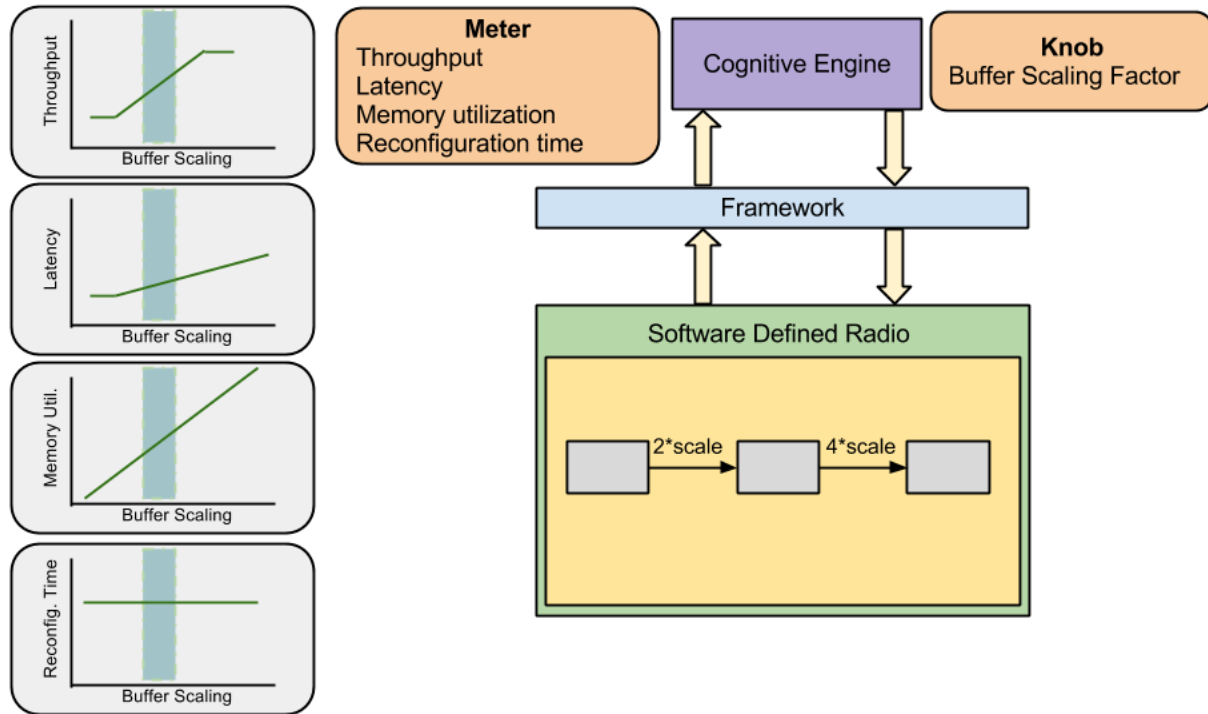

Figure 1.5: Computational Knobs and Meters Usage.

1.5 Contributions

The contributions of this dissertation include:

- Defining radio flowgraphs in a suitable reactive model which can be used by CEs.
- Defining real-time SDR flowgraphs using an appropriate real-time, or dataflow-based, model.
- Developing a framework which allows the model transformation between the CE and SDR radio models.
- Developing a set of computational knobs and meters which allow a CE to monitor and configure the computational performance of an SDR.

- Designing digital radio transmitter, receiver, and link using the developed framework and observe the performance measures using the developed knobs and meters.

1.6 Outline

The remaining chapters in the dissertation are as follows:

- Chapter 2: Literature review.
- Chapter 3: A summary of models of computation.
- Chapter 5: A summary of the tools used in the work presented.
- Chapter 6: Implementation details of the proposed framework.
- Chapter 7: Provides example scenarios of implementing Differential BPSK radio transmitter, receiver, and link using the proposed framework.
- Chapter 8: Provides a summary of the work presented and some concluding remarks.

Chapter 2

Literature Review

2.1 Introduction

This chapter summarizes relevant research found in literature. It includes the following areas:

- **Radio representation**: Reviews techniques and languages used to describe radio flowgraphs.
- **Radio design tools**: Reviews software tools and design mechanisms which can be used to enhance the development of radio flowgraphs.
- **Custom processors**: Reviews some of the custom processors intended to address computational and power needs of SDR applications.
- **Models-of-computation**: Review of works incorporating models-of-computation with radio flowgraph design.

2.2 Radio Representation

The Software Communications Architecture (SCA) [16] is a standard defined by the Joint-Tactical Radio System Program (JTRS) to address the issue of software integration and compatibility but it does not necessarily provide a formal computational model. The work presented in this dissertation uses computational models for CEs and SDRs to provide radio flowgraph definitions and to provide computational knobs and meters as provided by the underlying models used.

Radio Knowledge Representation Language (RKRL) [17] addresses the ontology of describing radio functionality and features. However, it does not provide a formal computational model which can be used to describe radio flowgraph definitions as is used in this dissertation.

SPEX [18] is a programming language intended for SDR, the principle is helpful in making better use of processor architecture while writing SDR based applications. By itself, it does not cover model based representation of SDR applications.

2.3 Radio Design Tools

OSSIE [8] is an open source SDR design tool based on the SCA specification. The software enables distributed processing of radio application by using Common Object Request Broker Architecture (CORBA) to facilitate message passing between the various signal processing functions [19]. OSSIE is able to run on desktop/laptop computers and embedded targets, but the use of the SCA and CORBA for signal processing does not provide a formal model of computation. GNU Radio [6] is another open source SDR development tool which can be used for laptop or embedded processing platforms. While the GNU Radio scheduler is data driven, it does not enforce specific computational models for radio implementation. There are examples of using dataflow based scheduling using GNU Radio as in [20]; however, this work does not expose computational knobs and meters as will be discussed in greater detail in the models-of-computation section. IRIS [10] is another software framework for developing SDR applications which was not initially available publicly but was just released as an open source framework in 2013. IRIS allows signal processing functionality to be divided into blocks connected via buffers in the same manner as GNU Radio and OSSIE. However, IRIS allows the definition of different *engines*, or execution domains. For example synchronous dataflow is used as a domain for static execution while process networks is used as a flexible physical layer engine allowing radio runtime reconfiguration. An engine for networking applications is also available. While IRIS applies dataflow models of computation for SDR it neither provides an engine for creating reactive radio definitions for CE use, nor does it provide computational knobs and meters from the dataflow domain for CE use. While IRIS supports synchronous dataflow, which is used in this dissertation, it was not publicly available when this dissertation work started so GNU Radio was selected as the framework for implementing radio flowgraphs.

Component based design [21] promotes software reuse by defining *components*, or basic computational functions, and implementing systems by assembling these components. Examples of component based design usage in SDR include [22] and [23]. The ability to reuse components is important in designing SDR systems, and the software suites GNU Radio, OSSIE, and IRIS support these principles by providing signal processing blocks, or components, for designers to develop applications. However, software reuse does not address the issue of defining appropriate models for CR nor does it address the issue of computational knobs and meters.

The Mathworks [24] Simulink software can be used to model, simulate, and implement system behavior [25]. Using Simulink, C/C++ code can be generated, using the Simulink Coder [26], for hardware-in-the-loop testing, where code running on the target hardware can be part of the simulation. In addition HDL code, for FPGAs can be generated using [27], and library functions written for specific computing devices can be instantiated, *e.g.* Xilinx System Generator [28] and TI eXpressDSP [29]. The Mathworks software suite is used in SDR for programming embedded platforms, including General Purpose Processors (GPPs), Digital Signal Processors (DSPs), and FPGAs. Examples of its use include [30] and [31]. Simulink enables component reuse, rapid system development, and hardware integrated simulations, however it does not support, or at least does not provide semantic support to the underlying computational models used. Therefore it is difficult to control and observe computational knobs and meters in such development environments.

In [32] and [33], Marojevic et al., present a framework for mapping signal processing functions to heterogeneous multiprocessors. In their framework, it is possible to account for processors with different processing capabilities and different inter-processor communication bandwidths. Such work can provide more computational knobs and meters to this dissertation work which can be used for multi-processor based CRs.

In [34], the author discusses that computational complexity can be reduced by reducing the radio flowgraph complexity when the RF channel or application allow it, *e.g.*, reducing number of filter taps. This work complements the work discussed in this dissertation in that [34] allows CEs to control algorithm complexity in response to RF channel conditions and application needs.

2.4 Custom Processors

In [35] Lin et al. introduce SODA, a multicore processor for SDR applications. The processor handles data processing using its multiple cores and has a dedicated core which controls the overall data processing in the remaining cores. Other examples of custom SDR processors includes the MONTIUM tile processor [36], Sandblaster [37], and the Infineon MuSIC [38]. Custom processors attempt to meet the computational throughput needed by wireless protocols while attempting to maintain low power consumption, thus making them useful for both mobile and base station applications. The work presented in this dissertation does not address the issue of necessary processor architectures to provide the computational density needed for SDR applications. However, part of the dissertation analyzes the ability of modifying computational parameters, buffer scaling, to provide implementations of varying memory utilization, throughput, and latency measures.

2.5 Models-of-Computation

In [39], Berg et al., analyze the applicability of various models-of-computation for SDR applications. The authors discuss the following models: synchronous dataflow, cyclo stationary dataflow, heterochronous dataflow, cyclo dynamic dataflow, and parametrized synchronous data flow. They explore the issues and benefits of each model in the context of designing a Long Term Evolution (LTE) [40] receiver but do not discuss implementation. In [41], Siyoum et al. discuss the use of *scenario aware dataflow* models for LTE systems. These models use Finite State Machines (FSMs) to define different computational *scenarios*, which are different operational modes that a radio can expect during its use [42] as for an LTE receiver. Each FSM state, or scenario, is then implemented using the Synchronous Dataflow model. In [43], Hsu et al. apply the mixed-mode vector-based dataflow model to simulate the LTE physical layer. By using mixed static and dynamic dataflow models, the authors are able to use static scheduling techniques used in static dataflow models while making use of dynamic dataflow models to express conditional aspects of the LTE physical layer. The authors present alternative models that can be used for implementing SDR systems which can complement the work presented in this dissertation. This dissertation looks at radios model in a CE domain and how to transform them to appropriate models in the SDR domain.

In [20] and [44] Zaki et al., discuss implementing SDR flowgraphs using NVIDIA Graphic Processor Units (GPUs) [45]. GPUs have multiple processors which allows the parallel execution of algorithms. The authors developed GPU accelerated signal processing functions in the GNU Radio software. In addition the authors use an integer program to schedule signal processing functions on cores while taking into account communication times between processors, signal processing function execution times, and the number of processors in the GPU. In addition, the authors use the Dataflow Interchange Format (DIF) language [46], a language for defining dataflow based graphs. They also explore the affect of *vectorizing*, or scaling, of buffers on flowgraph performance. Their incorporation of dataflow analysis is similar to the work presented in this dissertation, as is exploring the effect of buffer scaling. This work is the most relevant to the dissertation in terms of similarity. However, instead of considering local flowgraph resource performance, this dissertation explores exposing local flowgraph performance in the form of computational knobs and meters for CEs. In addition, in this dissertation radio flowgraphs are described using Occam, a reactive language which allows the expression of reactive radio models in CEs versus DIF, a dataflow based language.

In [47], Shen et al., propose a Lightweight Dataflow (LWDF) model to SDR system implementation. LWDF provides libraries which support graph definition and manipulation and actor based communication libraries. The intention is to enable the use of dataflow models in SDR systems by using the LWDF library. This work is beneficial in terms of adding dataflow support to applications which lack it.

In [48], Lin et al. explore the hierarchical composition of SDR flowgraphs, meaning that

each node in the flowgraph is able to contain other dataflow graphs. By allowing hierarchical definition of blocks, SDR graphs with differing sampling rates are to be scheduled separately which the authors state allows more efficient scheduling of the flowgraphs. In addition, the authors explore scheduling signal processing functionality in multiprocessor environment and evaluate the efficiency of different scheduling strategies.

In [49], Palkovic et al., discuss Dataflow for Run-Time (DART), a framework allowing them to reconfigure SDR graphs during execution. The authors argue that runtime reconfiguration reflects the change in desired radio behavior and is a realistic expectation for SDR applications. While the authors provide a framework for performing runtime reconfiguration, they do not present the framework as a formal model-of-computation. This dissertation analyzes the impact of the underlying SDR models on properties such as bounded buffer execution and deadlock detection which are not discussed by [49].

Waveform Description Language (WDL) [50] describes radio functionality using statechart [51], a suitable computational model for describing reactive systems and discusses the generation of C and HDL code to implement necessary signal processing functionality but does not discuss the mapping to an SDR suitable model-of-computation. This dissertation discusses both the reactive representation of CE radio models and it transformation into the real-time, or dataflow, SDR models.

2.6 Conclusion

As we can see from the literature review, applying models-of-computation is already being done in SDR. However, using properties of the models to provide computational knobs and meters instead of using them for local scheduling is a novel contribution. Another contribution is the translation from reactive radio models to real-time, or dataflow based models.

Chapter 3

Models of Computation

Computing has become an integral part of complex systems. Cars, factories, and airplanes are now embedded with computers that interact with the surrounding physical environment and computers have even replaced classical mechanical and/or electrical systems, for example *drive-by-wire* [52] and *fly-by-wire* [52]. Computing applications vary in scope and requirements, therefore different paradigms are needed to implement these applications. Models of Computation (MoC) provide formal semantics for computing; by applying these models, we are able to inherit desirable properties in applications: deterministic behavior, bounded memory execution, deadlock detection, and finite periodic schedules calculation. Therefore, it is necessary to explore suitable MoCs for CR and SDR applications in order to explore the semantics of constructing such radios and investigate how their use can influence radio performance.

CEs are responsible for evolving radio configurations by monitoring, learning, and adapting to their environment. This perspective on the radio is reactive as was previously mentioned. Thus a suitable MoC does not necessarily need to provide timing information but would need to articulate the overall radio construction and define the communication channels between its building blocks. Conversely, SDR presents a real-time perspective on radio implementation therefore a suitable MoC would need to provide a notion of timing, whether it is relative or absolute. In this dissertation, I will explore dataflow based MoCs to find a suitable model for SDR. I will also discuss *Process Algebra* because it offers a way to find suitable CE models.

3.1 Dataflow

A graph, G, is defined as "a triple consisting of a vertex set, $V(G)$, an edge set, $E(G)$, and a relation that associates with each edge two vertices called endpoints" [53]. Dataflow graphs are directed graphs, meaning they have unidirectional paths. Nodes, or actors, represent

functions and data paths are represented by arcs [54]. Dataflow class of MoCs are *data driven*, meaning that computation can proceed as soon as input data is ready, in contrast to being *control driven*, where computation is performed as soon as instructions are available [55]. Dataflow programs are composed of *nodes* or *actors*, representing computational functions, connected by *arcs*, representing the First In First Out (FIFO) buffers connecting the nodes [54], and *tokens* represent the data transferred over the arcs. Nodes *fire*, or execute, once they have input data available for processing. Figure 3.1 illustrates this terminology.

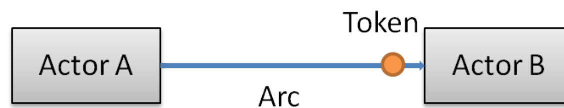

Figure 3.1: Dataflow Graph.

3.1.1 Kahn Process Networks

In the Kahn Process Networks (KPN) model [56], nodes, or *processes*, communicate via FIFO buffers which theoretically have infinite capacity. Processes can write to a channel asynchronously, meaning a writing process can write to the FIFO even if the reading process is not ready. However, a channel read is synchronous, meaning a reading process will block if the writing process is not ready to write. By having asynchronous, non-blocking, writes and synchronous, blocking, reads, the model is determinate as discussed in [56]. It is also assumed that there are no global variables in the model, meaning that communication can only occur between connected processes.

3.1.2 Synchronous Dataflow

SDF is a special case of KPN, where nodes consume and produce a fixed number of tokens [54]. As can be seen in Figure 3.2, integers to the right of a node represent the production rate, the number of tokens produced when a node fires. Integers on the left of a node represent the token consumption rate, the number of tokens consumed after firing. The consumption and production rates do not need to be equal; if the production rate is higher than the consumption rate, then, after the nodes fire, not all tokens are consumed at once. If the consumption rate is higher than the production rate, then a consuming node does not fire until there are enough tokens on its connecting arc.

By allowing fixed production and consumption rates, the SDF model can provide bounded memory execution in addition to other desired properties which are discussed in the following sections.

Figure 3.2: Synchronous Dataflow Graph.

Topology Matrix

SDF graphs can be analyzed to determine whether they can be executed with bounded buffers and whether or not they deadlock, meaning that the flowgraph is not able to run because of insufficient input tokens to actors. In order to perform such analysis, a *topology matrix* must be constructed for an SDF graph. A topology matrix describes the arc connections and the relative production/consumption rates between nodes, where rows correspond to the arcs and columns correspond to the nodes. The (i, j) topology matrix entry corresponds to node j's relative rate on arc i. The production rate of a block would be a positive number while the consumption rate would be negative. The topology matrix of the graph in Figure 3.2 is shown in Equation 3.1. Note that the rows and columns are numbered from zero.

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -2 \end{bmatrix} \tag{3.1}$$

For example, Arc 1 corresponds to the first row in the topology matrix, and the $(0, 0)$ entry, +1, is the positive output rate of Actor 1. The $(0, 1)$ entry, -1, is the negative input rate of Actor 2. The $(0, 2)$ entry is 0 since Arc 1 is not connected to Actor 3. The second row in the topology matrix corresponds to Arc 2, the $(1, 0)$ entry is 0 since Arc 2 is not connected to Actor 1, the $(1, 1)$ entry is +2 since it represents the output rate of Actor 2, and the $(1, 2)$ entry is -2 since it is the input rate of Actor 2. The topology matrix is important in constructing an execution schedule because it describes the changes in the buffer requirements between firings.

Schedule Construction

To manage the execution of an SDF graph, a scheduler must be implemented to manage the execution and allocation of buffers between nodes. The following assumptions about executing SDFs as outlined in [57]:

- SDF graphs are non-terminating, meaning they must execute without deadlocking.
- Non-connected SDF graphs are separate graphs, meaning that they have nothing to do with the execution of each other.

A scheduler that manages the execution of an SDF graph must be a Periodic Admissable Parallel Schedule (PAPS). *Periodic* because SDF graphs are non-terminating, *admissible* because blocks can only execute when data is available, and *parallel* because nodes can execute concurrently and utilize processing resources in parallel. Instead of scheduling SDF actors to execute in parallel, actors can be executed sequentially and the scheduler in this case is a Periodic Admissable Sequential Schedule (PASS).

The rank of the topology matrix provides insight into whether a graph's sample rates are *inconsistent*, basically whether the sample rates will lead to a deadlocked graph or a graph with unbounded buffer sizes. A necessary condition for the existence of a schedule is described in [57] where Equation 3.2 must hold.

$$rank(\Gamma) = s - 1 \tag{3.2}$$

s = Number of blocks in the graph

The node firing sequence can be described using the *firing vector* $v(n)$, where each row in the vector indicates whether a node is firing at time n. Please note that SDF does not provide a global notion of time. Basically *time* n does not correspond to an absolute clock time but corresponds to the relative iteration of the flowgraph schedule. Therefore, timing in this context is relative not absolute in the sense as timing is classically referenced in describing real-time systems. In Equation 3.3, $v(0)$ corresponds to the nodes firing at time $n = 0$. Since the first entry is 1, node 1 is firing at time 0, while nodes 2 and 3 are not firing, since the second and third rows are zero. So at $n = 0$ node 1 is firing, at $n = 1$ node 2 is firing, and at $n = 2$ node 3 is firing.

$$v(0) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, v(1) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, v(2) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.3}$$

We define another vector, $b(n)$, to describe the FIFO buffer sizes, at time n. As defined in Equation 3.4, $b(n)$ describes the new buffer requirements between executions of the graph. Where $b(n+1)$ is the buffer size requirement at time $n+1$, $b(n)$ is the buffer size requirement at time n, Γ is the topology matrix, and $v(n)$ is the firing vector at time n.

$$b(n + 1) = b(n) + \Gamma * v(n) \tag{3.4}$$

To ensure that the graph can execute with bounded buffer sizes, we are interested in observing its behavior over an entire execution period. Therefore, instead of looking at the individual firings at each n, we look at the total firings of nodes in a period, p. In Equation 3.5, the q vector corresponds to the *repetition vector*, a vector detailing how many times each

block should fire during the course of a flowgraph period. Equation 3.5 looks at the firing behavior of all blocks in the course of a single cyclic execution period, p, where q-vector is the summation of all graph firing vectors, $v(n)$, over a period, p.

$$q = \sum_{n=0}^{p-1} v(n) \tag{3.5}$$

Since the schedule is periodic, we can rewrite Equation 3.4 in the form of Equation 3.6 as described in [57].

$$b(np) = b(0) + n\Gamma * q \tag{3.6}$$

To ensure that an SDF graph can execute with bounded buffer sizes, we need to ensure that the buffer sizes remain the same between graph execution periods [57]. This implies that we want $b(np) = b(0)$ meaning that the initial buffer space needed by the graph should remain the same after each execution period of the graph. This implies the condition in Equation 3.7, where $\mathbf{0}$ is a vector of all zeros.

$$\Gamma * q = \mathbf{0} \tag{3.7}$$

By finding the non-trivial vector, q, in the nullspace of the topology matrix, Γ, we are able to find the number of firings necessary for each actor for the graph to execute with bounded buffer sizes.

A sufficient condition for the existence of a PASS for an SDF graph is given in Equation 3.8, which means that during the course of a graph period, each block needs to fire as many times as dictated by the repetition vector q, which is calculated by Equation 3.7. In Equation 3.8, $\mathbf{1^T}$ is a row vector of all 1's used to sum the entries in the q vector.

$$p = \mathbf{1}^T q \tag{3.8}$$

If the scheduler detects a deadlock before executing the blocks as specified by the q vector, then the SDF graph is deadlocked and no PASS exists [57]. An example of a deadlocked graph can be seen in Figure 3.3. The graph can be fixed by adding an initial token on Arc 2 as can be seen in Figure 3.4

The flowgraph in Figure 3.5 illustrates a situation where tokens on Arc 3 will accumulate and eventually require an infinitely sized buffer. This is because Actor 3 consumes one token from Arc 2 and one token from Arc 3 after each firing but Actor 1 produces two tokens on Arc 3 after each firing. There will be one unconsumed token on Arc 3 after each firing, eventually causing an infinite size buffer for Arc 3. The flowgraph can be fixed by changing the output rate for Actor 2 as shown in 3.6.

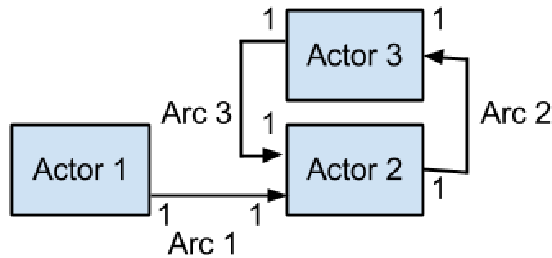

Figure 3.3: SDF Graph with Deadlock.

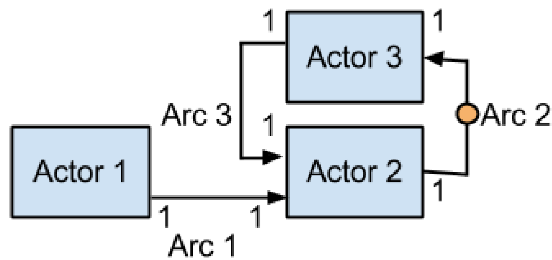

Figure 3.4: SDF Graph with no Deadlock.

3.1.3 Boolean Controlled Dataflow and Integer Controlled Dataflow

Boolean-Controlled Dataflow (BDF) [58] extends the SDF MoC by adding boolean controlled actors, namely *switch* and *select* actors. The use of such control actors means that *control ports* must be used to control these actors. Control ports accept boolean data, either *TRUE* (T) or *FALSE* (F) to select between the two possible inputs for a select actor and two possible outputs for a switch actor. Figure 3.7 provides an example BDF graph showing how the *switch* and *control* actors can be used. This is a situation where Actor 2 might be used to detect a packet header and Actor 3 might be processing the body of the packet. The switch actor will pass the tokens to the appropriate actor while the select actor routes the data from the actor currently processing data.

An SDF graph can be constructed to address the same application as in Figure 3.7; however, the SDF MoC does not support selective execution of actors. Therefore, to accomplish the desired functionality, when Actor 2 is processing a packet header Actor 3 runs and generates dummy tokens just to comply with SDF semantics. Actor 4 can be alerted through a control port that Actor 3 is passing dummy tokens. The SDF example is illustrated in Figure 3.8.

For a BDF graph, the consumption and production rates of actors are either fixed, such as the SDF case, or else are two-valued function dependent on the control input of an actor, *e.g.*, for a switch actor if control port value is F then the F output port is selected; if the

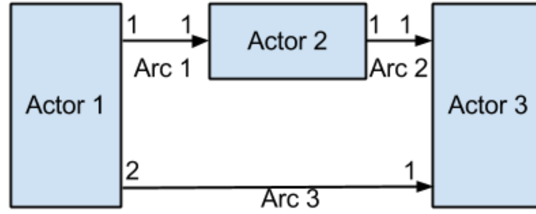

Figure 3.5: Inconsistent SDF Graph.

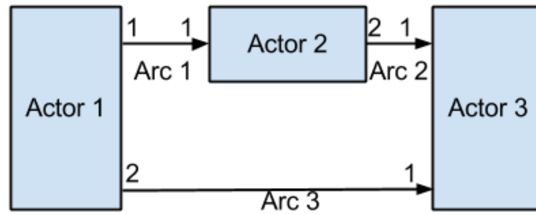

Figure 3.6: Consistent SDF Graph.

control port value is T then the T output port is selected.

In forming the topology matrix of BDF graphs, the statistical quantity p_i is the long term proportion of TRUE tokens over the control port of the *switch* or *control* actors [58] and i indicates the referenced control port. For example, the topology matrix for the BDF graph in Figure 3.7 is constructed in Equation 3.9

$$\Gamma = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & (1-p_1) & 0 & 0 \\ 0 & 0 & -1 & 0 & p_1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -(1-p_2) & 0 \\ 0 & 0 & 1 & 0 & 0 & -p_2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \tag{3.9}$$

If a valid non-trivial repetition vector exists, using Equation 3.7, for any p_i value then the BDF graph is *strongly consistent*. However, if a repetition vector only exists for some values of p_i then the graph is *weakly consistent*. It is important to note that the authors in [58] indicate that it can be impossible to determine whether a BDF graph can be scheduled with bounded memory.

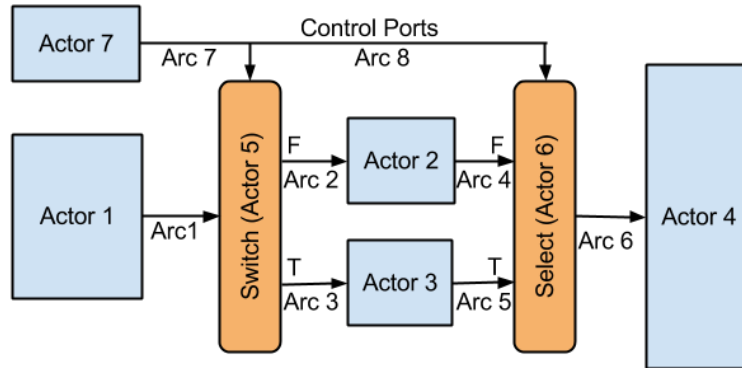

Figure 3.7: BDF Graph Example.

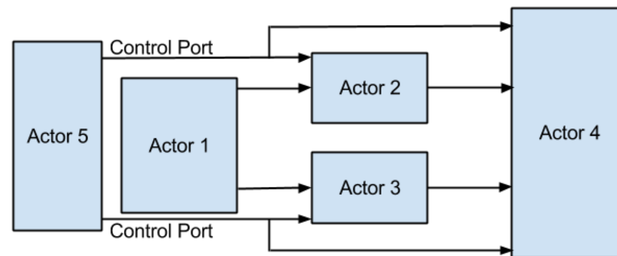

Figure 3.8: SDF Graph Control Example.

The Integer (IDF) model [59] extends BDF by allowing control port values to have integer rather than just boolean values. This extension allows the *switch* actor to have more than two outputs and the *select* actor to have more than two inputs. This extension allows the construction of loops where control port tokens can represent the number of loop iterations.

3.1.4 Cyclo-Static Dataflow and Cyclo-Dynamic Dataflow

The select and switch actors as defined in BDF allow the selective execution of blocks depending on some control sequence. SDF does not provide this capability because it assumes that each node in a graph will execute at each iteration of the graph which means that in a scenario where *select* and *switch* capabilities are needed, dummy tokens must be produced in order to accommodate this functionality.

In Cyclo-Static Dataflow (CSDF) [60], applications are able to express conditional execution of actors by being able to define *phases* of actor execution; basically an actor can have different code modules (phases) to be executed. While actors can have multiple phases, the application is assumed ultimately to exhibit a periodic behavior. Figure 3.9 represents a CSDF with three phases, as indicated by the digit sequences, *e.g.*, “0,1,1,” where a *0*

indicates the absence of tokens and a *1* indicates the existence of tokens. During the first phase, Actors 1, 3, and 4 fire while in phases 2 and 3 Actors 1, 2, and 4 will fire. As we can see, the behavior of the graph is periodic but there are different underlying phases. This MoC provides *switch* and *select* capability in the same manner as BDF without the use of control ports.

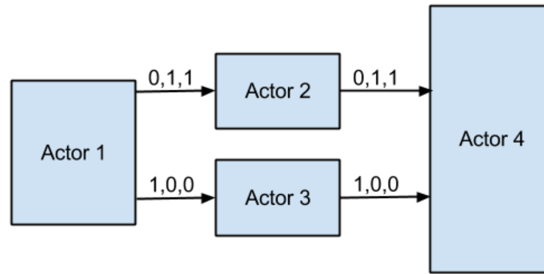

Figure 3.9: CSDF Graph Example.

The benefit of this MoC, is that graph consistency and its execution with bounded buffers can be analyzed statically in the same manner as SDF since the periodic behavior of the graph is specified statically. In BDF the *proportion* of TRUE statements is observed while in CSDF the exact proportion of TRUE statements are defined.

Cyclo-Dynamic Dataflow (CDDF) [61] is an extension of CSDF where the number of actor phases and amount of token consumption/production are dynamic. By introducing dynamic runtime behavior it can be impossible to prove graph consistency, similar to the situation with BDF and IDF.

3.1.5 Parametrized Synchronous Dataflow

Parametrized Synchronous Dataflow (PSDF) [62] allows the reconfiguration of graphs actors during flowgraph runtime. The authors of [62] discuss parameters having associated *domains*, which are the set of all possible configurations for a particular parameter. To guarantee bounded memory execution, every actor has an associated *maximum token transfer function* providing an upper bound on the maximum number of tokens that actors can produce. Each arc also has an associated *maximum delay value* which determines the maximum number of delay tokens that can exist on an arc.

The notion of reconfiguration in PSDF graph uses *quiescent points* [63], where a graph can reconfigure upsampling and downsampling factors, essentially modifying the relative rates of nodes at particular points of graph execution. A quasi-static schedule, where token rates are described symbolically, can be evaluated once the graph reconfiguration parameters are defined during execution.

3.2 Process Algebra

Process algebra can be defined as "the study of the behavior of parallel or distributed systems by algebraic means" [64]. Process behavior, composition, and input/output interfaces can be described by using the algebra, where an outcome of relying on algebraic expressions in describing behavior is that systems lend themselves to verification [64]. From this perspective, process algebra provides semantics suitable for describing reactive systems and allowing their verification.

3.2.1 Communicating Sequential Processes

CSP [65] [66] is a process algebra which treats input/output operations as the basic primitives of computation. The detailed semantics of the language can be found in [66]; the following highlights a small set of the language which is relevant to the work presented. The CSP language is very rich and the following is not intended to be a complete summary of the language.

Basic Semantics

In describing a process, a set of associated events need to be identified, known as the process *alphabet*; the expression, αP, refers to the alphabet α of process P. For example, the alphabet of a vending machine, V, might be {*dollar, choose_drink1, choose_drink2, dispense_drink1, dispense_drink2*}. A user is able to interact with the machine by inserting *dollar*, choosing either *drink1* or *drink2* by pressing on the associated buttons. By pressing the buttons, the user generates one of two events in the vending machine *choose_drink1* or *choose_drink2*. In return, the machine dispenses the requested drink by generating either the *dispense_drink1* or *dispense_drink2* events. Therefore the alphabet of the vending machine process, V, is as follows:

$$\alpha V = \{\mathit{dollar}, \mathit{choose_drink1}, \mathit{choose_drink2}, \mathit{dispense_drink1}, \mathit{dispense_drink2}\}$$

The behavior of the process V is restricted to the indicated alphabet and it is not able to understand new events or generate new events.

The prefix operator, $\rightarrow$, can be used to combine an event, x, and a process P, to generate a new process, as an example:

$$x \rightarrow P \qquad \text{(x then P)}$$

In the previous vending machine example, if a customer, C, keeps feeding a *dollar* into the vending machine and requesting *drink1* without picking the drink up and keeps doing so infinitely the prefix operator can describe this behavior recursively:

$$C = \mu X : \{dollar, choose_drink1, dispense_drink1\} \bullet$$
$$(dollar \rightarrow (choose_drink1 \rightarrow (dispense_drink1 \rightarrow C)))$$

For convenience, the alphabet from the process description will be dropped as described in [66]. The alternative customer definition is:

$$C = dollar \rightarrow (choose_drink1 \rightarrow (dispense_drink1 \rightarrow C))$$

Choice

The vending machine, V, allows the customer, C, to select the drink they desire so the process description for the vending machine needs to account for a customer's choice. The choice operator, $|$, in the context of $(x \rightarrow P|y \rightarrow Q)$ allows a choice to be made between processes P and Q depending on which event occurs, either x or y. The vending machine description is as follows:

$$V = dollar \rightarrow ((choose_drink1 \rightarrow dispense_drink1 \rightarrow V)|$$
$$(choose_drink2 \rightarrow dispense_drink2 \rightarrow V))$$

Concurrency

For now, we have defined a vending machine process, V, and a customer process, C, and provided equations describing each of the processes. However, in a real life scenario a customer interacts with a vending machine to yield the drink they desire, so we must be able to describe this scenario as a composition of both processes. We can do so by composing parallel processes using the following notation.

$$P \| Q \qquad \text{(P in parallel with Q)}$$

The concurrent composition of processes can occur in synchrony, meaning that both processes must contain process definitions allowing both processes to interact. Some relevant laws governing the concurrency operator are:

$$(c \rightarrow P)\|(c \rightarrow Q) = (c \rightarrow (P\|Q))$$

$$(c \rightarrow P)\|(d \rightarrow Q) = STOP \qquad\qquad if c \neq d$$

Where $STOP$ corresponds to the execution stopping, as will be discussed some more later. The concurrency operator can be used to provide *choice* or *interleave* capabilities to the concurrent composition of processes. An example of how the concurrency operator can be applied to the vending machine, V, and customer C in the *choice* context:

$$X = (C\|V)$$

$$\begin{aligned} X = (&dollar \rightarrow choose_drink1 \rightarrow dispense_drink1 \rightarrow X)\| \\ &(dollar \rightarrow ((choose_drink1 \rightarrow dispense_drink1 \rightarrow X)| \\ &(choose_drink2 \rightarrow dispense_drink2 \rightarrow X))) \end{aligned}$$

$$\begin{aligned} X = (&dollar \rightarrow choose_drink1 \rightarrow dispense_drink1 \rightarrow X)\| \\ &(dollar \rightarrow (choose_drink1 \rightarrow dispense_drink1 \rightarrow X)| \\ &dollar \rightarrow (choose_drink2 \rightarrow dispense_drink2 \rightarrow X)) \end{aligned}$$

$$\begin{aligned} X = (&dollar \rightarrow choose_drink1 \rightarrow dispense_drink1 \rightarrow X)\| \\ &(dollar \rightarrow choose_drink1 \rightarrow dispense_drink1 \rightarrow X) \end{aligned}$$

$$X = (dollar \rightarrow choose_drink1 \rightarrow dispense_drink1 \rightarrow X)$$

In the previous example, we can see that after composing two parallel processes C and V we can describe the resulting behavior as a single process X. The single process behavior description is possible because the process interaction can only yield a single behavior as derived in the previous example. This result demonstrates how the processes C resulted in a choice to be made when composed in parallel with the process V.

The concurrency operator can operate with two processes which do not have the same alphabet, where an *interleaving* of events can occur when events not shared by both processes occur. For example, let us assume a second vending machine, *V2*, which makes a *beeping* sound after a drink is selected. The associated process definition is as follows:

$$\begin{aligned} V2 = dollar \rightarrow (&(choose_drink1 \rightarrow beep \rightarrow dispense_drink1 \rightarrow V)| \\ &(choose_drink2 \rightarrow beep \rightarrow dispense_drink2 \rightarrow V)) \end{aligned}$$

Let us assume there is a customer, *C2*, who continuously chooses a drink from the vending machine, but never picks it up. However, after choosing the drink the customer would *cheer* out of anticipation for the drink. The *C2* process definition is as follows:

$$C2 = (\mathit{dollar} \rightarrow \mathit{choose_drink1} \rightarrow \mathit{cheer} \rightarrow C2)$$

The concurrent composition of *V2* and *C2* is as follows:

$$X = (C2 \| V2)$$

$$X = (\mathit{dollar} \rightarrow \mathit{choose_drink1} \rightarrow \mathit{beep} \rightarrow (\mathit{dispense_drink1} \rightarrow \mathit{cheer} \rightarrow X \\ | \mathit{cheer} \rightarrow \mathit{dispense_drink1} \rightarrow X))$$

Deadlock

A process with alphabet A which never interacts with any of the events in the alphabet A is defined as $STOP_A$. Going back to our vending machine example, assuming there is a customer *C3* who insists on getting an unavailable drink from the vending machine, *drink3* and wants to do so an infinite number of times without picking up the drink. The customer process definition is as follows:

$C3 = (\mathit{dollar} \rightarrow \mathit{choose_drink3} \rightarrow \mathit{dispense_drink3} \rightarrow C3)$

When customer *C3* interacts with vending machine, *V*, the following occurs:

$(C3 \| V) = (\mathit{dollar} \rightarrow \mathit{STOP})$

The concurrent combination of both processes terminates in the *STOP* process, indicating that a *deadlock* has occurred. The customer is trying to find the *choose_drink3* button which does not exist in the vending machine and the machine is waiting for either the *choose_drink1* or the *choose_drink2* events to occur. At this point there isn't any further viable interaction between both process.

Communication

Processes should be able to communicate with each other over *channels*, where

$\alpha c(P) = \{v | c.v \in \alpha P\}$

represents the set of all messages, v, which process, P, can communicate over channel c. A process, P accepts input v over channel c and then goes back to behaving like P, is described as

$(c?v \rightarrow P) = P$

While a process, P, outputting a value over channel c then behaving like P is described as

$(c!v \rightarrow P) = P$

Communication channels in CSP are synchronous, meaning if a process is ready to receive data over a channel it will block until the associated writing process is ready to write over the channel. If a writing process is ready to write over a channel it will block until the receiving process is ready to receive.

3.2.2 Alternative Process Algebra

Calculus of Communicating Systems (CCS) [67] is an alternative process algebra worth noting. Many of the semantics discussed in this CSP section can be applied using CCS semantics, however a closer inspection of algebra semantics between both algebras would reveal differences not discussed here since they are not being used in the work presented.

Π-calculus [68] is an algebra providing process mobility facilitating dynamic reconfiguration of channel connections between processes. In terms of a CE, Π-calculus is able to express runtime reconfiguration of radio flowgraphs since the semantics of the algebra supports it. Π-calculus is not used in this work because while it expresses process mobility the underlying dataflow MoCs surveyed do not supports such reconfigurations.

3.3 Summary

The SDF dataflow MoC is used to describe real-time, radio flowgraphs. While SDF does not allow runtime reconfiguration, it allows the detection of graph inconsistency, static buffer allocation, and static graph analysis. In reconfiguring SDF radios, graph resources need to be de-allocated followed by an analysis of the new graph description for consistency and static buffer sizes. The new graph resources then need to be allocated. Alternative dataflow MoCs are discussed to provide insight on how control can be introduced in dataflow graphs even though such mechanisms are not explored in this dissertation. The CSP MoC is used to describe reactive radio models because it allows the definition of parallel and concurrent process behavior in addition to describing process IO interfaces. While verification of CSP-based systems is not covered in this dissertation, its use allows verification to be explored in future work. The details of how the MoCs are used in describing radio flowgraphs follows in Chapter 6.

Chapter 4

Digital Communication

This chapter is a brief introduction to digital communication for readers who are unfamiliar with the field. It is not intended to be an inclusive summary, but, rather as a summary of those aspects which bear on the work included in this dissertation.

4.1 Protocol Layering

Before discussing the design of a Differential Binary Phase-Shift Keying (DBPSK) radio, it is helpful to discuss the radio design in the context of a protocol architectures.

4.1.1 OSI Model

The International Organization for Standardization (ISO) defined a reference model for protocol definition named the Open System Interconnection (OSI) model which encompasses seven layers [69]:

1. **Physical**: Responsible for taking a stream of data bits and physically transmitting them over a specified medium.
2. **Data Link**: Responsible for the reliable transfer of data between two nodes, point-to-point data transfers, including any necessary synchronization and control. In addition, it breaks down the data into packets adding information such as source and destination addressing.
3. **Network**: Responsible for starting, terminating, and establishing connections between nodes.

4. **Transport**: Provides the flow control and reliable transfer of data between end nodes.
5. **Session**: Manages and terminates connections between applications running on communicating nodes in terms of *sessions*.
6. **Presentation**: Deals with differences in data representation, *e.g.*, the order of most and least significant bits, the data *endianness*.
7. **Application**: Allows user applications to send/receive data to and from the OSI framework.

The OSI model provides a standard framework for designing and analyzing communication protocols. This structured approach makes it possible to address separate aspects of communication at various layers in the framework model independently. Application data is broken down into smaller chunks, and each layer adds header and footer control information to provide context for addressing and error handling.

While protocol layering is not directly addressed in this dissertation, blocks which encode and decode data into packets are mentioned in the implementation chapter. The introduction of protocol layering here provides insight into how data is passed to the physical layer.

4.2 Binary Phase Shift Keying

Binary Phase-Shift Keying (BPSK) modulation depends on shifting a carrier signal by 180° to indicate a symbol 0 and passing the carrier unmodified to indicate a symbol 1, as shown in Equation 4.1 [70].

$$s_i(t) = \begin{cases} \sqrt{\frac{2E_b}{T_b}} cos(2\pi f_c t) & \text{symbol 1} \\ \sqrt{\frac{2E_b}{T_b}} cos(2\pi f_c t + \pi) = -\sqrt{\frac{2E_b}{T_b}} cos(2\pi f_c t) & \text{symbol 0} \end{cases} \tag{4.1}$$

$s_i(t)$ = Voltage representing one symbol
E_b = Energy per Bit
T_b = Bit Duration

A BPSK transmitter essentially toggles between two different phases of the carrier to represent the symbols 0 and 1. BPSK depends on the use of an Nonreturn-to-Zero (NRZ) encoder, meaning that a symbol 1 corresponds to a positive amplitude signal and a symbol 0 corresponds to a negative amplitude value equal in magnitude. This can be seen in Equation 4.1 with the 1 and 0 values corresponding to equal magnitude signals but with different signs.

To demodulate the signal, a receiver detects the phase changes of the carrier to figure out if the received bits are 1's or 0's. Since in a BPSK radio the information is demodulated

Table 4.1: Differential Coding

bit	ϕ_{t-1}	ϕ_t
0	0	π
0	π	0
1	0	0
1	0	0
0	0	π

relative to the carrier signal, the receiver phase must be locked to the transmitter signal requiring the use of a Phase-Locked Loop (PLL). Therefore a BPSK receiver is a *coherent* receiver.

4.3 Differential Binary Phase Shift Keying

DBPSK modulation is similar to BPSK but it allows the use of *non-coherent* signal detection in the receiver design; the receiver's phase does not need to be locked to the transmitter's carrier phase. A DBPSK signal is constructed using differentially encoded signals, where a logic level transition, from logic levels 0 to 1 or 1 to 0, corresponds to symbol 0 and the lack of a transition, the signal remaining either on logic level 0 or 1 corresponds to symbol 1. Table 4.1 summarizes the carrier phase offset, ϕ_t, at time t for the symbol sequence 00110.

The use of differential encoding in DBPSK, allows non-coherent detection signals since it depends on tracking the change in the carrier signal's phase instead of tracking its absolute phase.

4.4 Pulse Shaping

Pulse shaping is needed to conserve the transmission bandwidth and to reduce the effect of symbols bleeding into each other in the receiver causing Intersymbol Interference (ISI), a symbol received while the previous symbol is still present in the time domain [70]. The pulse shaping filter tailors the time and frequency domain response of the waveform to the finite bandwidth of the channel in a way that mitigates the ISI.

A raised-cosine filter mitigates ISI and is specified by the definition of a *flat*-portion and *roll-off*-portion of the pulse [70]. In Figure 4.1, the black plot has zero roll-off and corresponds to an *optimum* pulse, requiring the least amount of bandwidth as can be seen from the figure where $P(f)$ represents the normalized power spectral density and f represents the normalized

frequency. However, the perfect rectangular filter response is unrealizable. Therefore, by allowing a *rolled-off* portion in the pulse, it is possible to implement such filters making them more practical at the expense of increasing the signal bandwidth. Looking at Figure 4.1, the red plot corresponds to pulse with an only a roll-off portion, a sinusoid, and the blue plot is a pulse with some flat and some rolled-off portion. As we can see that increasing the roll-off portion increases the required bandwidth for transmitting the pulse. The equation for the spectral response of a raised-cosine filter is given by 4.2.

$$P(f) = \begin{cases} \frac{\sqrt{E}}{B_0} & 0 \le |f| < f_1 \\ \frac{\sqrt{E}}{4B_0}[1 + cos(\frac{\pi*(|f|-f_1)}{2(B_0-f_1)})] & f_1 \le |f| < 2B_0 - f_1 \\ 0 & 2B_0 - f_1 \le |f| \end{cases} \tag{4.2}$$

P = pulse power spectral density in watts/Hz
E = transmitted signal energy per bit
B_0 = signal bandwidth
f_1 = frequency cut-off for *flat* portion
f = frequency

A Root-Raised-Cosine Filter (RRC) filter allows the pulse shaping filter implementation to be split between the transmitter and receiver. The transmitter implements the square root of a raised cosine filter and a matched filter in the receiver implements a square root of a raised cosine filter, and the combination of both filters operate as a single pulse filter. Partitioning the pulse shaping filter in this manner helps mitigate the effect of noise in the receiver as discussed in more detail in [70].

4.5 Summary

This chapter discusses some fundamentals of digital communication used in this dissertation. In Chapter 7, DBPSK transmitter, receiver, and link examples are discussed to demonstrate how the framework in Chapter 6 can be used to construct radio applications. In constructing DBPSK radios, RRCs filters are used for pulse shaping and data streams are broken down into chunks so they can be transmitted and received as packets.

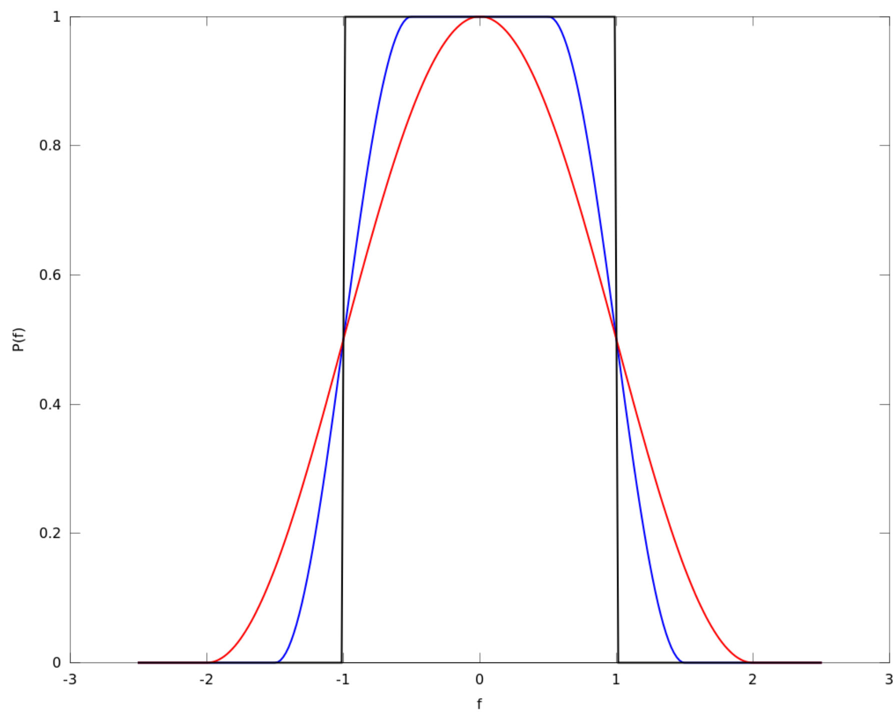

Figure 4.1: Raised-Cosine Pulse Shaping Normalized Power Spectral Density.

Chapter 5

Tools

To accomplish the work presented in this dissertation, three main software tools are used. GNU Radio [6] for implementing SDR radio flowgraphs, the Occam language [71] for developing reactive radio flowgraphs, and Ptolemy for simulating SDF based radio flowgraphs. In terms of hardware, a first generation USRP [15] is the programmable radio front-end used for over-the-air communication.

5.1 Software

5.1.1 GNU Radio

GNU Radio is an open source SDR tool providing the signal processing blocks necessary to build and analyze analog and digital radios. It uses the interpreted language Python to invoke signal processing blocks, configure them, define the input/output connections, and to start and stop flowgraph execution. Once a radio graph starts executing, the physical block implementation is done in C++ with the Simplified Wrapper and Interface Generator (SWIG) acting as a wrapper interface for the C++/Python as shown in Figure 5.1, which summarizes the GNU Radio architecture.

5.1.2 Occam

Occam is a language with semantics largely based on CSP [71] and the KRoC compiler [72] is used for compiling the reactive radio flowgraph definitions. Π-Occam is an extension of Occam, which incorporates process mobility aspects of Π-calculus into the language. There are CSP-based libraries for many languages, *e.g.*, the CCSP [73] library for the C language, C++CSP [74] for C++, and JCSP [75] for Java. However, Occam is used to avoid the radio

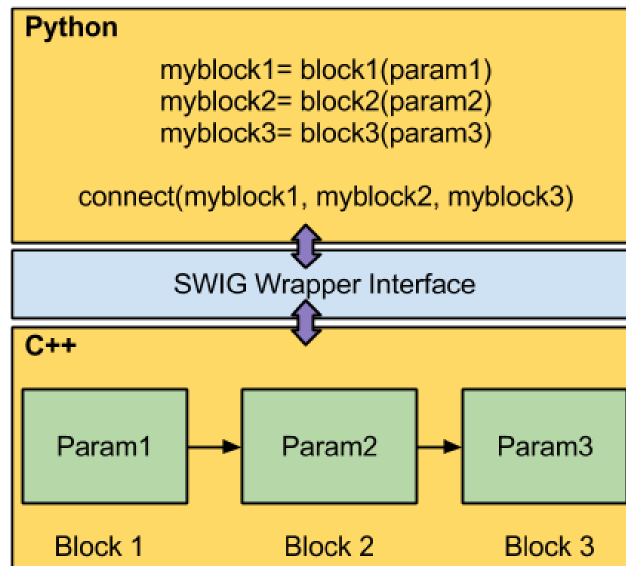

Figure 5.1: GNU Radio Framework.

definitions incorporating aspects of the C, C++, or Java languages not compatible with CSP semantics. Occam lends itself to describing concurrent processes and has been used to target a processor for parallel computing application, the *transputer* [76]. Concurrent processes in the transputer are mapped to various transputer processors, and the communication links between the processes can be translated to the physical communication channels connecting the various transputers.

5.1.3 Ptolemy

Overview

Contemporary systems can be composed of sub-modules exhibiting different computing behavior, *e.g.*, the cruise control software in a car has different characteristics than the GPS navigation software. Ptolemy provides an integrated simulation environment allowing the simulation of systems whose internal structure are based on different MoCs. For example a car can be simulated in Ptolemy, where the different sub-components of the car are represented by different MoCs and the simulation software handles the interaction between the various MoCs used [77]. By allowing such system simulations, designers are able to have a more holistic view of their systems and can analyze and anticipate *emergent behavior*, unintended system behavior, which can appear in the final system [77].

Ptolemy allows developers to instantiate *directors*, which define the simulation execution

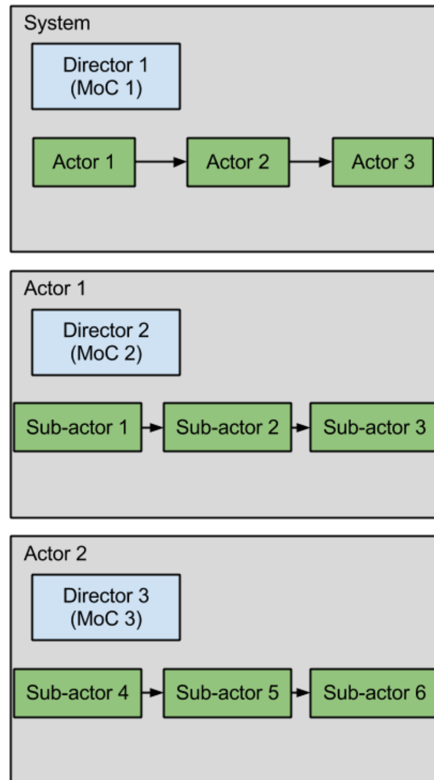

Figure 5.2: Ptolemy Framework.

semantics for a specified MoC. Functional blocks, or actors, execute system functionality, and actors can be *atomic*, specifying a singular functional unit or *composite*, having a combination of sub-blocks specifying behavior. Composite actors can have directors specifying different MoCs, and Ptolemy is able to handle the interaction between the different actors and different MoCs. Figure 5.2 provides an example of system specification using Ptolemy, where Director 1 handles the execution of the overall system, Director 2 handles the execution of Actor 1 and its sub-actors, and Director 3 handles the execution of Actor 2 and its sub-actors. In the scope of the work presented, Ptolemy is used to simulate the SDF based radio flowgraph execution.

5.2 Hardware

An SDR platform is composed of the following main components, as can be seen in Figure 5.3:

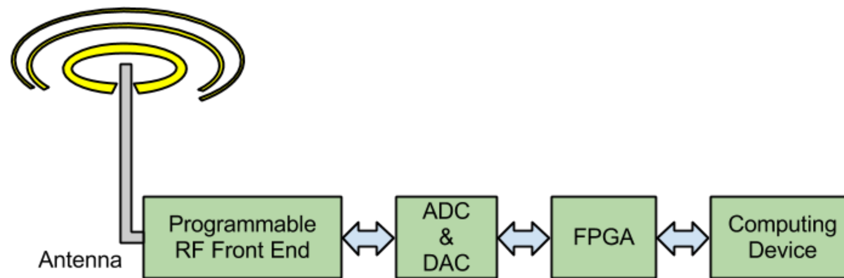

Figure 5.3: Software Defined Radio Platform.

- **Programmable RF Front-End**: The user can specify the transmitter and/or receiver center frequency. The front-end can also allow users to specify a particular IF frequency amongst other parameters. RF interfaces can support multiple receive and/or transmit signals concurrently allowing for various RF configurations such as Single-Input and Single-Output (SISO), Single-Input and Multiple-Output (SIMO), Multiple-Input and Single-Output (MISO), and Multiple-Input and Multiple-Output (MIMO). More information regarding such antenna configurations can be found in [78]. The antenna used with the front-end needs to be suited for the frequency band of interest.

- **ADC and DAC**: The Analog-to-Digital Converter (ADC) digitizes the received analog signal and the Digital-to-Analog Converter (DAC) converts the digital transmit signal to an analog one which can be transmitted using the RF interface.

- **FPGA**: An FPGA is typically used for decimation and down conversion for receivers and interpolation and up conversion for transmitters. The FPGA can also be used for implementing SDR applications.

- **Computing Device**: The computing device typically implements either some or all of the SDR applications. It can be an embedded processor connected to the FPGA over a bus interface or it can be host computer connected over a networking interface.

5.2.1 Universal Software Radio Peripheral

A first generation USRP [15] is the SDR platform used in the work presented in this dissertation to provide over-the-air communication, and is connected to a host computer using a USB 2.0 interface. It contains two dual 64 MS/s 12-bit ADCs, two dual 128 MS/s 14-bit DACs and an Altera Cyclone FPGA. Users can select from a number of RF daughterboards with different frequency coverages from the platform manufacturer, Ettus Research [79]. For the work presented, the WBX [80] board is used providing RF coverage in the 50-2200 MHz

range for both transmit and receive operation. The USRP allows users to connect one or two daughterboards to the platform. Each daughterboard provides a single dedicated receiver interface and a second interface which can be configured as a transmitter or secondary receiver.

5.3 Summary

In this dissertation, GNU Radio is used to implement over-the-air radio flowgraphs, to provide buffer scaling control between signal processing components in the flowgraph, and to monitor computational performance of the radio implementations. A first generation USRP is used as the programmable radio front-end for over-the-air testing. Ptolemy is used to simulate digital radio implementation to ensure that they behave as expected in the SDF domain. Finally, Occam is the language used to define CE reactive radio models.

Chapter 6

Framework

6.1 Overview

The goal of the proposed framework is to provide CEs with the capability to map a reactive model of a radio flowgraph to an implementation-specific dataflow model suitable for SDR. By applying suitable MoCs to both domains, a CE is able to gain access to a set of computational knobs, allowing the CE to modify the performance of the radio. By accessing a set of computational meters, a CE can observe the effects of the radio reconfigurations.

The work presented in this section provides a framework allowing CEs to define CSP based radio flowgraph models and have the framework interpret them to SDF based SDR implementations. By applying the SDF MoC it is possible for CEs to configure the size of the FIFO buffers streaming data between blocks through modifying the sizes of buffer scaling factors. By reconfiguring the buffer sizes, the throughput, latency, reconfiguration time, and total memory consumption of a radio flowgraph are controlled and can be observed by a CE while trying to achieve its computational performance goals.

A brief summary of how a CE interacts with the framework is as follows:

- A CE starts by specifying the radio flowgraph it wants implemented. It can specify its desired *meter* readings for total memory usage, latency, throughput, and reconfiguration times and have the framework attempt to achieve them.
- The selected radio flowgraph is then interpreted and associated Ptolemy and GNU Radio flowgraphs are generated.
- The framework then proceeds to run the GNU Radio model to run the physical implementation.
- So far as the *computational knobs* available to the CE are concerned, the framework

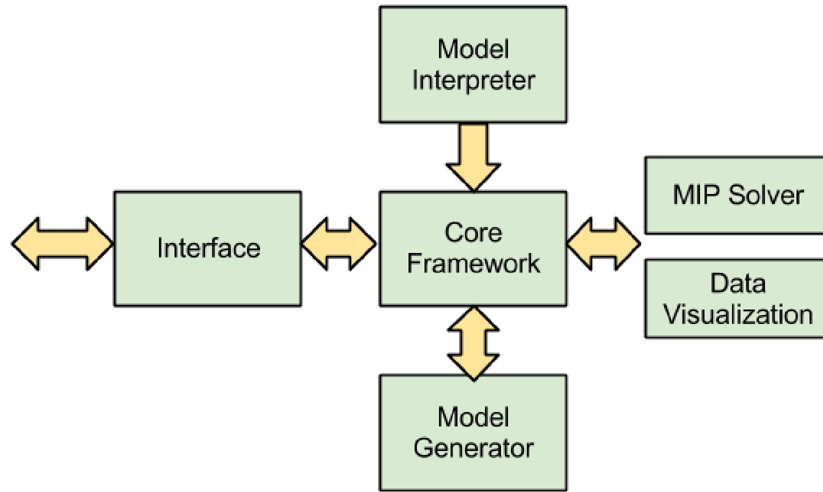

Figure 6.1: Framework Software Architecture.

can modify the buffer scaling for the SDF radio graphs. By iterating over various buffer scalings, it is able to explore implementations with various throughput, latency, and total memory consumption values as will be discussed in a later chapter.

The framework is composed of the following main components, as can be seen in Figure 6.1:

- **Model Interpreter**: Performs the physical interpretation of the radio models.
- **Model Generator**: Generates GNU Radio and Ptolemy based flowgraphs from the information gathered by the model interpreter.
- **Core Framework**: Generates the topology matrix for the SDF radio graph, checks for data rate consistency, and sets up a Mixed Integer Program (MIP) to find the repetition vector for the radio flowgraph. The core framework also runs the physical SDR implementation in GNU Radio and collects performance measurement information.
- **Interface**: Provides the interface between CEs and the framework. A CE passes radio flowgraph definitions through the interface, specifies a desired range for the throughput, latency, reconfiguration time, and memory consumption for a flowgraph. A CE is also able to reconfigure the overall SDR implementation buffer sizing by specifying buffer scaling factors.
- **MIP Solver**: Provides an interface to a MIP package used in calculating the repetition vector of SDR flowgraph implementation.
- **Data Visualization**: Provides an interface to a data visualization package.

6.2 Model Interpreter

6.2.1 Language

The model interpreter reads in radio flowgraph description files passed by CEs which are reactive models described in CSP, in this case written using the Occam language.

6.2.2 Model Definitions

Three radio flowgraphs are implemented:

1. **DBPSK Transmitter** - Which will be referred to as process *TX*.
2. **DBPSK Receiver** - Which will be referred to as process *RX*.
3. **DBPSK Link over an Additive White Gaussian Noise (AWGN) Channel** - Which will be referred to as process *TXRX*.

A CE initially can choose between the above three radio flowgraphs:

$$CE = (select_TX \rightarrow TX)|(select_RX \rightarrow RX)|(select_TXRX \rightarrow TXRX)$$

Each radio flowgraph at this stage is an Occam program that instantiates the specific signal processing functionality necessary to implement the radio. For example, the DBPSK transmitter is defined from the following processes:

- **Parameter Generator (PG)**: This process is responsible for defining the configuration parameters, or *meta-data*, for the DBPSK transmitter.
- **Data Source (DS)**: Instantiates the data source for the radio.
- **Packet Encoder (PE)**: Creates packets by breaking up the input data stream into fixed length arrays and encapsulating them with packet headers and footers containing information such as packet length and checksum values.
- **DBPSK Modulator (DM)**: Instantiates a DBPSK modulator.
- **Channel Filter (CF)**: Defines an RRC filter.
- **Baseband Scale (BS)**: Scales the baseband signal before sending it over the RF interface.
- **RF Out (RO)**: Instantiates the output RF interface.

The CSP semantic for defining the transmitter chain is as follows:

$$TX = PG \| DS \| DE \| DM \| CF \| BS \| RO$$

Concurrent combination of the sub-processes which define the radio operation define the behavior where sub-process events are intended to be interleaved. The parameter generator defines the model *meta-data*, which are implementation specific parameters that are not necessarily meaningful in the reactive realm but provide the context necessarily in translating the reactive CE radio model to its SDF SDR implementation. The following are some of the specified parameters:

- **First stage sampling rate (sr1)**: The data processing sampling rate.
- **Second stage sampling rate (sr2)**: Sampling rate of data going to and from the RF interface.
- **Baseband gain (bbgain)**: Specifies the scaling factor for the baseband signal before it is sent or received over the radio interface.
- **Center frequency (cf)**: Specifies the radio front-end's RF transmit/receive frequency.
- **RRC filter roll-off factor (roff)**: Used to define the channel filter roll-off factor. This parameter is used to define the channel filter for the transmitter and the matched filter in the receiver.
- **AWGN gain (ag)**: Defines the gain of the Gaussian noise source.

Input and output channels are instantiated between the blocks to indicate the block connections. In the the Occam program, the channel connections are used to transfer the *meta-data* between the processes; this essentially allows the radio parameters to disseminate the parameters to the various processes.

The *parameter generator* block is a source in the Occam program. It is only composed of the output channel as defined here:

$$PG = \mathit{right!sr1} \rightarrow \mathit{right!sr2} \rightarrow \mathit{right!bbgain} \rightarrow \mathit{right!cf} \rightarrow \mathit{right!roff} \rightarrow \mathit{right!ag} \rightarrow PG$$

The *radio out* block is a sink, so it is only composed of input channels with the process definition being:

$$RO = \mathit{left?sr1} \rightarrow \mathit{left?sr2} \rightarrow \mathit{left?bbgain} \rightarrow \mathit{left?cf} \rightarrow \mathit{left?roff} \rightarrow \mathit{left?ag} \rightarrow RO$$

The remaining blocks input the configuration parameters and then output them to the remaining blocks:

$$BLOCK = left?sr1 \rightarrow right!sr1 \ldots \rightarrow left?ag \rightarrow right!ag \rightarrow BLOCK$$

Before the radio flowgraphs are passed to the model interpreter, they are compiled using the Occam compiler [72] to make sure that they are semantically correct and to ensure that there are no deadlocks in the radio flowgraphs. Deadlocks indicate that some processes are either not reading all of the necessary configuration parameters or are not forwarding all the parameters correctly to the next block.

6.2.3 Interpretation

The interpreter parses the Occam radio flowgraph model and constructs lists summarizing the instantiated signal processing blocks, meta-data, and communication channels. The reason for having two sampling rates for the radio is that the RRC filter in the transmit graph interpolates the sampling rate from the data processing rate to the RF interface sampling rate. Similarly the RRC filter in the receive graph decimates incoming data from the RF front-end to a suitable sampling rate for data processing. Alternative radio flowgraph strategies can be used.

6.3 Model Generator

After the Occam radio flowgraph definitions are interpreted, the framework knows what processes are instantiated, the defined meta-data parameters, and the needed IO channel definitions. At this stage, the framework is able to generate suitable SDF-based SDR flowgraphs for Ptolemy and GNU Radio. For each target environment, there are associated *domain-mapping* tables. For example, a DBPSK modulator block has an associated block definition for GNU Radio and Ptolemy. The domain mapping tables include the exact block name, IO port names, sampling rates, and meta-data mapping for each for each of the target environments. Meta-data such as signal gain, RRC filter roll-off factor, RF transmit/receive, and center frequency are all passed to the designated block definitions. Figure 6.2 provides a summary of the model generation process.

GNU Radio has the basic signal processing block definitions to implement the DBPSK link, but Ptolemy does not. Therefore, to provide meaningful Ptolemy simulation files, the needed blocks are implemented in Ptolemy, and they include differential message encoders/decoder blocks, integer to binary data representation, and binary to integer representation. However, the creation of message packets is not implemented in Ptolemy since the scope of the work

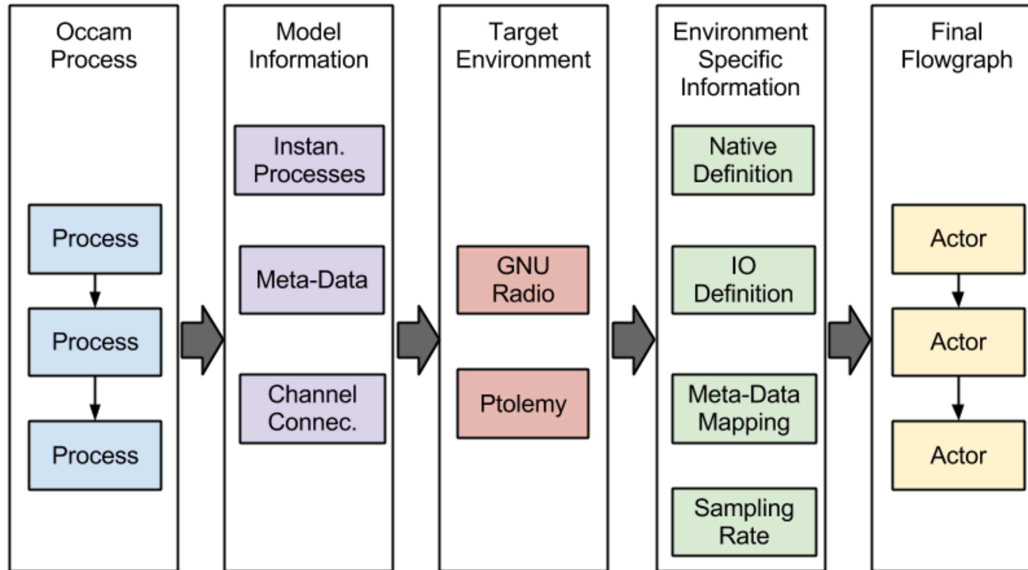

Figure 6.2: Model Generation Domain Mapping Process.

is not to create detailed DBPSK simulations in Ptolemy, but to provide a framework for CE and SDR mapping and to define associated computational knobs and meters.

Ultimately, the model generator takes the interpreted radio flowgraphs and generates XML formatted files for Ptolemy and GNU Radio which can be opened in each environment.

6.4 Core Framework

The target SDR flowgraphs are implemented using the SDF MoC. In order to provide proof-of-concept applications and collect performance measurements, the framework is able to generate and run GNU Radio based flowgraphs. A first generation USRP [15] is the radio interface used for over-the-air flowgraphs.

There are two different stages for implementing the radio flowgraphs. The first stage is based on the pure process definitions from the Occam files, where the instantiated processes, I/O connections, and meta-data are defined. During the second stage, the flowgraph definition interpreted previously is passed to GNU Radio and *expanded*. This means that if some blocks are hierarchical, composed of multiple sub-blocks, GNU Radio will replace the hierarchical blocks with the sub-blocks contained in it, hence *expanding* the flowgraph definition.

The core framework provides the following analysis and calculations:

- Topology matrix construction.

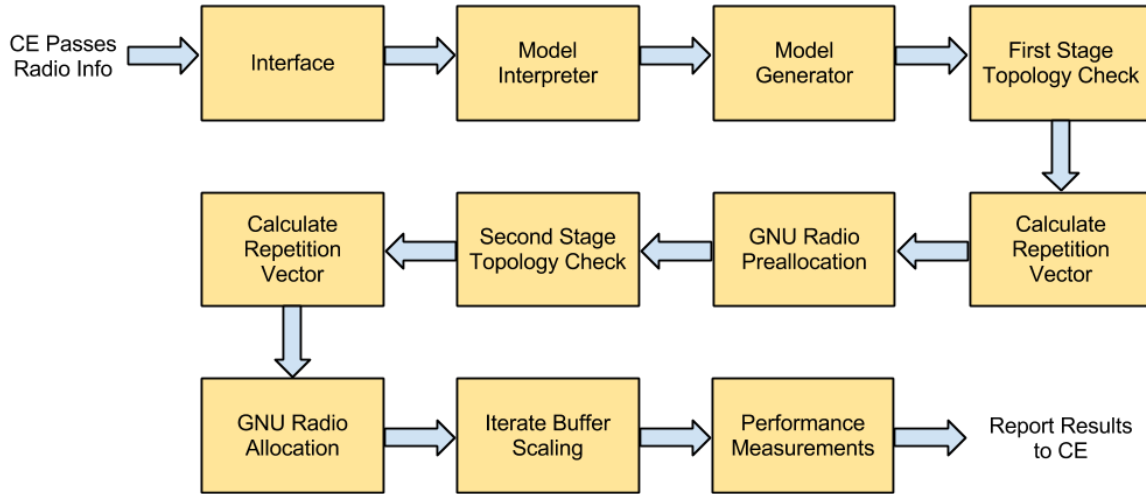

Figure 6.3: Sequence of Events for CE/Framework Interaction

- Sampling rate consistency check.
- Repetition vector calculation.
- Performance measurement collection.

Figure 6.3, provides a summary of an example interaction between a CE and the framework which are discussed in detail throughout this chapter.

6.4.1 First Stage

During this stage, a topology matrix is constructed using information in the domain-mapping tables created by the model generator. The meta-data from the Occam file provides the exact sampling rates for each of the associated blocks. However, the topology matrix is constructed using the relative sampling rates and not absolute sampling rates, *e.g.*, if the sampling rate of block 1 is 16,000 samples/second and the sampling rate of block 2 is 32,000 samples/second then the relative rate of block 1 is $\frac{16000}{16000} = 1$ and the relative rate for block 2 is $\frac{32000}{16000} = 2$. Basically each sampling rate is divided by the greatest-common-divisor of the flowgraph's sampling rates.

After the first stage topology matrix is constructed, a sampling rate consistency check is performed by calculating the rank of the topology matrix as presented previously in Equation 3.2. The repetition vector is then calculated by constructing and solving a MIP as will be discussed in more detail in a later section. Finally, the total memory needed to implement

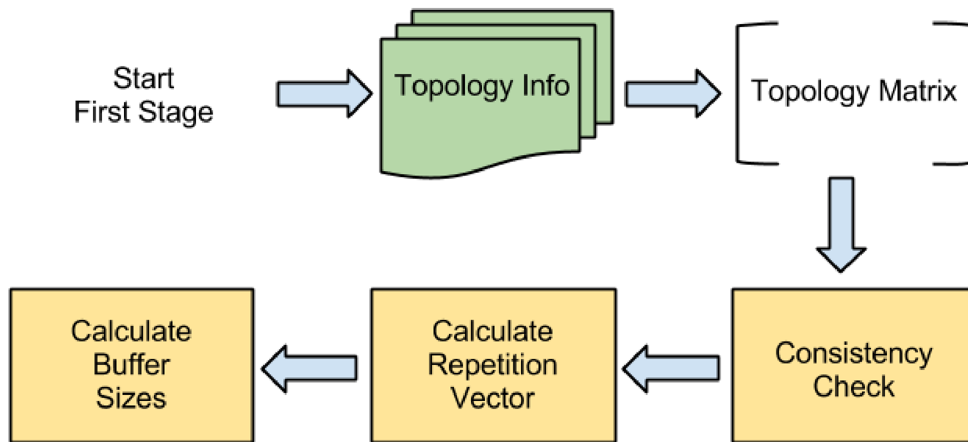

Figure 6.4: First Stage Analysis

the buffers is calculated by the framework. At this point only the memory utilization can be calculated since throughput, latency, and reconfiguration times are measured from the actual implementation. Figure 6.4 summarizes the progression of events during the first stage of the core framework. At this stage a CE will know if the flowgraph definition is consistent and the memory utilization for the current radio definition.

6.4.2 Second Stage

During the second stage, the framework queries GNU Radio for the *expanded* radio flowgraph. If any of the signal processing blocks defined during the first stage are hierarchical, the framework needs to redefine the topology matrix to account for the extra blocks and associated buffers. The repetition vector also needs to account for the extra blocks. A summary of the steps performed during the second stage are shown in Figure 6.5, which is followed by a more detailed explanation.

Assume that a CE selects a DBPSK link running over a AWGN channel. Figure 6.6 illustrates the framework's perspective of the flowgraph at the end of the first stage. The upward arrow and number 2 on top of the first RRC filter indicates an interpolation factor of 2 and the downward arrow and the number 2 in the second RRC filter indicate a decimation factor of 2. We can see that all blocks prior to the first RRC filter operate at a set sampling rate, assumed to be 1. Blocks between the first and second RRC filter operate at a rate of 2. All blocks after the second RRC filter operate at a rate of 1 because the decimation in the filter. Note that, while the Gaussian noise generator does not specify any interpolation nor decimation, it should still operate at a sampling rate of 2 so the noise is generated at the

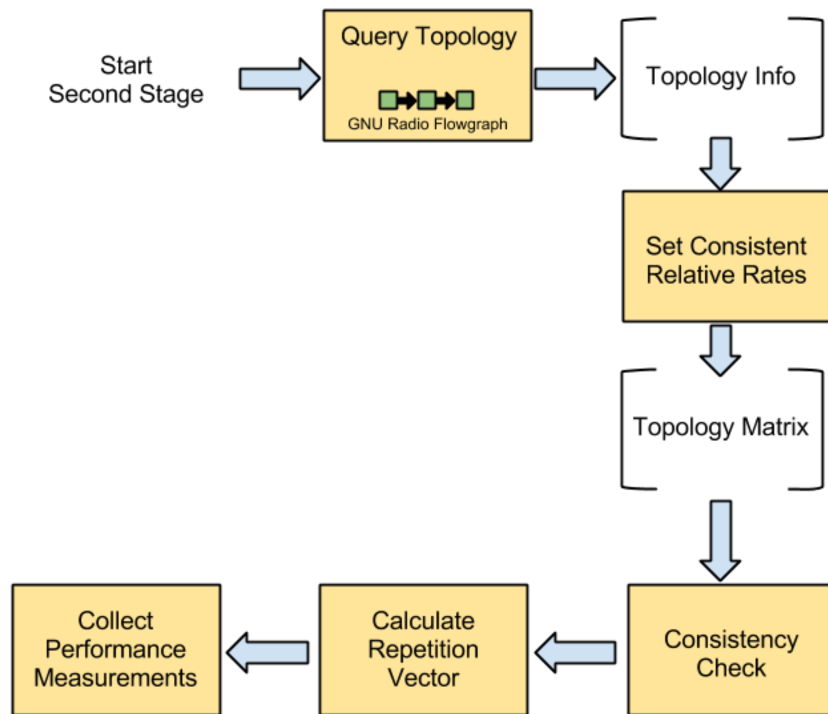

Figure 6.5: Second Stage

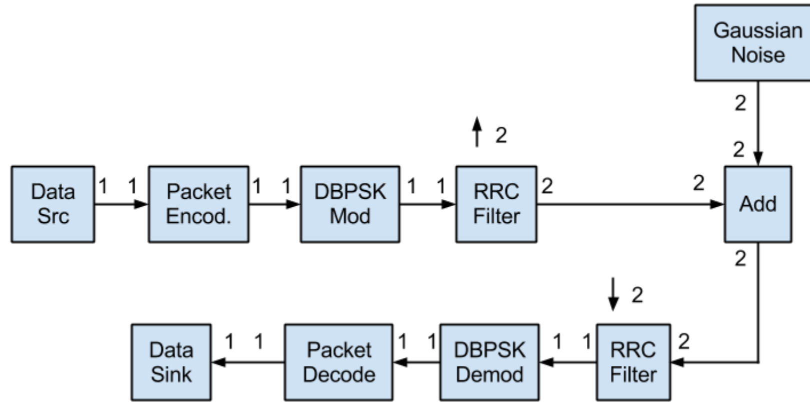

Figure 6.6: Flowgraph Perspective After the First Stage

same rate as the DBPSK radio signal.

The framework queries GNU Radio for the expanded flowgraph during the second stage, an example expanded flowgraph for Figure 6.6 is shown in Figure 6.7. The expanded flowgraph indicates that there are two gain blocks which are not in the original flowgraphs: a gain block after the first RRC filter and a second one before the second RRC filter. The assumption here is that the previous definition of the RRC filter is hierarchical and that the expanded GNU Radio flowgraph for the RRC filter includes a gain block. Also note that there is no relative rate information in the topology which is necessary to the construction of the second stage topology matrix. The only information available is the interpolation and decimation rates of the RRC filters.

A recursive algorithm addresses the issue of deducing the relative rates between the blocks to construct a topology matrix from the decimation and interpolation information in the graph. The algorithm assumes the following:

1. The flowgraph is a directed graph which is already implied by the use of the SDF MoC.

2. The relative rates at an input/output ports of a single arc are the same, *e.g.*, if the output port of an arc is 1 then the input port is also 1. This assumption is made because interpolation/decimation in GNU Radio is not implemented on a block's input port but physically in the block implementation.

3. The relative rates at an input/output port can only differ when a *stream-to-vector* or *vector-to-stream* blocks are used. In GNU Radio, a *stream-to-vector* block takes a stream of data and lumps them into a single vector, *e.g.*, for performing FFT operations where a vector of 1024 might be needed to perform the calculation. The *vector-to-*

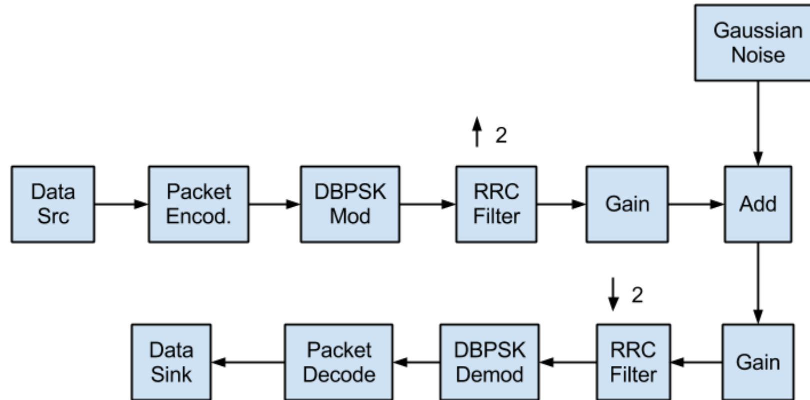

Figure 6.7: GNU Radio Expanded Flowgraph

stream block performs the inverse operation. These blocks do not change the sampling rates.

4. All blocks performing interpolation and/or decimation should declare those factors explicitly.

5. Each arc traversed in the flowgraph has a Boolean value indicating whether or not it was visited.

6. The algorithm's base-case (terminating-case) occurs when a sink node is reached or if the algorithm reaches a previously visited node which does not require a relative rate correction.

The algorithm pseudocode is shown in Algorithm 1. The helper function *CalcRevRelativeRate*, is used to traverse a flowgraph in the opposite direction, traversing backwards towards a source node instead of towards a sink node. This adjusts the relative sampling rate of a previously traversed path. The *CalcRevRelativeRate* function terminates as soon as it reaches a source node or a node which has not been visited before.

Going back to the DBPSK link example, the algorithm traverses the flowgraph recursively starting from the source blocks. The source blocks in this case are the *Data Source* and *Gaussian Noise* blocks. Beginning at the *Data Source* block, the algorithm starts setting the relative rate as 1, then continues recursively traversing the flowgraph assigning a relative rate to each block as 2 after the RRC filter because of the interpolation factor. The relative rate will continue being set to 2 until the second RRC filter block is reached then the remain blocks will have a relative rate of 1. Figure 6.8 shows the flowgraph overall relative rates at this stage.

Algorithm 1 CalcRelativeRate(node, cur_rate)

```
if node ≠ visited AND node ≠ sink then
    node_visited ← True
    if node_rate * cur_rate ≥ 1 then                                ▷ No fractional rates
        cur_rate ← factor * cur_rate
        node_rate ← cur_rate
        nodes ← get_next_nodes
        CalcRelativeRate(nodes, cur_rate)
    else                                      ▷ decimator node causing fractional rate
        CalcRevRelativeRate(nodes, 1/factor)   ▷ recursive fix rel rate of prev nodes
    end if
else if (node = visited) then  ▷ visited in past and rate does not match prev calculated rate
    if (node_rate ≠ cur_rate) then                            ▷ find new relative rate
        temp_rate ← least_common_multiple(cur_rate, node_rate)
        node_rate ← temp_rate
        correct1 ← temp_rate/cur_rate  ▷ find correction factor for current traversed path
        correct2 ← temp_rate/prev_node_rate  ▷ find correction factor for prev traversed paths
        if correct1 ≠ 1.0 then
            CalcRevRelativeRate(prev_nodes, 1/factor)
        else if correct2 ≠ 1.0 then
            CalcRevRelativeRate(prev_paths, 1/factor)
        end if
    else                                                             ▷ Base-case
        return True
    end if
else                                                                 ▷ Base-case
    return True
end if
```

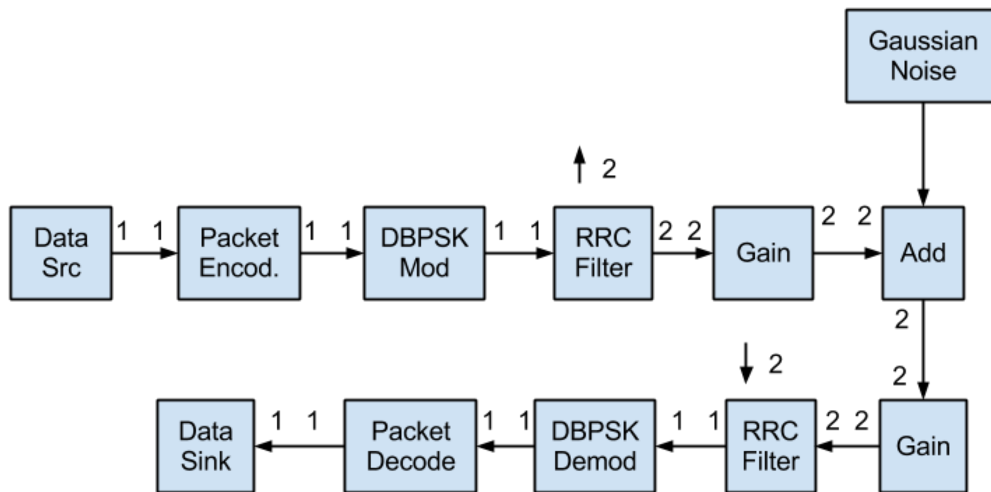

Figure 6.8: Setting Flowgraph Relative Rates Starting at Data Source Block

The algorithm then starts setting the relative rates starting from the *Gaussian Noise* block. When it reaches the *add* block, it realizes that it already traversed this block. But when it compares the expected relative rate of 1 with the *add* block's current relative rate is 2, it realizes that it needs to increase the relative rate calculation for the path it just traversed, the *Gaussian Noise* source, by a factor of 2. This stage is shown in Figure 6.9. The algorithm then terminates, resulting in the expected flowgraph with the relative rates show in Figure 6.10.

Figure 6.11 provides another example of how the algorithm can handle setting the relative rates of flowgraphs. In the figure, the algorithm starts by setting the relative rates at *Block 1*, *Block 2*, *Block 5*, and *Block 6*. When the algorithm sets the relative rates from *Block 3*, it continues to *Block 4*, and reaches *Block 5* it detects an input rate mismatch on the two input ports of *Block 5* where one port has a relative rate of 3 and another as 2, as shown in Figure 6.12. Figure 6.13 then demonstrates how the greatest common multiple of the ports is set as the new rate. While Figure 6.14 shows the final flowgraph with the final relative sampling rates. It is important to note that having different relative rates on the input ports of blocks is fine in terms of SDF semantics; however in the context of the discussed radio applications it is not acceptable. If the ability to perform decimation and/or interpolation on the input ports is needed, then the application designer needs the ability to set the relative rates explicitly for each block. This is not inherently supported in GNU Radio and might not be applicable in the context of the presented radio flowgraphs.

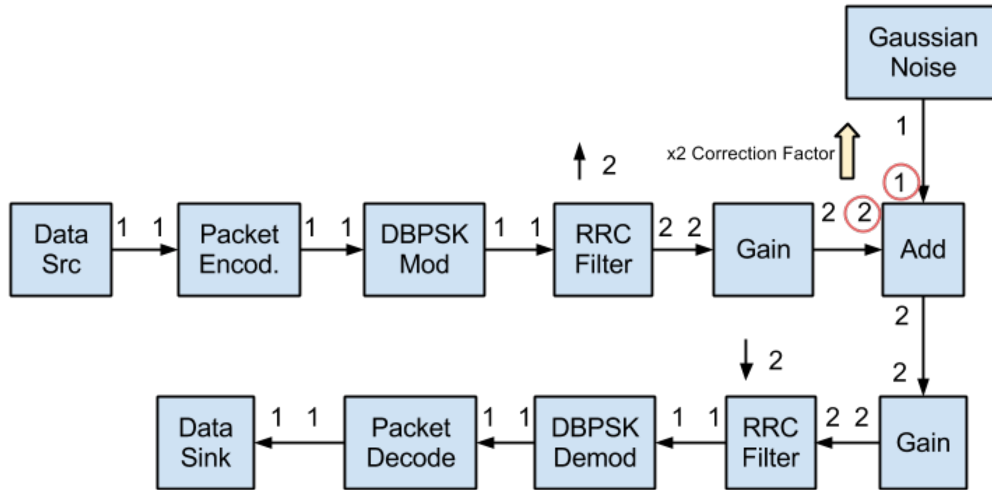

Figure 6.9: Flowgraph Relative Rate that Must be Fixed

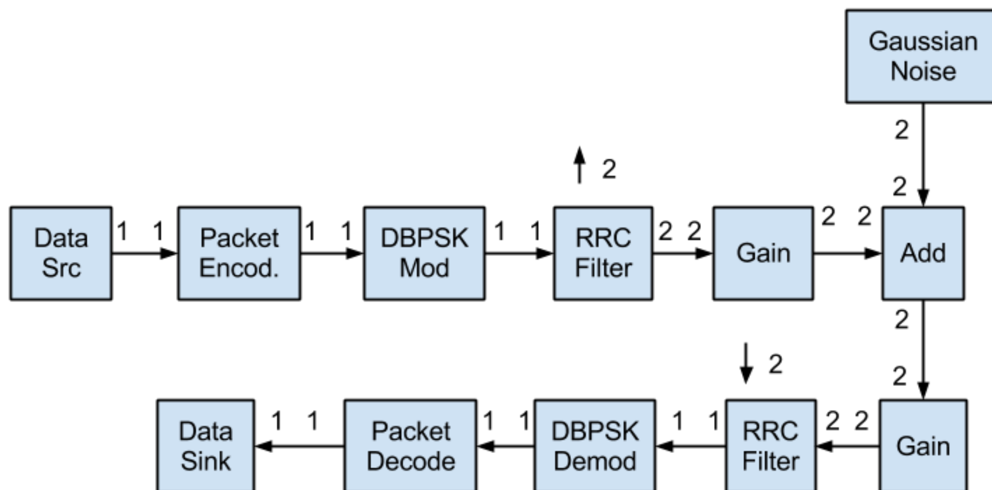

Figure 6.10: Final Flowgraph with Correct Relative Rates

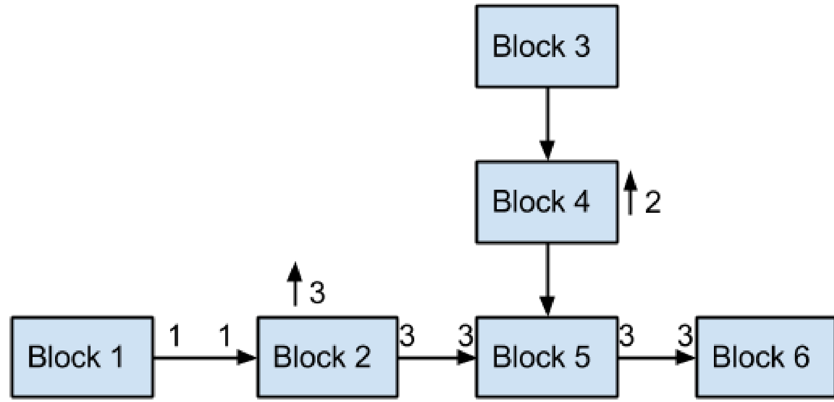

Figure 6.11: First Iteration of Setting Relative Rate Starting at Block 1.

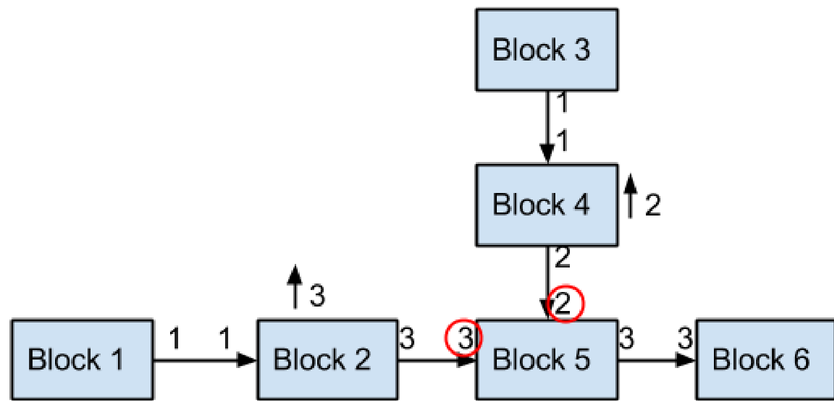

Figure 6.12: Second Iteration of Setting Relative Rate Starting at Block 3.

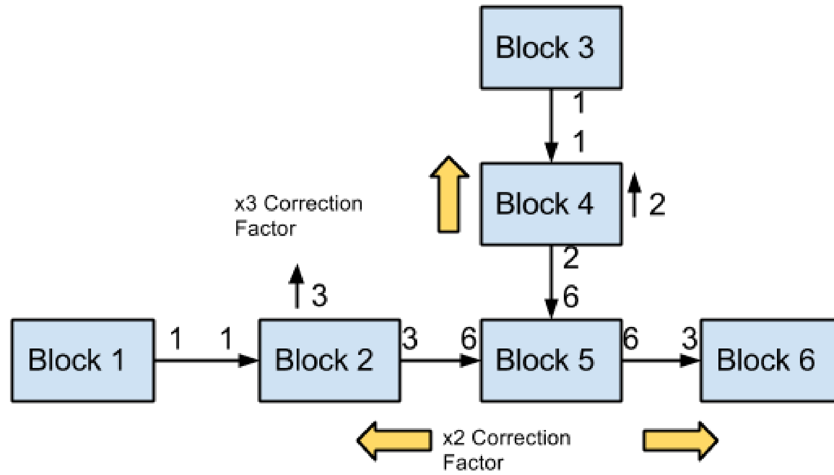

Figure 6.13: Flowgraph Relative Rate Mismatch.

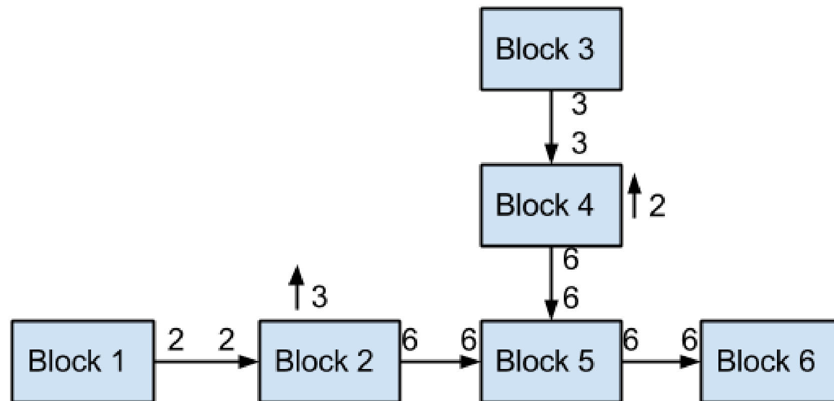

Figure 6.14: Flowgraph Relative Rate After Algorithm is Done.

6.4.3 Customized GNU Radio

GNU Radio does not support the construction of a topology matrix since it does not support the notion of block relative rates on a flowgraph level. It is necessary to add functionality to GNU Radio supporting the construction and calculation of topology matrices and repetition vectors needed in determining buffer sizes between blocks. The way GNU Radio currently determines buffers sizes is by allocating a fixed size of 64kB for each buffer. Blocks typically try to consume all of the samples in their input buffers to mitigate the overhead of context switching between threads. However, the ability to control the amount of data consumed in a single block execution can provide throughput/latency trade-off as discussed in Chapter 7.

Scheduler

GNU Radio schedules block uses a run-time scheduler for defining block execution. This means that when the flowgraph execution starts, the scheduler runs blocks which have sufficient input data to execute. This makes the GNU Radio scheduler *data-driven* or admissible, where blocks only execute when data is available. There are two variations of the scheduler, a Single-Thread-Scheduler (STS) which executes all the blocks in a single thread and a Thread-Per-Block (TPB) scheduler which runs each block in a separate thread. The STS scheduler has been deprecated in GNU Radio, because the TPB has better performance, therefore all GNU Radio performance measurements are taken using the TPB scheduler.

Customization

I developed a customized version of GNU Radio to accommodate creating a topology matrix, a repetition vector, a blocks list, and a channels list. The blocks list provides the names of the blocks found in a GNU Radio flowgraph where the index of the block in the list corresponds to an associated column in the topology matrix. The channels list provides all the channels found in the flowgraph with the name of the associated source and destination nodes. The index of the channel in the list corresponds to a row entry in the topology matrix.

The flowgraph start up sequence is modified in GNU Radio. Typically a flowgraph is created, then run using GNU Radio until the user decides to terminate the flowgraph. In the modified version, shown in Figure 6.15, after the flowgraph is created, the framework runs through the first stage design check as discussed earlier. A *pre-allocation* stage is then run allowing GNU Radio to expand the definition of any *hierarchical* blocks and define the overall flowgraph connections, but without allocating the buffers between blocks. At this stage the framework is able to query GNU Radio for the expanded flowgraph and can run through the second stage design check. The GNU Radio *allocation* stage is run next; this is where the buffers are allocated. Finally, the *go* stage indicates that the GNU Radio flowgraph has been fully defined and is ready to start executing.

Figure 6.15: Modified Start Up Sequence of Flowgraphs in GNU Radio.

Memory Allocation

As discussed before, the framework provides a *knob* which influences the buffer allocation between blocks by modifying the buffer scaling defined for the buffers. The actual size for the GNU Radio buffers is calculated as shown in Equation 6.1.

$$buffer_size = \Gamma_{i,j} * q_j * scale * item_size \tag{6.1}$$

Γ = topology matrix
q = repetition vector
$scale$ = buffer scale
i = channel number
j = block number
$item_size$ = size of data type, *e.g.*, int, float, ... etc

6.5 Interface

A CE is able to interact with the framework through the *interface* module. A CE defines an input radio flowgraph written in Occam and establishes a buffer scaling factor, a desired range of radio flowgraph total memory usage, throughput, and latency.

6.6 MIP Solver

The repetition vector of the SDF graph is calculated by solving the Equation 3.7 by setting up and solving an associated MIP. The GNU Linear Programming Kit (GLPK) [81] is an open source solver which is used in conjunction with its Python interface [82] to solve the MIPs. The general format for solving the MIP is as follows for the SDF graph in Figure 6.16:

$$\Gamma * q = \mathbf{0} \tag{6.2}$$

Figure 6.16: SDF Graph Used to Calculate Repetition Vector.

$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -2 \end{bmatrix} * \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Minimize:

$$q_1 + q_2 + q_3$$

Subject To:

$$\begin{aligned} q_1 \quad -q_2 &= 0 \\ 2q_2 \quad -2q_3 &= 0 \end{aligned}$$

Bounds:

$$\begin{aligned} q_1 &\geq 1 \\ q_2 &\geq 1 \\ q_3 &\geq 1 \end{aligned}$$

The objective function of the MIP is denoted by the *Minimize* construct, indicating that the objective is to minimize the total number of executions for each block. This calculates the minimum number of firings necessary to construct a repetition vector. The *Subject To* construct denotes the constraints of the MIP which represent the topology matrix multiplied by the nullspace vector q as shown in Equation 3.7. The *Bounds* of the MIP indicate that each block in the q vector should execute at least once, thus excluding the trivial solution of all zeroes. The *integer* restriction is because the repetition vector entries need to be integers since the number of block executions cannot be a fractional.

6.7 Data Visualization

This module is responsible for taking all the raw performance measures from the GNU Radio runs and plot them using the matplotlib Python library [83].

6.8 Perspective

The process of collecting performance measurements using various buffer scalings and observing its impact on performance measurements can take several hours making the proposed

framework more suited for design space exploration for CR implementation and design. Alternatively, the proposed framework can be used as part of the *learning* process of a CR. For example, neural networks is an artificial intelligence technique which can be applied for CRs [84], using this technique implies that a CR goes through a learning process by observing the impact of applying various configurations on the radio [84].

Chapter 7

Example Application Use

To test the presented framework, three proof-of-concept digital radios are implemented: DBPSK transmitter, DBPSK receiver, and a one-way DBPSK communication link. For the experiment scenarios, a CE is able to specify the radio flowgraph semantics using Occam programs and the framework then generates SDF based SDR implementations. A CE is then able to control buffer scaling, the computational knob, and it can observe its effect on total memory utilization, latency, throughput, and reconfiguration times for the flowgraph.

In this chapter I will first present an overview of the knobs and meters available to CEs and follow that with a discussion of the digital radio architectures used. Next I present the reactive models for the radio flowgraphs, and follow that by a discussion of the generated real-time, or SDF, models. Finally, I present the effect of buffer scaling in regards to flowgraph total memory utilization, latency, throughput, and reconfiguration times.

7.1 Testing

In developing the framework and sample applications, it was important to check if the outcome of each step was as expected. Basically the reactive radio models need to be correct so the Ptolemy and GNU Radio flowgraphs can be generated. The DBPSK radio designs need to be correct for the radios to be implemented properly, and the framework needs to calculate the topology matrix and repetition vectors correctly for the output buffers to be allocated. The following details the test checks taken during the programming process.

7.1.1 Reactive Models

The Occam radio flowgraph definitions are compiled and run using the KRoC compiler. By making sure that there are no compile errors we know that the flowgraph definitions are

syntactically correct. By running the programs and have them exit gracefully we are able to ensure that there are no deadlocks in the program; deadlocks can occur if the meta-data information is not propagating to the downstream processes correctly. This happens, for example, if a reading process reads in more or less data than it should or if a writing process writes more or less data than it should. In both of these situations the program can get stuck in a deadlock since Occam implements blocking reads and writes in accordance with CSP.

To check that the Occam programs are correctly setting up input/output connections, the output XML files for GNU Radio and Ptolemy are opened and the block connections are visually inspected.

7.1.2 DBPSK Radio

In testing the DBPSK transmitter, a spectrum analyzer is used to make sure that a DBPSK signal is received at the expected RF center frequency 462.5625 MHz with a symbol rate of 320 kHz which is the equivalent of the first stage sampling rate in this scenario. The output of the spectrum analyzer can be seen in Figure 7.1, where the x-axis represents the real element of the signal and the y-axis is the imaginary axis of the signal. A DBPSK signal only needs a real component of the signal and would switch between phases 0 and π with a symbol rate of 320kHz, as seen in the figure. The receiver is tested by sending a music file from the transmitter and listening to it, by sending and receiving the music file over-the-air, two way communication is established between both radios.

7.1.3 SDF Graphs

Graph definitions from [54] are passed to the framework and the results from the framework are compared with those from [54]. The purpose is to ensure that the framework can properly identify inconsistent graphs and that the framework is also able to calculate graph repetition vectors after determining that the graph is consistent. The framework does not detect whether a deadlock occurs in a graph since the GNU Radio scheduler is able to do so, therefore this functionality is not tested here.

The repetition vector and second stage topology matrices for the example applications presented in this chapter will be provided. In doing so readers are able to manually inspect the correctness of the output by comparing it to the expected behavior as seen from the first stage topology matrix. Also the equations from Chapters 3 and 6 can be used to further check the correctness of the output values.

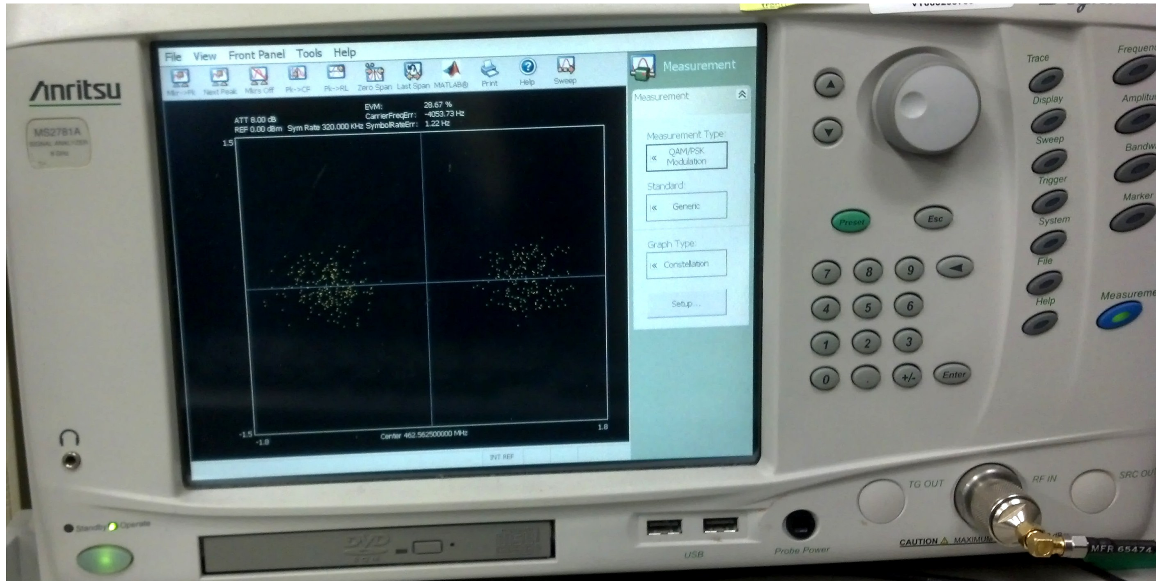

Figure 7.1: DBPSK Transmitter Constellation.

Inconsistent Graph

Graph consistency is measure of whether the sampling rates are *malformed*, therefore causing a deadlock or buffer overflow during execution. In case a CE requests an inconsistent radio flowgraph, it is important to detect it and report the error back to the CE. Here I demonstrate that the framework is able to detect and inconsistent graph by passing it an inconsistent graph, Figure 7.2, provided by the authors in [54].

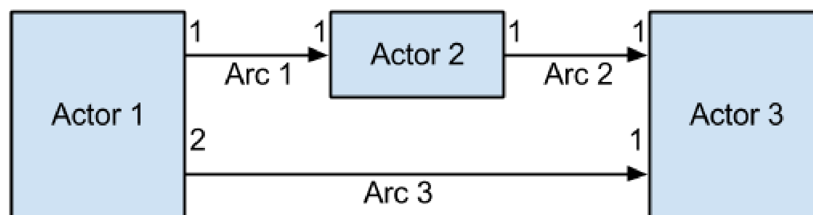

Figure 7.2: Inconsistent Graph.

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix}$$

This graph should be inconsistent since the rank of the matrix is 3 and from Equation 3.2, we know that the rank must be less than the number of actors for the graph to be consistent. When the associated topology matrix is inputted into the framework it states that the graph is inconsistent and shows that the rank of the matrix is 3.

Repetition Vector

The repetition vector of a graph specifies how many times each block must fire during the periodic execution of the schedule. The repetition vector and the topology matrix are needed to calculate the exact buffer sizes needed for bounded memory execution. Here I demonstrate that the framework is able to calculate the repetition vector for an example graph, Figure 7.3, provided by the authors of [54].

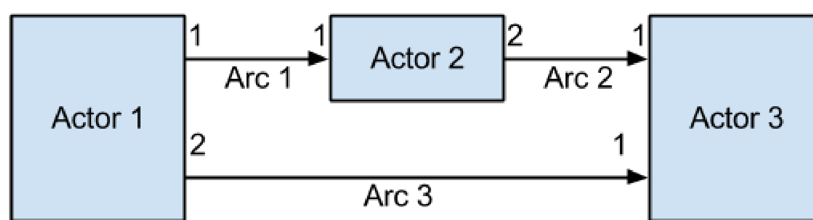

Figure 7.3: Consistent Graph.

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix}$$

This graph should be consistent since the rank of the matrix is 2 and from Equation 3.2, we know that if the rank is less than the number of actors then the graph is consistent. When the associated topology matrix is passed to the framework, the framework states that the graph is consistent with rank 2. When the topology matrix is passed to MIP interface, the output repetition vector is:

$$\Gamma = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

This is the same expected repetition vector from [54].

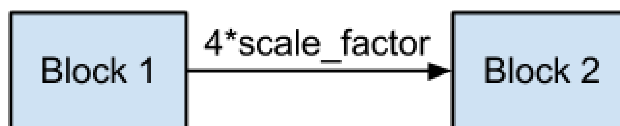

Figure 7.4: Buffer Scaling.

7.2 Performance Measurement Runs

All performance measurement testing is done using a first generation USRP with a WBX daughterboard connected to a desktop computer. The desktop computer has the following specifications:

- **OS**: Ubuntu 12.04, Kernel 3.2.0-39
- **CPU**: Intel Core 2 CPU 6600 @ 2.4 GHz
- **Memory**: 2 GB

It is important to highlight that the results presented in this chapter are specific to the architecture of the computer used, meaning that performance measurements can vary depending on the specific platform running these applications.

7.2.1 Knob

The computational knob available is the buffer scaling factor indicating how much bigger flowgraph buffers should be than the calculated minimum size. Looking at Figure 7.4, assuming that the minimum buffer size necessary between Block 1 and Block 2 is 4, then the scaling factor is a measure of how much bigger the buffer should be.

The scaling factors being considered are powers of two, after each iteration the scaling factor is multiplied by two. The flowgraph performance measurements are collected by first executing the flowgraph using the GNU Radio default buffer allocation policy, which is 64KB per buffer. The performance measures from the default GNU Radio allocation is then stretched to a DC value, where the values stretch over the buffer scaling interval to make comparing the default GNU Radio allocation method and the topology based method easier. The final size of the buffer is described by previously discussed equation, Equation 6.1 which is:

$$buffer_size = \Gamma_{i,j} * scale * q_j * n * item_size$$

Γ = topology matrix
q = repetition vector
$scale$ = buffer scale
i = channel number
j = block number
$item_size$ = size of data type, *e.g.*, int, float, ... etc

7.2.2 Meters

Flowgraphs are run the first time using the GNU Radio default buffer allocation method and repeated 10 times so the results can be averaged. Each graph execution runs for 5 minutes, Flowgraphs are then run for multiple buffer scaling values as in Equation 7.1. Please note that if the USB interface to and from the USRP is small enough it can cause a segfault; therefore a scaling exponential offset value is used increase the buffer scaling to the point the USB buffer does not segfault.

$$buffer_scale = 2^{n+m} \tag{7.1}$$

n = Scaling exponential value
m = Scaling exponential offset value

For each iteration the following performance measures, or computational meters, are collected:

- **Memory Utilization**: This is done using Equation 6.1 to calculate the memory needed for each output buffer when the topology matrix based buffer allocation method is used. When the default GNU Radio buffer allocation method is used, the total memory utilization is calculated as 64kB multiplied by the number of output buffers present in the flowgraph.
- **Latency**: GNU Radio measures the average execution time for each block and the latency is calculated as the sum of the average execution time for each block in the radio data path.
- **Throughput**: GNU Radio measures the average number of output samples produced by each block and the throughput is calculated as the average number of output samples for a certain block divided by the average amount of execution time for the block.
- **Reconfiguration Time**: This is measured as the amount of time needed to set up a GNU Radio flowgraph in addition to the amount of time needed to tear down the

flowgraph. Looking at reconfiguration time in the performance measurement graphs in this section, we see that it is not directly correlated with the amount of buffer memory allocated. This means that buffer allocation does not dominate the time required in setting up GNU Radio flowgraph. Many other steps are involved in setting up a flowgraph, like parsing through flowgraph blocks, expanding the definition of hierarchical blocks, setting up block implementation details, configuring the scheduler, and translating back and forth between Python/C++ through the SWIG wrapper interface. The variations in the reconfiguration times can be attributed to unpredictable cache performance [85] and the fact that operating systems can influence the cache performance [86]. So the reconfiguration time measurement provides an order of magnitude assessment for CEs to take into account when deciding whether or not to choose new radio configurations.

7.3 Architecture

This section presents the radio architecture used for the DBPSK transmitter, receiver, and one way communication link.

7.3.1 DBPSK Transmitter

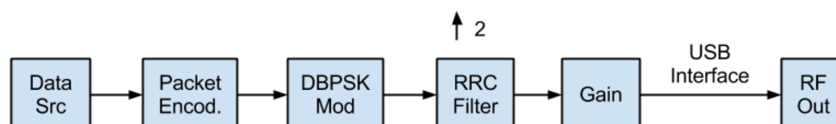

Figure 7.5: DBPSK Transmitter

Figure 7.5 describes the DBPSK transmitter architecture used. The data source is a pre-recorded music wave file which is running in an infinite loop. The packet encoder takes the data from the file and breaks it down into chunks of data and encapsulates them into a packet. The packets are then modulated in the DBPSK modulation block which differentially encodes the packets. The RRC filter is used for pulse shaping and it interpolates the incoming data from the data processing sampling rate to a sampling rate compatible with the USRP. The data stream is then passed over the USB interface to the USRP for transmission.

7.3.2 DBPSK Receiver

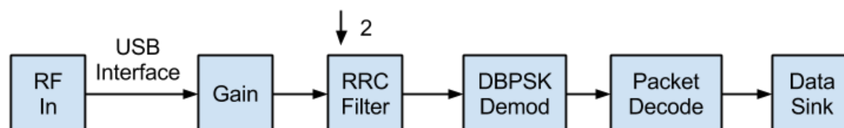

Figure 7.6: DBPSK Receiver

The DBPSK receiver design used can be seen in Figure 7.6. The *RF In* block corresponds to the USRP receiver interface which streams data into the flowgraph via USB. The *Gain* block performs scaling on the signal coming in from the USRP before it is processed. The RRC filter performs the receiver pulse shaping functionality and decimates the sampling rate to the rate at which the data will be processed. The DBPSK demodulator uses a differential decoder to demodulate the received signal. The packet decoder block reads the incoming packets, extracts the data payload, and streams the data content to the *Data Sink* block which saves the received data into a file.

7.3.3 DBPSK Link

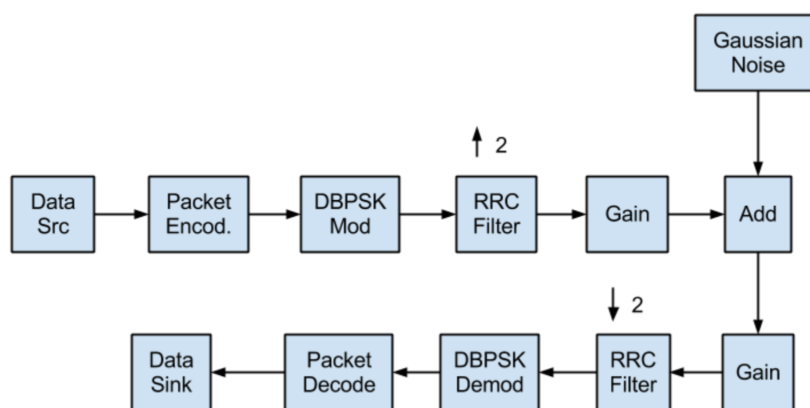

Figure 7.7: DBPSK Link Over an Additive White Gaussian Noise Channel

The DBPSK link design used can be seen in Figure 7.7. The design is essentially composed of a transmitter and receiver with the addition of the *Gaussian Noise* and *Add* blocks. Gaussian noise is added to the signal as a representation of noise in the spectrum which

is a reasonable method of simulation channel noise since it represents a uniform signal in the frequency domain [70]. This flowgraph provides a simulation of a DBPSK transmitter sending data over a Gaussian channel to a corresponding receiver.

7.4 Reactive Models

This section discusses the reactive models defined using CSP for the DBPSK transmitter, receiver, and link. The models are implemented in the Occam language as discussed previously. The following is a description of the radio flowgraph using the concurrency operator and the meta-data discussed earlier in this section are passed through the processes.

7.4.1 Processes and Meta-Data

The following is a listing of the processes used in defining the radio flowgraph and the associated meta-data.

- Parameter Generator (PG).
- Data Source (DS).
- Data Sink (DSI).
- Packet Encoder (PE).
- Packet Decoder (PD).
- DBPSK Modulator (DM).
- DBPSK Demodulator (DD).
- RRC Pulse Shaping Filter (RRC).
- Baseband Scale (BS).
- Gaussian Noise Channel (GN).
- Additive Channel (ADD).
- RF Out (RO): Representing a USRP transmit interface.
- RF In (RI): Representing a USRP receive interface.

The following is a summary of the parameters getting pass by the parameter generator process:

- **Sampling Rate**: The data processing sampling rate. The sampling rate used is 320 KHz.
- **Sampling Rate2**: The sampling rate going into and out of the USRP over the USB interface. The second sampling rate used is 640 KHz.
- **Carrier Frequency**: The transmit/receive frequency. The public Family Radio Service (FRS) channel 1: 462.5625 MHz is used for all the experiments discussed.
- **Roll-off**: Defines the roll-off factor used for the RRC filters. All the radio flowgraphs use 0.35 as the roll-off factor.
- **Gaussian Gain**: The Gaussian noise's gain level which is set to 0.3.
- **RF Gain Transmit**: The gain for the USRP's transmit interface.
- **RF Gain Receive**: The gain for the USRP's receive interface.

7.4.2 DBPSK Transmitter

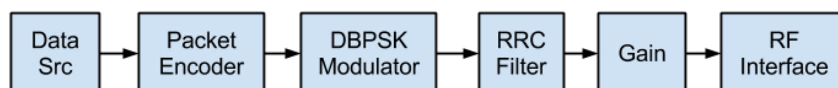

Figure 7.8: DBPSK Transmitter Reactive Model

Figure 7.8 shows the expected DBPSK transmitter reactive model. We can see that the model only contains information regarding which blocks are instantiated and the interconnect between the components but does not contain information about rates nor interpolation factors. The DBPSK transmitter system is defined as follows:

$$TX = PG \| DS \| PE \| DM \| RRC \| BS \| RO$$

Since the parallel operator does not define any connections between the processes, channels are used to implement the connections between each process as discussed in Chapter 6. The channel definitions for the Occam flowgraph definition can be seen in Table 7.1, where the process name is detailed along with its input and output channels.

The parameter generator block is a source block which sends the meta-data to the various processes in order to share the radio configuration parameters. The subsequent processes forward the configurations until they reach the *rfOut* process as discussed in Chapter 6.

Table 7.1: Channel Connections for DBPSK Transmitter Reactive Model.

Process Name	**Output Channel**	**Input Channel**
parameterGen	setupC	N/A
dataSrc	dataToEncC	setupC
dbpskEnc	encToModC	dataToEncC
dbpskMod	txToChanFiltC	encToModC
channelFilter	filterToScaleC	txToChanFiltC
basebandScale	scaleToAddC	filterToScaleC
rfOut	N/A	scaleToAddC

The process definition for the parameter generator block, a source block, is as follows:

$$PG = right!sr1 \rightarrow right!sr2 \rightarrow right!bbgain \rightarrow right!cf \rightarrow right!roff \rightarrow right!ag \rightarrow PG$$

The process definition for the RF Out block, a sink block, is as follows:

$$RO = left?sr1 \rightarrow left?sr2 \rightarrow left?bbgain \rightarrow left?cf \rightarrow left?roff \rightarrow left?ag \rightarrow RO$$

The process definition for the remaining blocks is as follows:

$$BLOCK = left?sr1 \rightarrow right!sr1 \ldots \rightarrow left?ag \rightarrow right!ag \rightarrow BLOCK$$

7.4.3 DBPSK Receiver

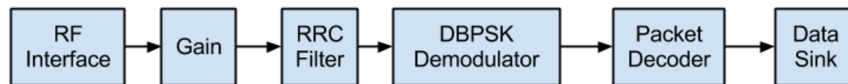

Figure 7.9: DBPSK Receiver Reactive Model

Figure 7.9 shows the expected DBPSK transmitter reactive model. We can see that the model only contains information regarding which blocks are instantiated and the interconnect between the components but does not contain information about rates nor decimation factors. The DBPSK receiver system is defined as follows:

Table 7.2: Channel Connections for DBPSK Receiver Reactive Model.

Process Name	**Output Channel**	**Input Channel**
parameterGen	setupC	N/A
rfIn	rfInToScaleC	setupC
basebandScale2	scaleToRrcFiltC	rfInToScaleC
channelFilter2	rrcFiltToDemodC	scaleToRrcFiltC
dbpskDemod	demodToDecC	rrcFiltToDemodC
dbpskDec	decToDataOutC	demodToDecC
dataOut	N/A	decToDataOutC

$$RX = PG \| RI \| BS \| RRC \| DD \| PD \| DSI$$

Since the parallel operator does not define any connections between the processes, channel definitions are used to define the connections between each process as discussed in Chapter 6. The channel definitions for the Occam flowgraph definition can be seen in Table 7.2, where the process name is detailed along with its input and output channels.

The parameter generator block is a source block that sends the meta-data to the various processes in order to share the radio configuration parameters. The subsequent processes forward the configurations until they reach the *dataOut* process as discussed in Chapter 6.

7.4.4 DBPSK Link

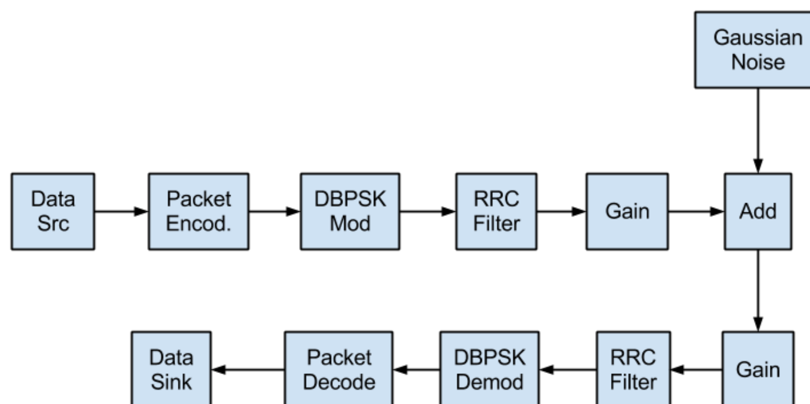

Figure 7.10: DBPSK Link Reactive Model

Table 7.3: Channel Connections for DBPSK Link Reactive Model.

Process Name	**Output Channel**	**Input Channel**
parameterGen	setupC	N/A
parameterGen	setup3C	N/A
dataSrc	dataToThrottC	setupC
throttle	throttToEncC	dataToThrottC
dbpskEnc	encToModC	throttToEncC
dbpskMod	txToChanFiltC	encToModC
channelFilter	filterToScaleC	txToChanFiltC
basebandScale	scaleToAddC	filterToScaleC
gauss	gaussToScaleC	setup3C
gaussScale	gaussScaleToAddC	gaussToScaleC
add	addToBasebandScale2C	scaleToAddC gaussScaleToAddC
basebandScale2	scaleToRrcFiltC	addToBasebandScale2C
channelFilter2	rrcFiltToDemodC	scaleToRrcFiltC
dbpskDemod	demodToDecC	rrcFiltToDemodC
dbpskDec	decToDataOutC	demodToDecC
dataOut	N/A	decToDataOutC

Figure 7.10 shows the expected DBPSK link reactive model. We can see that the model only contains information regarding what blocks are instantiated and the interconnect between the components but does not contain information about rates nor interpolation factors. The DBPSK link system is defined as follows:

$$TXRX = PG \| DS \| PE \| DM \| RRC \| BS \| BS \| RRC \| DD \| PD \| DSI \| GN \| ADD$$

The channel definitions for the Occam flowgraph definition can be seen in Table 7.3, where the process name is detailed along with its input and output channels. Please note that the *throttle* process is added because GNU Radio expects that there will be a block in the flowgraph which will enforce the requested sampling rates, which is done via the USRP blocks in the transmit and receive flowgraphs. Since there is no USRP block in the DBPSK link flowgraph, the throttle block is used to enforce the requested sampling rates.

7.5 Real-Time (SDF) Model

7.5.1 DBPSK Transmitter

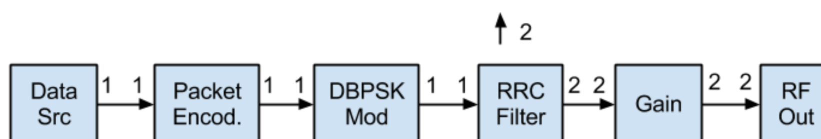

Figure 7.11: DBPSK Transmitter SDF Model

The SDF model of the DBPSK transmitter can be seen in Figure 7.11. The first stage topology matrix yields the following:

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 2 & -2 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 \end{bmatrix}$$

Each column in the topology matrix corresponds to one block in the following list.

0. Data Source.
1. Packet Encoder.
2. DBPSK Modulator.
3. RRC Pulse Shaping Filter.
4. Baseband Scale.
5. RF Out.

The calculated repetition vector is:

$$q = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

After generating the first stage topology matrix and passing it to GNU Radio the flowgraph definition is revised in GNU Radio to include the expanded definitions of any hierarchical blocks in the flowgraph definition. The following is the list of blocks which were instantiated by GNU Radio; please note that GNU Radio has renamed some blocks from what they were named in their reactive definitions.

0. **file_source0**: The data file source.
1. **message_sink0**: The input of the packet encoder.
2. **message_source0**: The output of the packet encoder.
3. **packed_to_unpacked_bb0**: Breaks the bits in a data type into separate data types, *e.g.*, 1011 would be separated into four different objects.
4. **map_bb0**: Used to map binary data to their Grey Code equivalent. This is not relevant for DBPSK since it only has two binary values but is relevant to other phase shift keying modulation which transmit more than one bit at a time.
5. **diff_encoder_bb0**: Differential encoder.
6. **chunks_to_symbols_bc0**: Maps values to their associated symbols.
7. **pfb_arb_resampler_ccf0**: Polyphase filterbank used to implement the RRC filter.
8. **multiply_const_vcc0**: Used to scale the input data.
9. **multiply_const_vcc1**: Used to scale the baseband signal before passing it to the USRP.
10. **gr uhd usrp sink0**: The USRP interface.

After GNU Radio expands the flowgraph definition, a second topology matrix is constructed from information regarding any blocks performing data rate interpolation and/or decimation. The second stage topology matrix, Γ, and repetition vector, q, are as follows:

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 \end{bmatrix}$$

$$q = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

We can see from the topology matrix that the *file_source* and *message_sink* blocks are decoupled from the remainder of the flowgraph. This is because GNU Radio handles packet construction in Python and it uses the *message_sink* to move the data from the C++ domain in the flowgraph to the Python domain. The data is then returned to the flowgraph using the *message_source* block, which sends the data from the Python domain back to the C++ domain. This is all right in terms of SDF semantics since it means that the flowgraph should be treated as two separate flowgraphs each with their own source blocks. Another thing to note is that in the topology matrix, the *packed_to_unpacked* block interpolates by a factor of 8 because it takes 8-bit input data, in this case a float object, and modulates each bit separately. The *pfb_arb_resampler* block which implements the RRC filter functionality also interpolates by a factor of 2 as expected since $\frac{samplingRate2}{samplingRate} = 2$.

Memory Utilization

The transmitter buffer scaling is varied from $n = [1, 14]$

$$buffer_scale = 2^n$$

Looking at Figure 7.12 we consider the radio flowgraph memory utilization in Kilo Bytes for the specified exponent scaling sizes. The GNU Radio default memory allocation remains constant since it depends on allocating a fixed output buffer size for all blocks, 64kB. The topology matrix allocation approach cross the default allocation plot when its memory utilization exceeds what GNU Radio typically needs.

As we can see from the figure, it is possible to execute the same DBPSK transmitter flowgraph with a wide range of memory utilization. Therefore, a CE is able to choose an appropriate buffer sizing to address any memory constraints it might have for radio implementation.

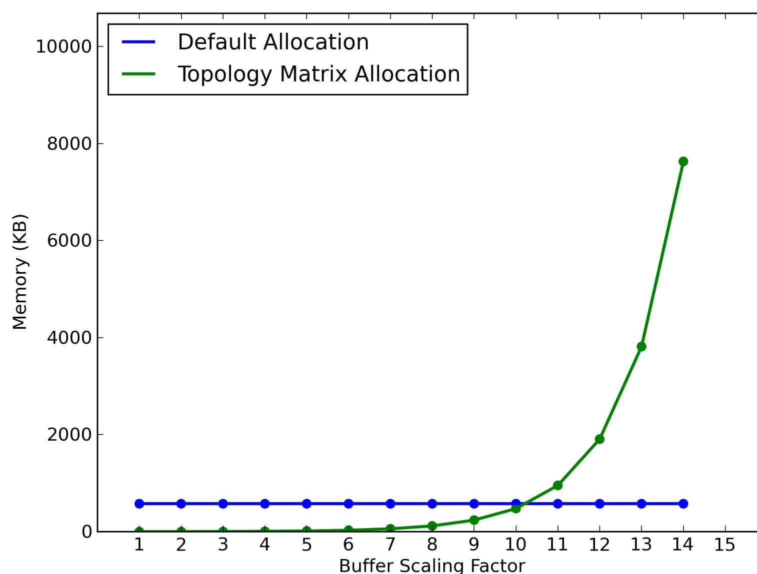

Figure 7.12: Memory Utilization for DBPSK Transmitter

Latency

The latency graph for the DBPSK transmitter can be seen in Figure 7.13. The latency is defined as the average amount of time the flowgraph spends processing data in the *Gain* block, from Figure 7.11. The buffer scaling is done in the same manner discussed previously. As we can see from the graph, a CE is able to achieve different latency measures depending on the scaling of the buffers. The bars in the graph represent the standard deviation for the different buffer allocation methods. For the default buffer allocation method, the standard deviation is shown for the first point only since it is the same for remaining points.

In situations where a CR is running radio and non-radio applications concurrently it can specify an acceptable latency range for the SDR implementations. This can be especially important if the SDR application is streaming voice or video data where the streaming nature of the application can provide certain latency requirements. Please note that as soon as the memory utilization in Figure 7.12 crosses the default memory utilization method the flowgraph latency starts increasing drastically. So it might not be worth it for a CE to scale the buffers by a factor of 2^{11}.

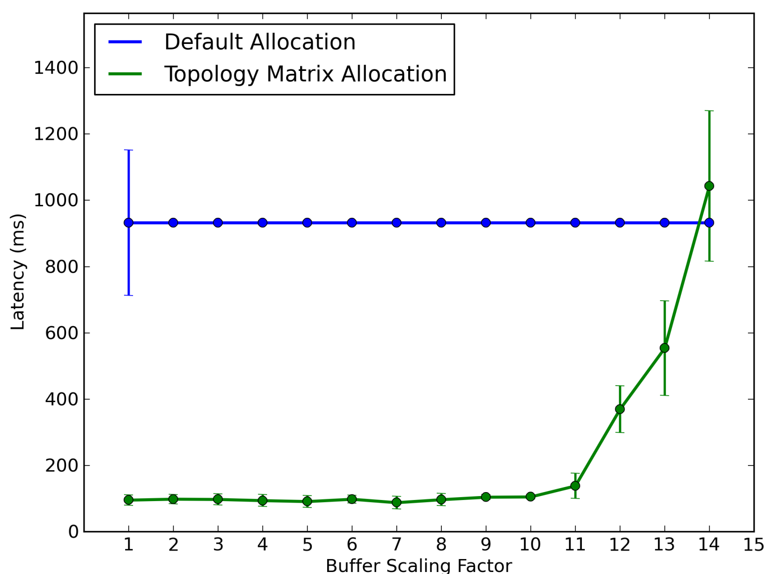

Figure 7.13: Latency for DBPSK Transmitter

Throughput

The throughput graph for the DBPSK transmitter can be seen in Figure 7.14. The throughput of the graph is defined as the average amount of data produced by the *Gain* block, from Figure 7.11, divided by the average amount of time the *Gain* block spends processing data. The bars representing graph standard deviation are small enough where they are not visible in the graph.

It can be seen from the figure that as the buffer sizes and throughput are directly proportional, where an increase in the buffer sizes can increase throughput. A CE is able to compromise between the desired throughput, latency, and memory utilization as needed by the application and/or RF channel environment by choosing appropriate buffer scaling factors. We can see that the improvement in throughput after the memory utilization graph exceeds that of the standard GNU Radio method, Figure 7.12, that it might not be worth it for CEs considering the increased amount of memory needed to implement the design and the increased latency, as was seen in Figure 7.13.

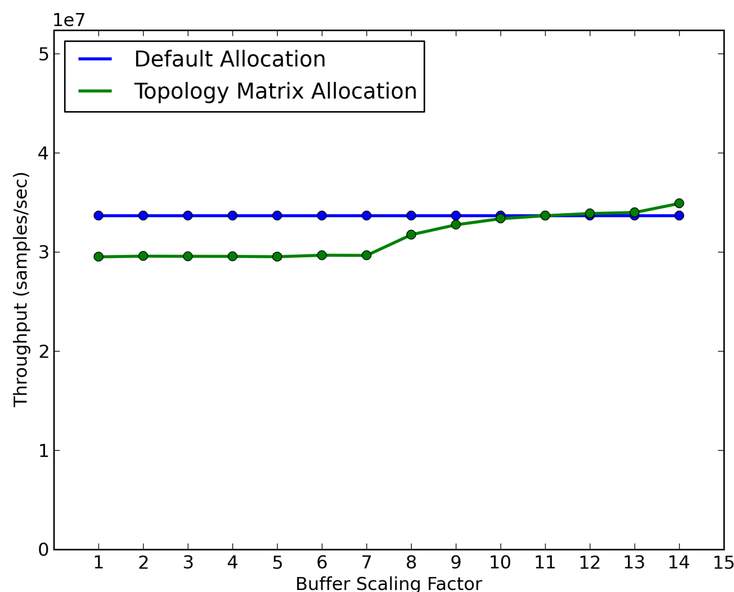

Figure 7.14: Throughput for DBPSK Transmitter

Reconfiguration Time

The graph of the reconfiguration time for the DBPSK transmitter appears in Figure 7.15. The reconfiguration time is defined as the amount of time spent setting up and configuring the flowgraph in addition to the time spent tearing down the flowgraph, basically deallocating the resources associated with the flowgraph. The bars in the graph represent the reconfiguration time standard deviation, for the default GNU Radio buffer allocation method, the standard deviation is plotted for the initial point but it is the same value for the remaining points. As can be seen in the graph, there is little correlation between the buffer sizing and the reconfiguration time. In this context CEs should use the reconfiguration time as an estimate of the time it takes to reconfigure the current flowgraph; *e.g.*, if a CE wants to modify filter parameters or flowgraph sampling rates, it can gauge whether it is worth spending the time to do so or not.

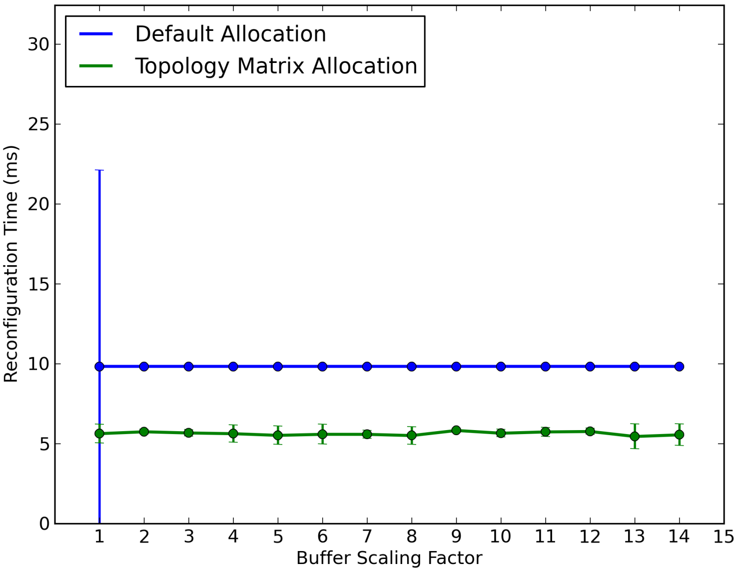

Figure 7.15: Reconfiguration for DBPSK Transmitter

7.5.2 DBPSK Receiver

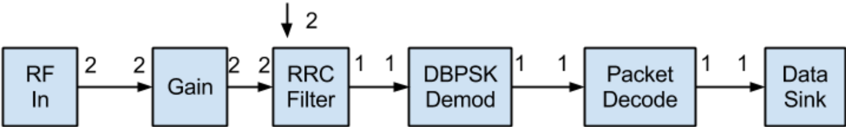

Figure 7.16: DBPSK Receiver SDF Model

The DBPSK receiver SDF model can be seen in Figure 7.16. The first stage topology matrix is as follows:

$$\Gamma = \begin{bmatrix} 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 2 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

The following is a list of the blocks present in the first stage topology matrix where the block

in each column of the topology matrix correspond to its enumeration in the list.

0. RF In.
1. Baseband Scale.
2. RRC Pulse Shaping Filter.
3. DBPSK Demodulator.
4. Packet Decoder.
5. Data Sink.

The calculated repetition vector is as follows:

$$q = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The generated flowgraph description is then passed to GNU Radio which will expand the definition of any hierarchical blocks. The new expanded topology matrix is:

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

The calculated repetition vector is as follows:

$$q = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The following is the list of blocks invoked in the expanded flowgraph as defined by GNU Radio, the enumeration of the blocks in the list corresponds to the column enumeration of the topology matrix.

0. **message_source0**: Outputs the data payload from a packet.
1. **file_sink0**: Used to save the received data to an output file.
2. **gr uhd usrp source0**: The USRP receive RF interface.
3. **multiply_const_vcc1**: Used for scaling.
4. **multiply_const_vcc0**: Used to scale the data stream received from the USRP interface.
5. **gr_agc2_cc0**: Automatic gain control, this block adjusts input receive power to maintain constant input power.
6. **fll_band_edge_cc0**: Frequency lock loop using band-edge filters. This block performs the matching RRC filter.
7. **pfb_clock_sync_ccf0**: Polyphase clock recovery filterbank. Generates a clock signal from the received data which helps in improving the receive quality. This blocks also implements the down sampling since it observes $\frac{samplingRate2}{samplingRate} = 2$ bit samples then generates a single bit.
8. **constellation_receiver_cb0**: Maps the received data into DBPSK constellation points, 0 and 1.
9. **diff_decoder_bb0**: Performs the differential decoding from the received bits.

10. **unpack_k_bits_bb0**: Unpacks data from a single data type into multiple data types.
11. **correlate_access_code_bb0**: Checks that the received packet has the same access code, receive address, the receiver is expecting, if they are different then the receiver will discard the packet.
12. **framer_sink_10**: Organizes received data into packets instead of bit streams so the packet decoder can extract data from the received packets.

We can see from the second stage topology matrix that the *message_source* and *file_sink* blocks form a disconnected SDF graph from the remainder of the receiver flowgraph. The reason is that GNU Radio handles the packet decoding process in Python and uses the *message_sink* block to send the data stream to the Python domain for processing and then returns data back into the flowgraph by using the *message_source* block.

Memory Utilization

The receiver buffer scaling is varied from $n = [1, 11]$ and is offset by a factor of 4 as can be seen in Equation 7.2. The offset factor is used to scale the USRP buffer sizes to where it would not segfault.

$$buffer_scale = 2^{n+4} \tag{7.2}$$

Looking at Figure 7.17, we can see a plot for the DBPSK receiver memory utilization when GNU Radio default buffer allocation method is used and when the topology matrix based is used. As we can see from the graph, it is possible to implement the same receiver design using lower memory utilization than the default GNU Radio allocation method allowing a CE to adjust the memory utilization according to any memory constraints or latency/throughput constraints the system might have.

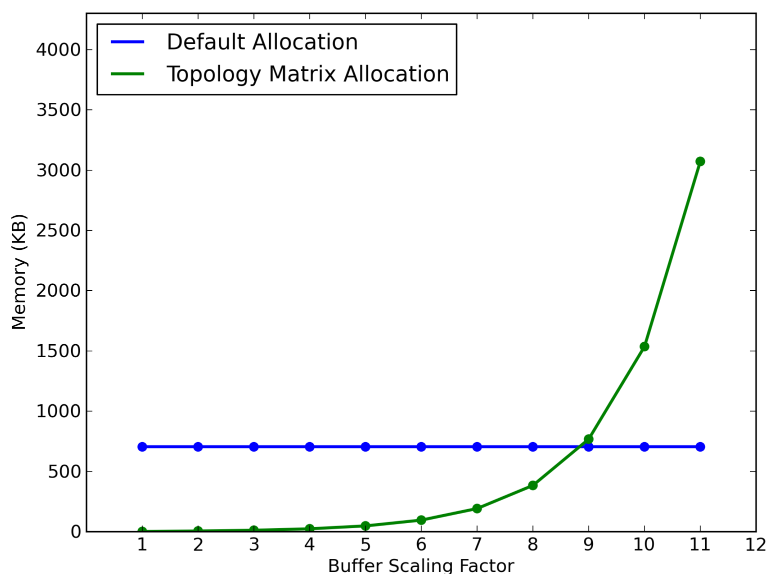

Figure 7.17: Memory Utilization for DBPSK Receiver

Latency

Figure 7.18 shows the DBPSK receiver flowgraph latency using the default and topology based buffer allocation methods. The latency in defined as the sum of the average computation time spent in all of the signal processing blocks in the graph. This basically provides the average time it takes a sample to propagate through the flowgraph from the USRP receiver interface all the way to the data sink block. The bars representing graph standard deviation are small enough where they are not visible in the graph. As we can see from the graph, using the topology matrix based buffer allocation method a CE is able to control the overall latency of the flowgraph depending on the buffer scaling factor that it chooses. This way a CE is able to balance between memory utilization and latency, which is relevant for applications having particular latency requirement.

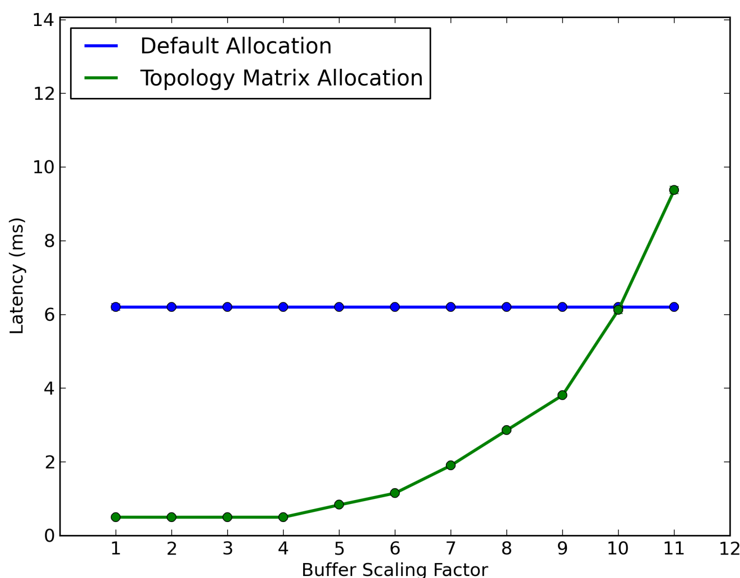

Figure 7.18: Latency for DBPSK Receiver

Throughput

Figure 7.19 shows the DBPSK receiver flowgraph throughput using the default and topology based buffer allocation methods. The receiver flowgraph throughput is defined as the average amount of output data from the DBPSK demodulation block divided by the average amount of time spent processing data in the same block. The bars representing graph standard deviation are small enough where they are not visible in the graph. The throughput is calculated using the demodulation block instead of the packet decoding block because the packet decoding block will potentially drop corrupted data packets. Therefore by looking at the demodulator throughput we are looking at the computational throughput of the graph since in this work we are addressing computational aspects of SDR implementation rather than other issues related to building a high fidelity DBPSK receiver.

As we can see from the figure, a CE is able to control the throughput of the flowgraph by controlling the overall flowgraph memory utilization. By observing throughput and latency a CE is able to control the buffer sizes to balance the memory, throughput, and latency requirement of the application. In addition, a CE is also able to observe the maximum throughput achievable by the receiver design running on the specific hardware platform it is running on. When the same receiver design is run on different hardware platforms, a CE can incorporate its knowledge regarding the maximum achievable throughput when configuring the radio design.

As we can see from the memory utilization graph, Figure 7.17, that as soon memory utilization starts exceeding that of the topology matrix based allocation method that the increase in throughput is not significant enough to where the increase in latency and throughput might be justifiable.

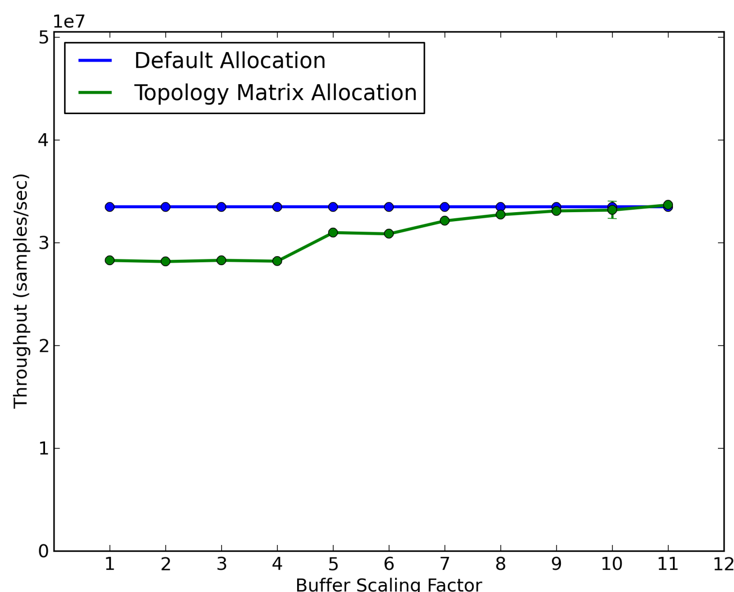

Figure 7.19: Throughput for DBPSK Receiver

Reconfiguration Time

Figure 7.20 shows the reconfiguration times associated with the DBPSK receiver flowgraph. The bars in the graph represent the reconfiguration time standard deviation, for the default GNU Radio buffer allocation method, the standard deviation is plotted for the initial point but it is the same value for the remaining points. We can see that it is difficult to draw a direct correlation between the flowgraph reconfiguration times and the amount of memory allocation for the buffers. As mentioned previously, buffer allocation in GNU Radio is probably not the most time consuming aspect of allocating a flowgraph. In addition cache performance is unpredictable.

It is worth noting that the reconfiguration time for the DBPSK receiver is significantly longer than the for the DBPSK transmitter, (Figure 7.15), and for the DBPSK link, discussed next (Figure 7.25). The DBPSK link includes the same receiver design used here except the receiver gets over-the-air data samples using a USRP. This means that the USRP receiver interface takes longer to set up than the transmit interface. In addition the USRP receive

interface takes longer to setup than the remainder of the flowgraph since the DBPSK link contains more blocks than the receiver design but still takes less time to reconfigure.

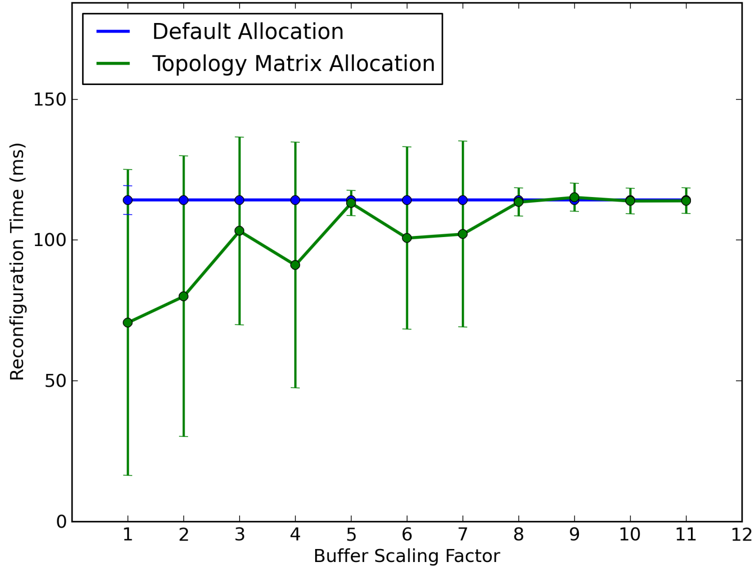

Figure 7.20: Reconfiguration for DBPSK Receiver

7.5.3 DBPSK Link

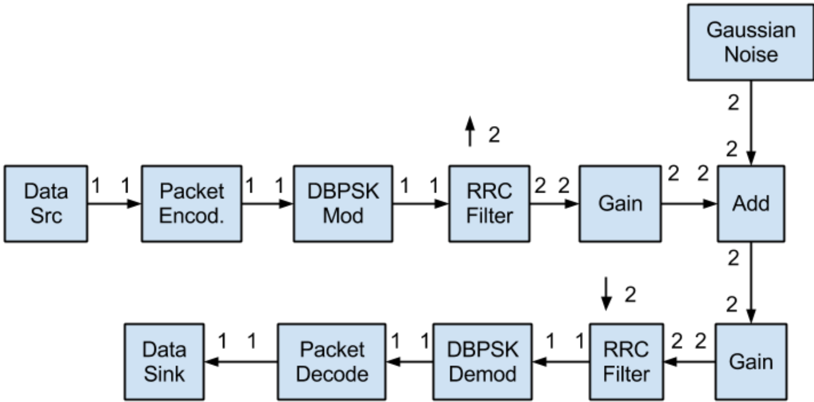

Figure 7.21: DBPSK Link SDF Model

The DBPSK link model used can be seen in Figure 7.21. The first stage topology matrix is as follows:

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

The following is a list of the blocks present in the first stage topology matrix where the block enumeration is the same as the block column index in the topology matrix.

0. Data Source.
1. Packet Encoder.
2. DBPSK Modulator.
3. RRC Filter.
4. Gain1.
5. Add.
6. Gaussian Noise.
7. Gain2.
8. RRC Filter.
9. DBPSK Demodulator.
10. Packet Decoder.
11. Data Sink.

The calculated repetition vector is as follows:

$$
q = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
$$

The generated flowgraph is expanded by GNU Radio generating the following second stage topology matrix:

$$
\Gamma = \left[\begin{array}{cccccccccccccccccccccccccc}
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & -16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & -16 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & -8
\end{array}\right]
$$

The calculated repetition vector for the graph is as follows:

$$q = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The following is the list of blocks in the flowgraph after GNU Radio expands the definition of the hierarchical blocks in the model. Please note that the enumeration for each block corresponds to the column in the topology matrix.

0. file_source0
1. throttle0
2. message_sink0
3. message_source1
4. file_sink0
5. message_source0
6. packed_to_unpacked_bb0
7. map_bb0
8. diff_encoder_bb0
9. chunks_to_symbols_bc0
10. pfb_arb_resampler_ccf0
11. multiply_const_vcc2
12. multiply_const_vcc4
13. noise_source_c0
14. multiply_const_vcc0
15. add_cc0
16. multiply_const_vcc3
17. multiply_const_vcc1
18. gr_agc2_cc0
19. fll_band_edge_cc0
20. pfb_clock_sync_ccf0
21. constellation_receiver_cb0
22. diff_decoder_bb0
23. unpack_k_bits_bb0

24. correlate_access_code_bb0
25. framer_sink_10

We can see that the final relative rate in the DBPSK link's topology matrix is 8 an not 1 as we might expect, since the initial relative rate of the transmit component of the flowgraph is 1. This occurs because the file sink block in GNU Radio perform the final *packing* of individual bits into a single data type, which is the inverse of what the *packed to unpacked* block does.

Memory Utilization

The DBPSK link buffer scaling is varied from $n = [1, 13]$.

$$buffer_scale = 2^n \tag{7.3}$$

Looking at Figure 7.22, we can see the total memory utilization for the DBPSK link flowgraph versus the buffer scaling factor for both the topology matrix buffer allocation method and the default GNU Radio buffer allocation method. We can see the same trend we have seen before, where a CE is able to control the amount of memory utilizing by changing the buffer scaling factors to suit application needs.

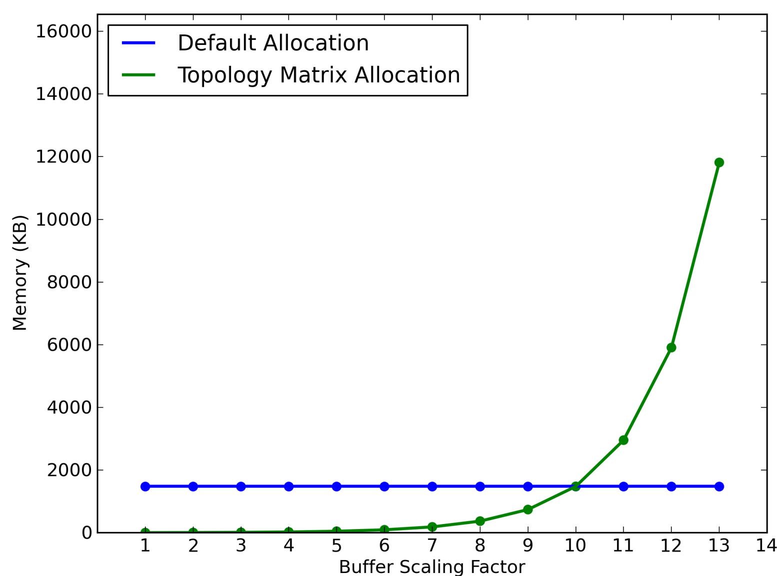

Figure 7.22: Memory Utilization for DBPSK Link

Latency

Looking at Figure 7.23, we can see the latency for the DBPSK link flowgraph versus buffer scaling factors. The bars in the graph represent the reconfiguration time standard deviation, for the default GNU Radio buffer allocation method, the standard deviation is plotted for the initial point but it is the same value for the remaining points. Latency in this flowgraph, is defined as the average processing time for all the blocks except for the Gaussian noise block and its associated gain block. The reason for excluding them is that following the data path from the data source all the way to the data sink we can see that the input data does not pass through these blocks therefore they are not in the effective data path and should be excluded.

We can see that a CE is able to control the latency of the flowgraph by modifying the buffer scaling factor which enables CEs to modify the amount of memory utilized in a flowgraph in order to yield the desired latency performance.

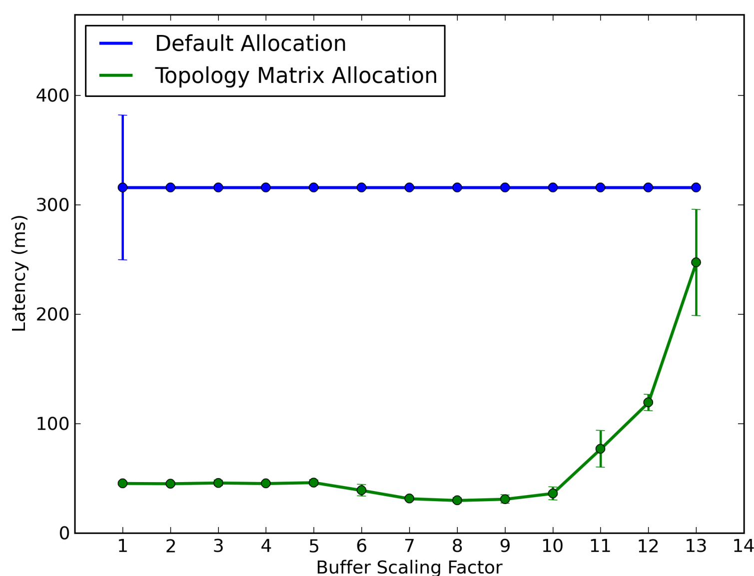

Figure 7.23: Latency for DBPSK Link

Throughput

Figure 7.24 shows the throughput of the DBPSK link flowgraph for the default GNU Radio buffer allocation method and the topology matrix allocation method. The bars representing graph standard deviation are small enough where they are not visible in the graph. The

throughput in the flowgraph is defined as the DBPSK demodulation block throughput, the average amount of data produced by the block divided by the average amount of time the block spends processing data. The reason for observing the demodulator block's throughput is the same as with the receiver flowgraph; we are interested in observing the computational throughput and since the packet decode block ignores corrupted packets it is not being used to monitor throughput.

As we can see from the graph, CEs are able to modify buffer scaling in the flowgraph to modify the throughput. However, the flowgraph throughput reaches an upper limit where it cannot be further improved as seen in Figure 7.23.

Looking at the memory utilization graph, Figure 7.22, we can see that as soon as the memory utilization for the topology matrix allocation method starts exceeding that of the default GNU Radio allocation method that the increase in throughput might not be justifiable considering the increase in memory utilization and latency needed.

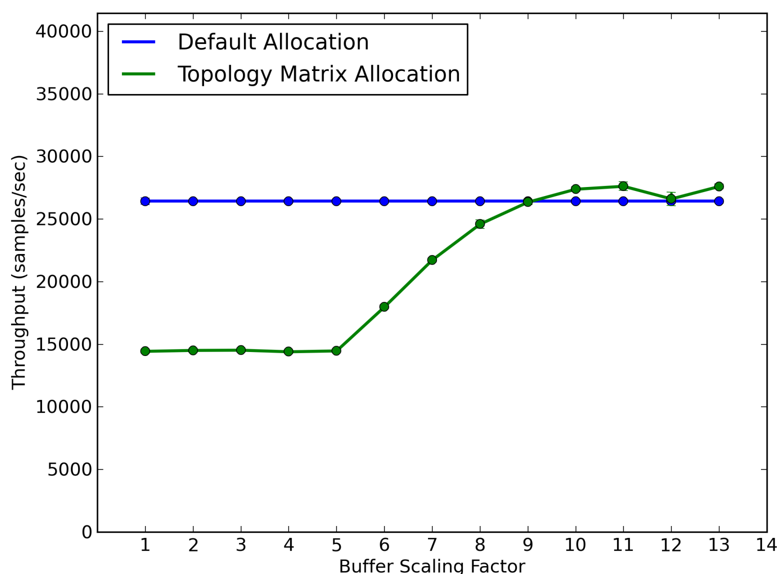

Figure 7.24: Throughput for DBPSK Link

Reconfiguration Time

Figure 7.25 shows the reconfiguration time for the DBPSK link graph. The bars in the graph represent the reconfiguration time standard deviation, for the default GNU Radio buffer allocation method, the standard deviation is plotted for the initial point but it is the same value for the remaining points. We can see again that there is no direct correlation between

the buffer scaling and the reconfiguration time of the graph. The presented reconfiguration graph presents a gauge of how much time a CE should anticipate to spend if it decides to modify the current radio flowgraph therefore causing the radio flowgraph to be torn down and reconstructed.

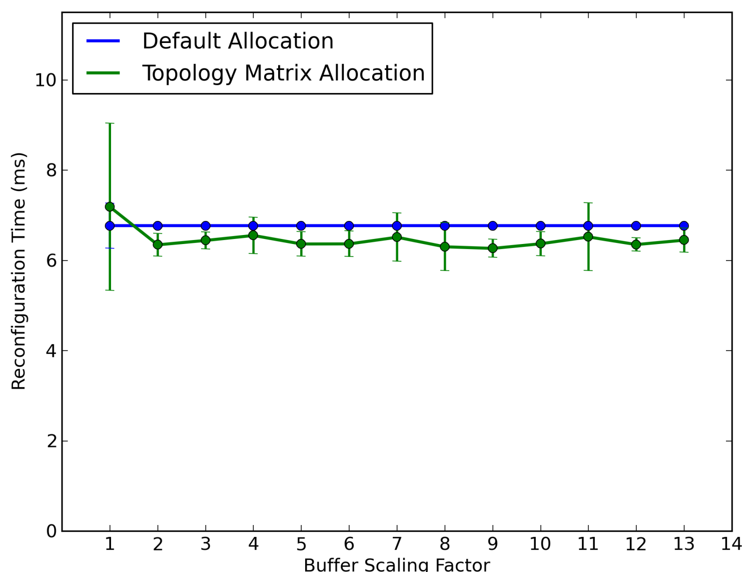

Figure 7.25: Reconfiguration for DBPSK Link

7.6 Concluding Remarks

By having CEs access the buffer scaling knob, they are able to control the amount of throughput and latency in the final SDR system implementation. While it is possible to use arbitrarily large buffer sizes to increase throughput, as is done in GNU Radio, sizing the buffers appropriately can decrease the amount of latency experienced in a radio flowgraph. In addition, by providing reactive to real-time model transformation, CEs are able to control and monitor the knobs and meters without having to be aware of the underlying implementation details associated with the SDR SDF domain.

Chapter 8

Conclusion and Recommendations for Future Work

8.1 Summary

Chapter 1 provides an overview of the model spacing for CEs and SDRs in the context of CR applications. It also discusses how applying a joint reactive and real-time view of CR applications can enable CEs to define overall radio architecture without having to specify the details involved. Dataflow MoC properties can be used in specifying radio design to develop computational knobs and meters for CE to control the memory utilization, throughput, and latency of SDR implementations. The chapter also introduces the motivation for the work presented in the dissertation and gives a big picture view of the work presented.

Chapter 2 provides the literature review relevant to the work presented. It provides the current status of knowledge in regards to implementing SDR and CR systems and discusses the application of MoCs to SDR systems and discusses how the presented work can be used to enhance the current state of knowledge.

Chapter 3 introduces the concept of MoCs and summarizes relevant MoCs for reactive and real-time, dataflow, modeling of SDR and CE applications. The chapter also summarizes the basic principles of the SDF and CSP MoC which are used in the dissertation work presented.

Chapter 4 provides a short introduction of digital communication principles used in defining radio flowgraphs in the dissertation.

Chapter 5 introduces the software and hardware tools used in the dissertation and explains how they are utilized.

Chapter 6 explains the details of the framework developed for transforming reactive radio, as defined by CEs, into dataflow-based models as needed to implement SDR radio flowgraphs.

The chapter also discusses how the framework enables the definition of computational knobs and meters to CEs via the reactive to real-time model transformation.

Chapter 7 presents proof of concept radio applications developed using the proposed framework. The example radio applications discussed demonstrate how CEs can utilize computational knobs, buffer scaling, to monitor and control meter readings, (latency, throughput, and memory utilization), which can be effectively used by CEs to account for application computational needs.

8.2 Conclusion

The dissertation demonstrates how CEs can define radio flowgraphs using reactive models and have those models transformed into real-time, or dataflow, based models which can be used to implement SDRs. By having CEs rely on reactive radio models, they are able to be detached from SDR implementation details and are able to control the computational performance outcome of the SDR implementation by controlling a computational knob, buffer scaling, and observing the outcome, by looking at associated computational meters.

Allowing CE to interact with SDR implementations through computational knobs and meters expands the scope of possible radio configurations explored by CEs which not only provides insight into the computational platforms that such radios are running on, but it can allow CE to yield radio solutions which have better performance. While the scope of *performance* in this dissertation only explores computational throughput, latency, and memory utilization, it provides a new understanding of how computational knobs and meters can be introduced into the CE design space.

The application of MoCs to SDR implementation is not a new concept as discussed in the literature review chapter. However, allowing CEs to inherit computational knobs and meters from the MoC used in defining SDR flowgraphs is a novel contribution. Instead of having SDR schedulers trying to yield good computational *performance*, the proposed work allows CEs to take computational performance into account while exploring appropriate radio solutions which meet application needs and RF channel conditions.

8.3 Contribution

- Using reactive radio flowgraph models for CE perspective.
- Transforming reactive radio models as defined by CEs to real-time, dataflow, based models for the SDR domain.
- Using MoC properties, such as analyzing graph consistency, calculating flowgraph buffer requirement, and buffer scaling to provide CEs with insight into computational

performance and consideration about the underlying implementation of its radio definitions.

- Developing a framework which allows the reactive to real-time model transformation.
- Development of sample applications using the proposed framework and demonstrating performance results in terms of how CEs can make use of the work presented in this dissertation.

8.4 Future Work

In the current work, reactive radio definitions are passed using Occam files which are parsed and interpreted to define the intended radio definitions. It would be interesting to allow CEs to define and run Occam programs and have CEs pass meta-data information via a running Occam program to the framework. This requires the development of an interface between running Occam programs and the framework, similar to how SWIG facilitates Python/C++ communication. By passing meta-data in this dynamic fashion, CEs are able to modify parameters which do not affect the SDR topology, such as gain settings, without needing to write new Occam programs, passing them to the framework, and triggering a complete system reconfiguration.

SDF semantics allow the use of multiple processor so it can be beneficial to extend the work where CEs are able to utilize computational knobs and meters which allows them to observe and control computational performance in a multi-core environment.

It would also be of value to look into more digital communication systems and test how the proposed framework and improve SDR implementations and provide CEs with insight in regards to the underlying systems.

Bibliography

[1] J. Mitola, "Cognitive Radio An Integrated Agent Architecture for Software Defined Radio," Dissertation, Royal Institute of Technology (KTH), 2000.

[2] C. Rieser, T. Rondeau, C. Bostian, and T. Gallagher, "Cognitive radio testbed: further details and testing of a distributed genetic algorithm based cognitive engine for programmable radios," in *Military Communications Conference, 2004. MILCOM 2004. IEEE*, vol. 3, Oct.-3 Nov. 2004, pp. 1437 – 1443.

[3] T. W. Rondeau and C. W. Bostian, *Artificial Intelligence in Wireless Communication (Mobile Communications)*. Norwood, MA, USA: Artech House Publishers, 2009.

[4] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[5] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[6] GNU Radio. [Online]. Available: http://gnuradio.org/redmine/wiki/gnuradio

[7] *Ettus Research USRP E100 Embedded Software Defined Radio*, Ettus Research. [Online]. Available: www.ettus.com/downloads/USRP_E100_Series_temporary_datasheet.pdf

[8] MPRG, "Ossie." [Online]. Available: http://ossie.wireless.vt.edu/

[9] M. Carrick, S. Sayed, C. Dietrick, and J. Reed, "Integration of fpgas into sdr via memory-mapped i/o," *SDR Technical Conference and Product Exposition*, December 2009.

[10] IRIS. [Online]. Available: http://www.softwareradiosystems.com/redmine/projects/iris

[11] P. D. Sutton, J. Lotze, H. Lahlou, S. A. Fahmy, K. E. Nolan, B. Ozgul, T. W. Rondeau, J. Noguera, and L. E. Doyle, "Iris: an architecture for cognitive radio networking testbeds," *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 114–122, 2010.

[12] A. Fayez, Q. Chen, A. Young, N. Kaminski, H. Alavi, and C. Bostian, "An embedded platform based public safety cognitive radio," *Wireless Innovation Forum Conference and Product Exposition*, January 2013.

[13] A. R. Young, "Unified Multi-Domain Decision Making: Cognitive Radio and Autonomous Vehicle Convergence," Dissertation, Virginia Polytechnic Institute and State University, 2012.

[14] J. Wang, M. Ghosh, and K. Challapali, "Emerging cognitive radio applications: A survey," *Communications Magazine, IEEE*, vol. 49, no. 3, pp. 74–81, 2011.

[15] *Ettus Research USRP1 Bus Series*, Ettus Research. [Online]. Available: https://www.ettus.com/content/files/07495_Ettus_USRP1_DS_Flyer_HR.pdf

[16] J. Place, D. Kerr, and D. Schaefer, "Joint tactical radio system," in *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, vol. 1, 2000, pp. 209–213.

[17] J. Mitola III and G. Q. Maguire Jr, "Cognitive radio: making software radios more personal," *Personal Communications, IEEE*, vol. 6, no. 4, pp. 13–18, 1999.

[18] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner, "Spex: A programming language for software defined radio," in *Software Defined Radio Technical Conference and Product Exposition, Orlando*, 2006.

[19] C. R. A. González, C. B. Dietrich, S. Sayed, H. I. Volos, J. D. Gaeddert, P. M. Robert, J. H. Reed, and F. E. Kragh, "Open-source sca-based core framework and rapid development tools enable software-defined radio education and research," *Communications Magazine, IEEE*, vol. 47, no. 10, pp. 48–55, 2009.

[20] G. F. Zaki, W. Plishker, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall, "Integration of dataflow-based heterogeneous multiprocessor scheduling techniques in gnu radio," *Journal of Signal Processing Systems*, vol. 70, no. 2, pp. 177–191, 2013.

[21] S. Mahmood, R. Lai, and Y. S. Kim, "Survey of component-based software development," *Software, IET*, vol. 1, no. 2, pp. 57–66, 2007.

[22] J. Castrillon, S. Schürmans, A. Stulova, W. Sheng, T. Kempf, R. Leupers, G. Ascheid, and H. Meyr, "Component-based waveform development: the nucleus tool flow for efficient and portable software defined radio," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, pp. 173–190, 2011.

[23] J.-P. Delahaye, J. Palicot, and P. Leray, "A hierarchical modeling approach in software defined radio system design," in *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on.* IEEE, 2005, pp. 42–47.

[24] Mathworks. [Online]. Available: http://www.mathworks.com/

[25] ——, “Simulink.” [Online]. Available: http://www.mathworks.com/products/simulink/

[26] ——, “Simulink coder.” [Online]. Available: http://www.mathworks.com/products/simulink-coder/

[27] ——, “Hdl code generation and verification.” [Online]. Available: http://www.mathworks.com/hdl-code-generation-verification/

[28] ——, “Xilinx system generator and hdl coder.” [Online]. Available: http://www.mathworks.com/fpga-design/simulink-with-xilinx-system-generator-for-dsp.html

[29] ——, “expressdsp software development tools.” [Online]. Available: http://www.mathworks.com/dsp/partnerships/ti_new.html

[30] W.-S. Gan, Y.-K. Chong, W. Gong, and W.-T. Tan, “Rapid prototyping system for teaching real-time digital signal processing,” *Education, IEEE Transactions on*, vol. 43, no. 1, pp. 19–24, Feb 2000.

[31] C. Dick, “The platform fpga: Enabling the software defined radio,” in *Software Defined Radio 2002 Technical Conference and Product Exposition*, 2002.

[32] V. Marojevic, X. R. Ballesté, and A. Gelonch, “A computing resource management framework for software-defined radios,” *Computers, IEEE Transactions on*, vol. 57, no. 10, pp. 1399–1412, 2008.

[33] I. Gomez, V. Marojevic, and A. Gelonch, “Automatic computing resource awareness in resource managers for cognitive radios,” in *Cognitive Information Processing (CIP), 2010 2nd International Workshop on*. IEEE, 2010, pp. 122–127.

[34] J. D. Gaddert, “Facilitating Wireless Communication through intelligent Resource Management on Software-Defined Radios in Dynamic Spectrum Environments,” Dissertation, Virginia Polytechnic Institute and State University, 2011.

[35] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, “Soda: A high-performance dsp architecture for software-defined radio,” *IEEE Micro*, vol. 27, pp. 114–123, 2007.

[36] G. K. Rauwerda, P. M. Heysters, and G. J. Smit, “Towards software defined radios using coarse-grained reconfigurable hardware,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 3–13, 2008.

[37] M. Palkovic, P. Raghavan, M. Li, A. Dejonghe, L. Van der Perre, and F. Catthoor, “Future software-defined radio platforms and mapping flows,” *Signal Processing Magazine, IEEE*, vol. 27, no. 2, pp. 22–33, 2010.

[38] U. Ramacher, "Software-defined radio prospects for multistandard mobile phones," *Computer*, vol. 40, no. 10, pp. 62–69, 2007.

[39] H. Berg, C. Brunelli, and U. Lucking, "Analyzing models of computation for software defined radio applications," in *System-on-Chip, 2008. SOC 2008. International Symposium on.* IEEE, 2008, pp. 1–4.

[40] M.-S. David, M. Jose F, C.-P. Jorge, C. Daniel, G. Salvador, C. Narcís *et al.*, "On the way towards fourth-generation mobile: 3gpp lte and lte-advanced," *EURASIP Journal on Wireless Communications and Networking*, 2009.

[41] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal, "Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications," in *System on Chip (SoC), 2011 International Symposium on.* IEEE, 2011, pp. 14–21.

[42] B. D. Theelen, M. Geilen, T. Basten, J. Voeten, S. V. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. Fourth ACM and IEEE International Conference on.* IEEE, 2006, pp. 185–194.

[43] C.-J. Hsu, J. L. Pino, and F.-J. Hu, "A mixed-mode vector-based dataflow approach for modeling and simulating lte physical layer," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE.* IEEE, 2010, pp. 18–23.

[44] G. Zaki, W. Plishker, T. Oshea, N. McCarthy, C. Clancy, E. Blossom, and S. Bhattacharyya, "Integration of dataflow optimization techniques into a software radio design framework," in *Signals, Systems and Computers, 2009 Conference Record of the Forty-Third Asilomar Conference on*, 2009, pp. 243–247.

[45] NVIDIA, "Cuda gpus." [Online]. Available: https://developer.nvidia.com/cuda-gpus

[46] C.-J. Hsu, M.-Y. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the 2005 workshop on Software and compilers for embedded systems*, ser. SCOPES '05. New York, NY, USA: ACM, 2005, pp. 37–49. [Online]. Available: http://doi.acm.org/10.1145/1140389.1140394

[47] C.-C. Shen, W. Plishker, H.-H. Wu, and S. S. Bhattacharyya, "A lightweight dataflow approach for design and implementation of sdr systems," in *Proceedings of the Wireless Innovation Conference and Product Exposition*, 2010, pp. 640–645.

[48] Y. Lin, M. Kudlur, S. Mahlke, and T. Mudge, "Hierarchical coarse-grained stream compilation for software defined radio," in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems.* ACM, 2007, pp. 115–124.

[49] M. Palkovic, J. Declerck, P. Avasare, M. Glassee, A. Dewilde, P. Raghavan, A. Dejonghe, and L. Van der Perre, “Dart - a high level software-defined radio platform model for developing the run-time controller,” *Journal of Signal Processing Systems*, vol. 69, no. 3, pp. 317–327, 2012.

[50] E. Willink and C. Waugh, “Waveform description language,” Thales Research Report, P6957-ll-0l4, Tech. Rep., 2000.

[51] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[52] N. A. Stanton and P. Marsden, “From fly-by-wire to drive-by-wire: safety implications of automation in vehicles,” *Safety Science*, vol. 24, no. 1, pp. 35–49, 1996.

[53] D. B. West, *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, Aug. 2000. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0130144002

[54] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[55] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, “Data-driven and demand-driven computer architecture,” *ACM Computing Surveys (CSUR)*, vol. 14, no. 1, pp. 93–143, 1982.

[56] G. Kahn and D. B. MacQueen, “Coroutines and networks of parallel processes,” in *IFIP Congress*, 1977, pp. 993–998.

[57] E. Lee and D. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *Computers, IEEE Transactions on*, vol. C-36, no. 1, pp. 24–35, 1987.

[58] J. T. Buck and E. A. Lee, “Scheduling dynamic dataflow graphs with bounded memory using the token flow model,” in *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, vol. 1. IEEE, 1993, pp. 429–432.

[59] J. T. Buck, “Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams,” in *Signals, Systems and Computers. Conference Record of the Twenty-Eighth Asilomar Conference on*, vol. 1. IEEE, 1994, pp. 508–513.

[60] M. Engels, G. Bilson, R. Lauwereins, and J. Peperstraete, “Cycle-static dataflow: model and implementation,” in *Signals, Systems and Computers. Conference Record of the Twenty-Eighth Asilomar Conference on*, vol. 1. IEEE, 1994, pp. 503–507.

[61] P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete, “Cyclo-dynamic dataflow,” in *Parallel and Distributed Processing, 1996. PDP’96. Proceedings of the Fourth Euromicro Workshop on.* IEEE, 1996, pp. 319–326.

[62] B. Bhattacharya and S. S. Bhattacharyya, “Parameterized dataflow modeling for dsp systems,” *Signal Processing, IEEE Transactions on*, vol. 49, no. 10, pp. 2408–2421, 2001.

[63] S. Neuendorffer and E. Lee, “Hierarchical reconfiguration of dataflow models,” in *Formal Methods and Models for Co-Design, 2004. MEMOCODE ’04. Proceedings. Second ACM and IEEE International Conference on*, 2004, pp. 179–188.

[64] J. C. Baeten, “A brief history of process algebra,” *Theoretical Computer Science*, vol. 335, no. 2, pp. 131–146, 2005.

[65] C. A. R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[66] ——, *Communicating sequential processes.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[67] R. Milner, *A calculus of communicating systems.* Springer-Verlag New York, Inc., 1982.

[68] ——, *Communicating and mobile systems: the pi calculus.* Cambridge university press, 1999.

[69] W. Stallings, *Wireless Communications and Networks.* Prentice Hall, 2002.

[70] S. Haykin and M. Moher, *Introduction to Analog and Digital Communications Second Edition.* Hoboken, NJ, USA: John Wiley and Sons, Inc., 2007.

[71] A. W. Roscoe, “Denotational semantics for occam,” in *Seminar on concurrency.* Springer, 1985, pp. 306–329.

[72] “Kroc.” [Online]. Available: http://www.cs.kent.ac.uk/projects/ofa/kroc/

[73] J. Moores, “Ccsp–a portable csp-based run-time system supporting c and occam,” *Architectures, Languages and Techniques for Concurrent Systems*, vol. 57, pp. 147–168, 1999.

[74] N. C. Brown and P. H. Welch, “An introduction to the kent c++ csp library,” *Communicating process architectures*, vol. 2003, pp. 139–156, 2003.

[75] “Java communicating sequential processes.” [Online]. Available: http://www.cs.ukc.ac.uk/projects/ofa/jcsp/

[76] C. Whitby-Strevens, “The transputer,” in *ACM SIGARCH Computer Architecture News*, vol. 13, no. 3. IEEE Computer Society Press, 1985, pp. 292–300.

[77] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, “Taming heterogeneity-the ptolemy approach,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[78] A. Gorokhov, D. A. Gore, and A. J. Paulraj, "Receive antenna selection for mimo spatial multiplexing: theory and algorithms," *Signal Processing, IEEE Transactions on*, vol. 51, no. 11, pp. 2796–2807, 2003.

[79] "Universal software radio peripheral." [Online]. Available: http://www.ettus.com

[80] Ettus Research. [Online]. Available: https://www.ettus.com/product/details/WBX

[81] "Gnu linear programming kit." [Online]. Available: http://www.gnu.org/software/glpk/

[82] "Pyglpk website." [Online]. Available: http://tfinley.net/software/pyglpk/

[83] "matplotlib." [Online]. Available: http://matplotlib.org/

[84] A. He, K. K. Bae, T. Newman, J. Gaeddert, K. Kim, R. Menon, L. Morales-Tirado, J. Neel, Y. Zhao, J. Reed, and W. Tranter, "A survey of artificial intelligence for cognitive radios," *Vehicular Technology, IEEE Transactions on*, vol. 59, no. 4, pp. 1578–1592, 2010.

[85] E. A. Lee, "Cyber-physical systems-are computing foundations adequate," in *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, vol. 2. Citeseer, 2006.

[86] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.