



Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

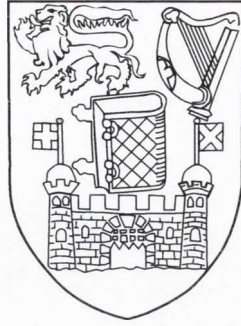
Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.



Enabling Cognitive Radio Experimental Research

Jörg Lotze

A thesis submitted to the University of Dublin
for the degree of Doctor of Philosophy

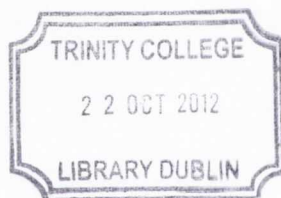
Department of Electronic and Electrical Engineering
University of Dublin, Trinity College

August 2011

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.



Thesis 9758

Summary

Cognitive radio technology is a promising solution for addressing the increasing demand in reliability, connectivity, bandwidth, and quality of future wireless communication systems. This is achieved through continuously and intelligently adapting the radio's operating parameters to current conditions and learning from the past. However, there are many open research questions in this area. Experimental research will play an important role if these questions are to be answered in a manner that convinces the sceptics of the potential of cognitive radio technology.

This thesis proposes methods for enabling and simplifying experimental research in the field of cognitive radio. It presents the cognitive radio development framework, Iris-FPGA, as an effective solution for testing theories and algorithms in real-world scenarios. Further, it proposes a model-based cognitive radio design technique that significantly simplifies the development process of radio prototypes for experimentation.

The framework, Iris-FPGA, is built on a field-programmable gate array (FPGA)-based platform with the software part executing on an embedded general purpose processor (GPP), combining high flexibility with high performance. The software part is based on Iris, a cognitive radio framework developed and maintained at Trinity College Dublin. Cognitive radio prototypes in Iris-FPGA are constructed based on a signal flow diagram of processing components and a radio controller responsible for cognitive decision-making. The components can be implemented in software or FPGA hardware and combined arbitrarily within a radio. Iris-FPGA supports run-time reconfiguration, whether the portion to be reconfigured is implemented in hardware or software. It makes it easy to exploit the processing power of FPGAs, thus, simplifying the development of cognitive radio prototypes for experimentation. A number of realistic radio demonstrators are developed to illustrate the capabilities of the framework.

The experience from developing the demonstrator radios is then compared to radio design approaches taken by a number of existing cognitive radio implementation frameworks and

hardware platforms. It is found that designing cognitive radios is generally ad-hoc, as every framework follows a different approach, which generally requires detailed platform knowledge. This makes cognitive radio experimental research a cumbersome task.

A novel model-based design technique is proposed as a solution to simplify the development of cognitive radio prototypes for experimentation. This allows platform-independent design of radios, abstracting low level details and focusing on the system composition and dynamic behaviour. The underlying model is the first that fully captures the reconfigurability of cognitive radios. It includes the radio composition from functional components, the radio reconfiguration options, and the interface to the cognitive engine. Synthesis steps for automatically generating real implementations for arbitrary target platforms from a model are outlined in the thesis. A case study, developing a radio from the idea to a working implementation, demonstrates the advantages of the proposed cognitive radio design technique by comparing with traditional methods. It becomes apparent that the proposed technique simplifies the development of cognitive radio prototypes for experimentation significantly.

While the model-based design technique is designed to be used on any cognitive radio implementation framework and hardware platform, i.e., it is platform-independent, the more systematic approaches and the theoretical foundation can also influence the design of the implementation frameworks themselves. These can be made more flexible and more efficient by applying techniques from the proposed model-based design approach. This is illustrated on the example of Iris2-FPGA, an improved version of Iris-FPGA.

By enabling and simplifying systematic experimental research with cognitive radios, this thesis represents a significant step towards answering many of the open research questions in the field of cognitive radio. It further places a stronger onus on those conducting research in the area to validate their ideas experimentally. This is essential if cognitive radio is to mature as a technology.

Acknowledgements

This thesis is the product of years of work with CTVR, Trinity College Dublin, in partnership with the Xilinx Research Labs. Many people deserve my gratitude. First and foremost, I would like to thank my supervisor, Linda Doyle, for her constant support, limitless encouragement, constructive criticism, and infectious enthusiasm. She shaped my professional identity and helped making me the researcher I am today.

I would like to thank all members of the CTVR Emerging Networks group for numerous interesting discussions. They made my time at Trinity most enjoyable. Special thanks to Suhaib Fahmy, for the fruitful time spent working together and for sharing his knowledge and experience with me. Further, I would like to thank Paul Sutton, Keith Nolan, Barış Özgül, Tim Forde, Andrew Cunningham, and Hicham Lahlou for their guidance and suggestions. I also would like to mention those who spent time reading my thesis, always giving constructive feedback.

Thanks also to the members of the Xilinx Research Labs, especially to Juanjo Noguera and Robert Esser. Without their tireless support, this thesis would not have been possible. I enjoyed working with this experienced team of industrial researchers, which helped me to understand the industry's view on the subjects covered.

Thanks to my family for their continuous help and praise and to my friends for providing the much needed distractions. A special thanks is devoted to my parents, Dagmar and Helmut, for teaching me the importance of ambition and self confidence, which made this thesis possible. Their respective partners, Jochen and Birka, also deserve my gratitude for shaping my personality and teaching me important lessons in life.

A huge thanks to Katja Vogler, for pushing me to make the vital phone call that opened the opportunity to pursue a PhD at Trinity. I thank her for her continuous support, for helping me through difficult phases, for being patient with me, and especially for the tremendous love and joy she brings into my life. I would like to mention my six-month-old daughter Phoebe. Her smiles and giggles provided the much-needed balance to the tough world of research, filling

my heart with pride and happiness every day.

Finally, I would like to thank Enterprise Ireland for funding the fruitful collaboration with Xilinx under its Innovation Partnership scheme, project IP20060367, and Science Foundation Ireland for funding CTVR under Grant No. 03/CE3/I405.

Contents

Introduction	1
The Concept of Cognitive Radio	2
On the Importance of Experimentation	3
Design of Cognitive Radios for Experimentation	4
Contributions	4
Thesis Outline	6
Publications Record	7
 Part I Cognitive Radio Experiments	 9
 1 Existing Cognitive and Software-Defined Radio Frameworks	 11
1.1 Joint Tactical Radio System: Software Communications Architecture	13
1.1.1 SCA Overview	13
1.1.2 Open Source SCA Implementation::Embedded (OSSIE)	15
1.1.3 Other Frameworks Implementing The JTRS SCA	16
1.1.4 Discussion	17
1.2 GNU Radio	18
1.2.1 Architecture	18
1.2.2 Radio Design	19
1.2.3 Discussion	20
1.3 The Iris Cognitive Radio Framework	21
1.3.1 Architecture	22
1.3.2 Radio Design	26
1.3.3 Discussion	27
1.4 Kansas University Agile Radio (KUAR)	27
1.4.1 Architecture	27

- 1.4.2 Radio Design 29
 - 1.4.3 Discussion 30
- 1.5 Rutgers University WiNC2R 31
- 1.6 Virginia Tech Stalion Processor 31
- 1.7 Rice University WARP 32
- 1.8 Summary 33
- 2 An FPGA-based Cognitive Radio Framework 37**
 - 2.1 Hardware Platforms for Cognitive Radio Experimentation 39
 - 2.2 Iris-FPGA 40
 - 2.2.1 Requirements 41
 - 2.2.2 Virtual Architecture 46
 - 2.2.3 Full System Implementation 50
 - 2.2.4 Abstracting Low-Level Details 57
 - 2.3 Summary 60
- 3 Experiments with the FPGA-based Framework 63**
 - 3.1 Simple Audio Transmitter Design 65
 - 3.1.1 Experimental Setup 65
 - 3.1.2 Radio Configuration 66
 - 3.1.3 Summary and Discussion 68
 - 3.2 Video Streaming Link with Spectrum Sensing 69
 - 3.2.1 Sensing Algorithm 70
 - 3.2.2 Experimental Setup 72
 - 3.2.3 Radio Configuration 74
 - 3.2.4 Summary and Discussion 77
 - 3.3 Video Streaming Link with Adaptive Coding 79
 - 3.3.1 Experimental Setup 80
 - 3.3.2 Radio Configuration 80
 - 3.3.3 Summary and Discussion 84
 - 3.4 Combining Adaptive Coding and Spectrum Sensing 84
 - 3.4.1 Moving to a New FPGA Board 85
 - 3.4.2 Experimental Setup 87
 - 3.4.3 Radio Configuration 88
 - 3.4.4 Summary and Discussion 90
 - 3.5 Summary 91

4 Observations and Analysis	93
4.1 Iris-FPGA: Reflections	93
4.1.1 Strengths	94
4.1.2 Weaknesses	99
4.2 Approaches to Cognitive Radio Design: An Analysis	105
4.2.1 Level of Abstraction and Model of Computation	105
4.2.2 What Application Domain is Targeted?	107
4.2.3 Ability to Cope with Reconfiguration During Design	108
4.2.4 Platform Summary	108
 Part II Model-based Cognitive Radio Design	 111
5 Model-based Design: Background	113
5.1 Introduction and Terminology	114
5.1.1 Models of Computation	115
5.1.2 Dataflow Graphs	116
5.1.3 Control Structures	118
5.2 Existing Dataflow Models of Computation	118
5.2.1 Process Networks	119
5.2.2 Synchronous Dataflow	122
5.2.3 Scalable Synchronous Dataflow	129
5.2.4 Cyclo-Static Dataflow	129
5.2.5 Boolean Dataflow	131
5.2.6 Core Functional Dataflow	134
5.2.7 Relation between Dataflow MoCs	135
5.3 Existing Control Flow Models of Computation	136
5.3.1 Publish and Subscribe	136
5.3.2 Finite State Machines	137
5.3.3 Petri Nets	140
5.4 Summary	141

- 6 A Model-based Approach to Cognitive Radio Design 143**
 - 6.1 A Model for Cognitive Radios 144
 - 6.1.1 Data Plane 145
 - 6.1.2 Reconfiguration 148
 - 6.1.3 Control Plane 149
 - 6.1.4 Model Correctness 151
 - 6.1.5 Model Transformation 158
 - 6.1.6 Summary 160
 - 6.2 Model-based Cognitive Radio Design 161
 - 6.2.1 General Design Flow 161
 - 6.2.2 Synthesis Considerations 163
 - 6.2.3 Synthesis Steps 169
 - 6.2.4 Limitations 169
 - 6.2.5 Summary 171
 - 6.3 Implications for Cognitive Radio Frameworks on the Example of Iris-FPGA . . 172
 - 6.3.1 Flexible Dataflow Graphs 173
 - 6.3.2 Iris Engines 174
 - 6.3.3 Component Hierarchy 176
 - 6.3.4 Integrating FPGA Hardware 177
 - 6.3.5 Discussion 178
 - 6.4 Summary 178
- 7 Case Study 181**
 - 7.1 Case Study Description 182
 - 7.2 Model-based Radio Design 184
 - 7.2.1 Sketch Radio Model 184
 - 7.2.2 Map Components to Library 189
 - 7.2.3 Write Control Specification 192
 - 7.2.4 Run Model-checking 195
 - 7.2.5 Summary 209
 - 7.3 Radio Implementation 210
 - 7.3.1 Implement Missing Components 210
 - 7.3.2 Implement Control Plane 211
 - 7.3.3 Run Synthesis Tool 213

7.4 Analysis and Discussion	221
7.4.1 Comparison with traditional design	221
7.4.2 Limitations	224
7.4.3 Summary	225
Conclusions and Future Work	227
Summary of Contributions	227
Future Work	232
Conclusion	237
A The Iris XML Format	241
B Hardware Platforms for Cognitive Radio Testbeds	247
B.1 ASIC and ASSP	247
B.2 Microprocessors	248
B.2.1 General purpose processor (GPP)	250
B.2.2 Digital signal processor (DSP)	250
B.2.3 Graphics processing unit (GPU)	251
B.2.4 Cell Broadband Engine (CellBE)	252
B.3 FPGA	254
B.4 Summary	255
C Design Details of the Iris-FPGA Experiments	257
C.1 Simple Audio Transmitter Setup	257
C.2 Video Streaming Link with Spectrum Sensing	260
C.3 Video Streaming Link with Adaptive Coding	261
C.4 Combined Adaptive Coding and Spectrum Sensing	264
D Kahn Process Networks: Continuous and Monotonic Functions	267
E The Iris2-FPGA XML Format	269

List of Figures

1.1	The Iris architecture.	23
1.2	The Iris design flow.	26
1.3	The KUAR software architecture.	29
2.1	Components of a typical radio.	38
2.2	Virtual architecture for FPGA-based platform.	46
2.3	The Iris-FPGA hardware component interface.	52
2.4	Illustration of the workaround to connect the USRP front-end to the FPGA board.	57
2.5	The software wrapper concept.	58
2.6	XSLT hardware composition tool.	60
3.1	Setup for audio transmission experiment.	66
3.2	Radio configurations for the audio transmission demonstrator.	67
3.3	Sensing algorithm to find the transmission signal.	71
3.4	Carrier frequency acquisition mechanism.	72
3.5	Setup of spectrum sensing rendezvous experiment.	73
3.6	Transmitter configuration for the sensing rendezvous experiment.	74
3.7	Receiver configuration for sensing rendezvous experiment.	76
3.8	Sensing demonstration at DySPAN 2008.	79
3.9	Setup of the adaptive coding demonstrator.	80
3.10	Transmitter and receiver configurations for the adaptive coding experiment.	81
3.11	BER vs. SNR for different codes, and an example code switching sequence.	82
3.12	Setup of the combined sensing and coding experiment.	88
3.13	Receiver configuration for the combined adaptive coding and sensing experiment.	89
3.14	Controller state machine for combined adaptive coding and sensing experiment.	90
3.15	Setup of the sensing and adaptive coding demonstrator at SIGCOMM 2009.	90

5.1	A dataflow graph with actors A, B, C, D. Channels are represented by directed arcs.	116
5.2	Actor with input vector $\mathbf{i}$, output vector $\mathbf{o}$, and actor function f .	117
5.3	States of an actor.	117
5.4	An actor with state.	121
5.5	An example SDF graph	123
5.6	Graph from Figure 5.5 with numbered channels.	125
5.7	A 4:1 downsampler in SDF.	130
5.8	A 4:1 downsampler in CSDF.	130
5.9	The BDF switch and select actors	132
5.10	If-then-else graph in boolean dataflow.	132
5.11	The switch actor in CFDF.	134
5.12	Relation between dataflow MoCs.	135
5.13	The state transition diagram of an example FSM.	138
5.14	UML statechart example.	139
5.15	An example Petri net for a sensing transceiver.	140
6.1	A model for cognitive radios.	147
6.2	An SDF graph with deadlock.	152
6.3	An inconsistent SDF graph.	152
6.4	Model transformation relationships.	159
6.5	Proposed design flow for model-based cognitive radio design.	162
6.6	Transforming a subsystem into a flat dataflow graph using switch and select components.	167
6.7	Example transformation from multiple levels to one level of hierarchy.	168
6.8	Application of model-based design technique for arbitrary target platforms.	172
6.9	Example flow graphs not realisable in Iris-FPGA.	173
6.10	The component class hierarchy of Iris2-FPGA.	177
7.1	Proposed design flow for model-based cognitive radio design.	184
7.2	The radio model of the transmitter.	186
7.3	Radio model for the receiver.	188
7.4	Controller statechart for the receiver radio.	189
7.5	The radio model of the transmitter, mapped to the component library.	191
7.6	The radio model of the receiver, mapped to the component library.	193
7.7	Transmitter data plane with annotated link data types.	198

7.8	Receiver data plane with annotated link data types.	199
7.9	Flattened SDF flow graphs for the transmitter radio.	202
7.10	Flattened SDF flow graphs for the receiver radio.	206
7.11	Viterbi decoder with BER estimation (erroneous SDF graph).	208
7.12	Transmitter model transformed to suit the target platform.	215
7.13	Receiver model transformed to suit the target platform.	216
7.14	Abstraction layer for the radio controller.	218
7.15	The transmitter XML configuration.	219
7.16	The receiver XML configuration.	220
A.1	Example Iris XML description.	241
A.2	Iris XML structure with branching.	243
A.3	Iris XML structure with branching and merging.	244
A.4	The Iris XML syntax.	245
B.1	The Cell Broadband Engine architecture.	252
C.1	The CPSS for the audio transmission experiment.	258
C.2	Measured FPGA power consumption of adaptive coding experiment.	263
E.1	Example Iris2-FPGA XML file with branching and merging.	270
E.2	The Iris 2 XML syntax.	271

List of Tables

1.1 Strengths and weaknesses of JTRS SCA.	34
1.2 Strengths and weaknesses of GNU Radio.	34
1.3 Strengths and weaknesses of Iris.	34
1.4 Strengths and weaknesses of KUAR.	35
1.5 Strengths and weaknesses of WiNC2R.	35
1.6 Strengths and weaknesses of the Stalion.	35
1.7 Strengths and weaknesses of WARP.	35
2.1 Mapping of virtual architecture elements on XUP and ML507 boards.	55
3.1 Concepts demonstrated by the different experiments.	91
4.1 Frameworks categorised by level of abstraction during radio design.	105
4.2 Frameworks supporting real-time hardware.	107
4.3 Frameworks categorised by application domain.	107
4.4 Feature summary of the discussed platforms.	109
5.1 The switch actor in CFDE.	134
7.1 Component information for the convolutional encoder.	190
7.2 Component information for the Viterbi decoder.	192
7.3 Control specification for the transmitter radio.	194
7.4 Control specification for the receiver radio.	195
7.5 Component parameters and checks for the transmitter chain.	196
7.6 Component parameters and checks for the receiver chain.	197
7.7 Comparison of traditional design and the proposed mode-based design for cog- nitive radios.	221

B.1 Comparison of processing platforms for cognitive radios. 255

C.1 FPGA utilisation for the audio transmission experiment. 259

C.2 FPGA utilisation for the sensing rendezvous experiment. 261

C.3 FPGA resource utilisation for the adaptive coding experiment. 262

C.4 Performance of adaptive coding experiment. 262

C.5 FPGA resource utilisation for the combined adaptive coding and sensing experiment. 265

List of Acronyms

3D	3-dimensional
ADC	analog-to-digital converter
ALU	arithmetic logical unit
API	application programming interface
ASIC	application specific integrated circuit
ASSP	application specific standard part
BER	bit error rate
BDF	boolean dataflow
BPSK	binary phase shift keying
CDMA	code division multiple access
CFDF	core functional dataflow
CellBE	Cell Broadband Engine
CORBA	Common Object Request Broker Architecture
CPSS	customisable processing subsystem
CPU	central processing unit
CR	cognitive radio
CSDF	cyclo-static dataflow
DAC	digital-to-analog converter
DBPSK	differential binary phase shift keying
DDC	digital down converter
DDF	dynamic dataflow
DDR	double data rate
DFT	discrete Fourier transform
DMA	direct memory access
DQPSK	differential quadrature phase shift keying
DSA	dynamic spectrum access
DSP	digital signal processing
DUC	digital up converter
EIB	Element Interconnect Bus

FER	frame error rate
FIR	finite impulse response
FEC	forward error correction
FFT	fast Fourier transform, a fast implementation of the DFT
FIFO	first in, first out
FPGA	field-programmable gate array
FSM	finite state machine
GPL	GNU General Public License
GPP	general purpose processor
GPU	graphics processing unit
GUI	graphical user interface
HDL	hardware description language
IC	integrated circuit
ICAP	Internal Configuration Access Port
IDL	Interface Description Language
IF	intermediate frequency
IOIF	Input/Output Interface
ISM	industrial, scientific, and medical, a license-free frequency band
JNI	Java Native Interface
JTRS	Joint Tactical Radio System
KPN	Kahn process network
KUAR	Kansas University Agile Radio
MAC	medium access control
MIC	Memory Interface Controller
MMU	memory management unit
MoC	model of computation
OEF	OSSIE Eclipse Feature
OFDM	orthogonal frequency division multiplex
OFDMA	orthogonal frequency division multiple access
OPB	On-Chip Peripheral Bus
OpenCL	Open Computing Language
OSSIE	Open Source SCA Implementation::Embedded
OWD	OSSIE Waveform Developer
PCI	Peripheral Component Interconnect
PCM	pulse code modulation
PHY	physical layer
PLB	Processor Local Bus
PLL	phase-locked loop

PN	process network
POSIX	Portable Operating System Interface
PPE	PowerPC Processor Element
PS	publish & subscribe
PSD	power spectral density
PSS	processor subsystem
QAM	quadrature amplitude modulation
QPSK	quadrature phase shift keying
RAM	random access memory
RF	radio frequency
ROM	read-only memory
Rx	receiver
SCA	Software Communications Architecture
SDF	synchronous dataflow
SDK	software development kit
SDR	software defined radio
SHP	specialised hardware processor
SIMD	single instruction, multiple data
SNR	signal to noise ratio
SoC	system-on-a-chip
SPE	Synergistic Processing Element
SRAM	static random access memory
SSDF	scalable synchronous dataflow
TCP	Transmission Control Protocol
Tx	transmitter
UDP	User Datagram Protocol
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VGA	Video Graphics Array
WARP	Wireless Open-Access Research Platform
WiMAX	Worldwide Interoperability for Microwave Access
WiNC2R	WINLAB Network Centric Cognitive Radio
WINLAB	Wireless Information Network LABoratory
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
XUP	Xilinx University Programme

Introduction

Contents

The Concept of Cognitive Radio	2
On the Importance of Experimentation	3
Design of Cognitive Radios for Experimentation	4
Contributions	4
Thesis Outline	6
Publications Record	7

Albert Einstein, when asked to describe radio, once said:

You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: You send signals here, they receive them there. The only difference is that there is no cat.

There are two possibilities to extend that description to cognitive radio: “*cognitive radio* ensures you hear the meow, no matter what the circumstances are,” or “*cognitive radio* can make the cat talk like a human.” In other words, “it can make radios better” or “it can make better radios.” However, cognitive radio technology is still a young research field and many open questions still need to be answered before it can become reality.

This thesis contributes to the field of cognitive radio through contributions that better enable experimental research. It attempts to answer the question: “How can cognitive radio experimentation be simplified for researchers, so that algorithms and ideas can be tested in the

real world?” It proposes a framework for developing cognitive radio systems for experimentation, identifies key challenges, and proposes a novel, simpler radio design methodology. This allows researchers to test their algorithms in the real world with considerably reduced effort. It can drive the research area forward and bring cognitive radio technology closer to reality.

The Concept of Cognitive Radio

Joseph Mitola coined the term *cognitive radio* in 1999 [1]. Further, he defines different levels of cognitive radios in his thesis [2]. Today, many different, sometimes contradicting definitions can be found in the literature. The IEEE Standard 1900.1 gives the following definition:

A type of radio in which communication systems are aware of their environment and internal state and can make decisions about their radio operating behaviour based on that information and predefined objectives. [3]

In other words, it is a radio that can intelligently adapt itself to the current conditions in order to achieve an objective. Possible objectives could be, for instance, to maximise the link quality or to minimise the power consumption.

In this thesis, cognitive radios are viewed as physical layer (PHY) radios with *meters* and *knobs*, following a definition commonly used by Virginia Tech (e.g., [4]). The meters are used to obtain information about the state of the radio and the environmental conditions. Example meters are the current bit error rate (BER), frame error rate (FER), or signal to noise ratio (SNR). The knobs can be adjusted to tune the radio’s behaviour. Examples are the applied modulation scheme, coding scheme, or transmission frequency.

The entity reading the meters and adjusting the knobs is commonly called *cognitive engine*. It executes the *cognitive cycle* of *observe*, *decide*, and *act* (a simplified version of the cycle introduced in [1]). During the *observe* stage, it reads the meters of the radio to become aware of the current state of the radio and environment. This information is used in the *decide* stage, to decide which actions to take in order to achieve some predefined objective, e.g., to maximise the throughput. Optionally, this stage might include machine learning, enabling the cognitive engine to learn from the past in order to make better decisions in future. During the *act* stage, the decided actions are affected on the radio, changing its behaviour and ideally achieving the objective. After that, the cognitive engine goes back into the observe stage and continues the cognitive cycle.

As opposed to cognitive radios, *cognitive networks* [5] involve the full network protocol stack from the physical (PHY) layer and medium access control (MAC) layer up to the application. This thesis is mainly concerned with PHY layer aspects.

A cognitive radio is typically built upon a software defined radio (SDR) platform, i.e., a radio platform where most functionality can be (re-)programmed. This is in contrast to traditional hardware radios, where the radio's function cannot be changed after it is manufactured. SDR is a solution to enable the reconfigurations required to achieve the objective of a cognitive engine, i.e., it provides a large number of knobs.

As suggested in the beginning, cognitive radio technology has two possible uses: 1) it can make existing radios work better, or 2) it can make new and better radios. The former can be achieved, for example, by allowing a radio to constantly connect to the best wireless access technology available, possibly switching between the IEEE 802.11 WiFi [6], UMTS [7], and WiMAX [8] standards. As an example for the second option, radios that opportunistically use unoccupied spectrum and adapt their operating frequency to changes in the spectrum usage are made possible by cognitive radio technology (this is an example of dynamic spectrum access (DSA) [3, 9]).

On the Importance of Experimentation

Often experiments are regarded as a secondary scientific method and theories or analytical studies are preferred. However, experimentation is one of the most fundamental principles in science; the ultimate test of all knowledge is experiment. It provides proof for theories and analytical studies.

In engineering, there are two types of experimentation: simulations on a model of the reality, or experiments in the real world. Simulations typically require significantly less effort than real-world experiments but there is a danger that the used model neglects important aspects. Then the simulation results are in no relation to real experimental results. For example, neglecting the imperfections of a realistic radio front-end in a simulation can lead to algorithms proved in simulation which may not work in the real world. Thus, the ultimate test of an algorithm is real-world experimentation.

The research area of cognitive radio is relatively new and forward-looking. There are many sceptics who doubt the feasibility of the concept as a whole. Experimental demonstrators provide a way to convince the sceptics and to establish the field, as experimental proof cannot

be ignored.

For all these reasons, it is important for researchers to experiment with cognitive radios. This thesis helps researchers to accomplish this task by enabling and simplifying experimental research with cognitive radios.

Design of Cognitive Radios for Experimentation

In order to conduct experiments with cognitive radios, prototypes have to be physically built. This involves a physical hardware *platform* and a cognitive radio *framework*, aiming to help implementing cognitive radios for experimentation. In this thesis, the term *platform* describes the physical hardware used for processing, while the term *framework* describes the combination of a software architecture, design tools, and techniques for development. A framework might optionally include a hardware platform.

In the context of this thesis, the term *cognitive radio design* describes the process of developing a plan for implementing an idea or specification of a radio on an experimentation framework or platform. This involves, among other things, the concept of the radio, architectural considerations, mapping to physical resources, algorithm design, and the choice of a target implementation framework or platform.

The stage following the design in the development process is the implementation, resulting in a realisation of the original radio concept. This can then be executed in order to conduct experiments.

Contributions

This thesis provides a number of contributions to the subject. The two key contributions are a novel cognitive radio framework for building prototypes for experimentation, and a novel platform-independent model-based design technique that simplifies designing prototypes significantly.

All contributions are listed in the following, categorised into three main groups.

Experimentation with real-time cognitive radios

1. An field-programmable gate array (FPGA)-based cognitive radio framework is proposed, called Iris-FPGA, based on the existing software-only framework Iris. It combines the

benefits of software and hardware and provides mechanisms to implement key cognitive radio concepts seamlessly in software and hardware. The FPGA implementation details are largely abstracted from the developer.

2. A series of demonstrators implemented on Iris-FPGA gives many insights into the development of cognitive radios for experimentation. They show the key concepts of cognitive radios implemented using the proposed framework.

Analysis of cognitive radio design approaches

3. A representative selection of existing cognitive and software defined radio platforms and frameworks is reviewed from the literature and key strengths and weaknesses are identified. It is found that the applied design approaches vary greatly, require in-depth understanding of the low-level platform architecture, and lack a structured way of dealing with radio reconfiguration and a cognitive engine.
4. An analysis of the proposed Iris-FPGA framework is performed and key strengths and weaknesses are identified. A key weakness is that the dynamic behaviour of the radio is not included in the radio description. That is, the possible reconfigurations the cognitive engine may trigger are not included in the description of the radio. Users have to implement this in the source code of the cognitive engine, which is prone to errors.

Model-based cognitive radio design

5. Model-based design is proposed as a solution to simplify cognitive radio design in a platform-independent fashion. A review of model-based design in general and of specific models for signal processing is given. Their suitability to cognitive radio, a special case of signal processing systems, is assessed.
6. A model of cognitive radios is proposed. It fully describes a cognitive radio, including its reconfiguration options and the connection to the cognitive engine. It allows designs to be understood and computationally verified without being first implemented. The model abstracts the radio design from the implementation platform, capturing the important aspects and omitting the implementation details.
7. The cognitive radio model is applied for model-based design. Real radio implementations can be generated largely automatic for arbitrary target platforms. This reduces design

effort, provides higher reliability, and allows easy deployment on different target platforms.

8. It is shown that the theory and concepts of the proposed model can improve the implementation of target platforms themselves. This results in a theoretically substantiated platform architecture with well-understood execution semantics and allows a more direct synthesis of the proposed model.
9. A case study applying the proposed model-based design approach is presented. An example cognitive radio is developed, from the idea to the execution on the target platform. This demonstrates the advantages of the model-based design approach. It allows radio designers to focus on the radio applications without the need to understand low-level platform details.

Thesis Outline

To represent this work's topics and to reflect the reasoning behind the proposed ideas and concepts, this thesis is organised into two main parts. Part I (chapters 2 to 4) deals with cognitive radio experimentation, i.e., the platforms, the architectures, the development of a number of real demonstrators, and the insights gained. It proposes Iris-FPGA, a new framework for the development of cognitive radio prototypes for experimentation. The experience from the first part and a thorough analysis of radio design approaches leads to the proposal of a model-based design technique for cognitive radios, the topic of Part II of this thesis (chapters 5 to 7).

The flow of the thesis begins with a review of a representative selection of existing cognitive and software-defined radio frameworks and platforms in Chapter 1. The shortcomings of these serve as a basis for the proposal of the new cognitive radio framework Iris-FPGA, described in Chapter 2. This is followed by a series of experiments in Chapter 3, focusing on demonstrating the key aspects of cognitive radios with Iris-FPGA. The insights and experience gained are highlighted in Chapter 4 and the different approaches to radio prototype design and the key challenges are analysed. This forms the basis for the second part of this thesis.

Chapter 5 proposes model-based design as a solution for the cognitive radio design challenges identified. It reviews a number of suitable models for designing signal processing systems from the literature, and assesses their suitability to cognitive radio design. The chapters that follow use this analysis and review. A new model for cognitive radios is introduced in Chapter 6, which can be synthesised to arbitrary target platforms in order to create real implementations,

i.e., it is used for model-based design of cognitive radios. Implications of this model for the implementation of the target platforms themselves are illustrated on the example of Iris-FPGA. Finally, the proposed cognitive radio design technique is tested and evaluated in a case study in Chapter 7, where a full radio is developed from the idea to the running implementation. Iris-FPGA is employed as a target platform. This demonstrates that the proposed model-based design technique can simplify the development of cognitive radios significantly, i.e, it simplifies experimental research in this field.

The literature review and the required background information for this thesis is split into two parts in order to provide the information close to where it is required. Existing experimental cognitive and software radio platforms are reviewed in Chapter 1, where this serves as a basis for designing a new cognitive radio framework in Chapter 2, and later for analysing the existing radio design methods in Section 4.2. The basics of model-based design as well as existing models for signal processing systems are reviewed in Chapter 5, as the cognitive radio model proposed in Chapter 6 uses many of these concepts.

Publications Record

The following publications directly or indirectly relate to this thesis:

Journal Articles

- J. LOTZE, S. A. FAHMY, J. NOGUERA, and L. E. DOYLE, “A model-based approach to cognitive radio design,” *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 2, pp. 455–468, Feb. 2011.
- 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,” *IEEE Communications Magazine*, vol. 48, no. 9, pp. 114–122, 2010.

Conference Papers

- P. D. SUTTON, J. LOTZE, H. LAHLOU, B. ÖZGÜL, S. A. FAHMY, K. E. NOLAN, J. NOGUERA, and L. E. DOYLE, “Multi-platform demonstrations using the Iris architecture for cognitive radio network testbeds,” in *International Conference on Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM)*, Cannes, France, Jun. 2010.

- L. E. DOYLE, P. D. SUTTON, K. E. NOLAN, J. LOTZE, B. ÖZGÜL, T. W. RONDEAU, S. A. FAHMY, H. LAHLOU, and L. A. DASILVA, “Experiences from the Iris testbed in dynamic spectrum access and cognitive radio experimentation,” in *IEEE Symposia on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, Singapore, Apr. 2010.
- J. LOTZE, S. A. FAHMY, J. NOGUERA, B. OZGUL, L. DOYLE, and R. ESSER, “Development framework for implementing FPGA-based cognitive network nodes,” in *IEEE Global Communications Conference, Exhibition, and Industry Forum (GLOBECOM)*, Honolulu, Hawaii, USA, Dec. 2009.
- J. LOTZE, S. A. FAHMY, B. OZGUL, J. NOGUERA, and L. E. DOYLE, “Spectrum sensing on LTE femtocells for GSM spectrum re-farming using Xilinx FPGAs,” in *SDR Forum Technical Conference and Product Exposition (SDR)*, Washington D.C., USA, Dec. 2009.
- S. A. FAHMY, J. LOTZE, J. NOGUERA, L. E. DOYLE, and R. ESSER, “Generic software framework for adaptive applications on FPGAs,” in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, Apr. 2009, pp. 55–62.
- M. SANCHEZ MORA, J. LOTZE, G. CORLEY, and R. FARRELL, “Experiences in the co-design of software and hardware elements in a SDR platform,” in *SDR Forum Technical Conference and Product Exposition (SDR)*, Washington D.C., USA, Oct. 2008.
- J. LOTZE, S. A. FAHMY, J. NOGUERA, L. E. DOYLE, and R. ESSER, “An FPGA-based cognitive radio framework,” in *Irish Signals and Systems Conference (ISSC)*, Galway, Ireland, Jun. 2008, pp. 138–143.
- J. LOTZE, S. A. FAHMY, J. NOGUERA, L. E. DOYLE, and R. ESSER, “High-level cognitive radio design using Xilinx FPGAs,” in *1st Collaborative International Software Defined Radio Workshop*, Maynooth, Ireland, May 2008.
- P. D. SUTTON, J. LOTZE, K. E. NOLAN, and L. E. DOYLE, “Cyclostationary signature detection in multipath rayleigh fading environments,” in *2nd International Conference on Cognitive Radio Oriented Wireless Networks and Communications (CrownCom)*, Orlando, Florida, Aug. 2007, pp. 408–413.

Demonstration Papers

- J. LOTZE, S. A. FAHMY, L. E. DOYLE, and J. NOGUERA, “An FPGA-based autonomous adaptive radio,” in *ACM SIGCOMM Conference*, Barcelona, Spain, Aug. 2009, demonstration paper.
- J. LOTZE, B. ÖZGÜL, S. A. FAHMY, J. NOGUERA, L. E. DOYLE, and R. ESSER, “Spectrum sensing to achieve frequency rendezvous using Xilinx FPGAs,” in *IEEE Symposia on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, Chicago, USA, Oct. 2008, demonstration paper.

Part I

Cognitive Radio Experiments

1 Existing Cognitive and Software-Defined Radio Frameworks

Contents

1.1 Joint Tactical Radio System: Software Communications Architecture	13
1.2 GNU Radio	18
1.3 The Iris Cognitive Radio Framework	21
1.4 Kansas University Agile Radio (KUAR)	27
1.5 Rutgers University WiNC2R	31
1.6 Virginia Tech Stalion Processor	31
1.7 Rice University WARP	32
1.8 Summary	33

Theoretical and analytical computations, simulations, and modelling are an important part of research on cognitive radio networks. Many researchers follow that route and contributed to the field significantly in recent years, e.g. in [1, 9–12]. However, to bring cognitive radio systems closer to reality, testbeds are required for real experiments. Only a combination of theory, simulations, and experiments can truly drive research forward.

This chapter reviews a selection of existing frameworks for experimenting with cognitive and software-defined radios in a research environment. It identifies their key strenghts and weaknesses for comparsion. Recall from the introduction that in this thesis, the term ‘platform’ refers to physical hardware for processing, while ‘framework’ refers to a combination of a soft-

ware architecture, design tools, and techniques for development. A framework may optionally include a hardware platform.

First, the *Joint Tactical Radio System (JTRS) Software Communications Architecture (SCA)* is reviewed (Section 1.1, as this is one of the most widely adopted architecture for software defined radios (SDRs) in industry. It is a standardised architecture designed for interoperability with many implementations available. An open source example that is widely used in university research and teaching is *Open Source SCA Implementation::Embedded (OSSIE)*, reviewed within the SCA section. Some more examples for SCA implementations from industry are outlined.

Second, the most widely used SDR framework in research is reviewed: *GNU Radio* (Section 1.2. This is an open source toolkit for use on standard PCs and off-the-shelf hardware that has a large community of developers and users. The easy availability and large code base makes this the platform of choice for many researchers.

Then Iris, a framework designed for cognitive radio experimentation from the start, including dynamic reconfiguration, is reviewed in Section 1.3.

A good example for a complete cognitive radio system developed by university researchers is *KUAR*. It combines hardware, from antenna to digital processing, and the supporting software within a portable package, making it an attractive solution for experiments. This platform is reviewed in Section 1.4.

As examples for hardware-oriented academic frameworks with innovative concepts, the *WINLAB Network Centric Cognitive Radio (WiNC2R)* by Rutgers University, the *Stalion* processor by Virginia Tech, and the *Wireless Open-Access Research Platform (WARP)* by Rice University are reviewed in sections 1.5 to 1.7, respectively.

Finally, the strengths and weaknesses of all discussed platforms are summarised in Section 1.8.

The selection reviewed here is by no means exhaustive. Rather, it reviews platforms and frameworks representing different design approaches for cognitive radios. Many more frameworks and platforms exist, but most are similar to one of those discussed here.

1.1 Joint Tactical Radio System: Software Communications Architecture

Joint Tactical Radio System (JTRS) is the US military's ongoing software radio architecture program. The main objectives of JTRS are to solve the military's interoperability problems between different devices and standards and to support legacy radio systems. The JTRS program develops an open platform-independent software radio architecture, the SCA, not limited to military use [13]. The current revision is 2.2.2 (May 2006) [14].

1.1.1 SCA Overview

JTRS designed its SCA to meet the following goals [15]:

1. Allow building an interoperable and affordable family of radios by using existing commercial off-the-shelf technology.
2. Allow (re-)programming of radios, including over-the-air.
3. Promote portability and re-use.
4. Software is independent from specific hardware solutions.
5. Extensible to new waveforms and/or hardware components.
6. Allow rapid insertion of new hardware and software technologies that become available over time.
7. Scalable from hand-held equipment to fixed stations.
8. Include embedded programmable information and security.
9. Industry acceptance as an open, commercial, international standard.

The SCA is an object oriented software architecture, defining all services and interfaces for the radio applications in Common Object Request Broker Architecture (CORBA) Interface Description Language (IDL). CORBA is a language-independent and platform-independent middleware, used to allow different parts of a distributed application to communicate through common interfaces. It allows software components to be executed as separate processes, possibly on different machines, and form a single distributed application.

The core of the SCA is the Core Framework. It consists of the Base Application Interfaces, managing all software components, the Base Device Interfaces, managing the hardware devices, the Framework Control Interfaces, controlling the instantiation, management and destruction/removal of software components, and the Framework Services Interfaces, providing additional support functions and services [14, Sec. 2.2.2].

The SCA is designed to run on a Portable Operating System Interface (POSIX) compliant operating system [16], and provides a subset POSIX functionality to Application Components. This ensures portability of applications across different operating platforms. The system-specific System Components have direct access to the operating system, since they are not required to be portable.

The core processing component in the SCA is the Resource, a software class that implements a set of radio functions. Communication with the other parts of the system, including other Resource classes, is performed via the Core Framework. If a Resource does not provide a CORBA interface, e.g., if it runs on an field-programmable gate array (FPGA) or if it is a legacy component, an Adaptor has to be created. This translates the SCA interface to the non-SCA component, effectively hiding the actual resource implementation from the system. This mechanism allows to integrate hardware and digital signal processing (DSP) resources into an SCA system.

To allow maximum flexibility and portability, typically all resources in an application are running asynchronously (as separate processes). That means that data needs to be buffered between resources and synchronisation is required for safe access to these buffers by the two processes.

Currently, the JTRS program is extending the SCA to allow interfacing to specialised hardware processors (SHPs), such as FPGAs or DSPs, via CORBA [17, 18]. Each SHP is represented to the SCA compliant radio system by a Logical Device and an Application Resource Proxy. Both of these components implement CORBA interfaces. The Logical Device represents the actual SHP device and its capabilities to the radio system. It loads or creates the task to run on the SHP directly, without a CORBA interface. The Application Resource Proxy is used to configure and control the SHP, i.e., it sets configuration parameters and provides ports for data input and output.

SHPs are typically used for tasks with a high computational complexity. For performance reasons, it is often required to connect hardware components directly, as opposed to use a separate Logic Device for each task running on the SHP. Therefore one Logic Device can handle multiple tasks on the SHP, or even on multiple SHPs. The remainder of the radio system only has access to the Logic Device, and not to the tasks therein, which allows for direct connections between tasks by the Logic Device.

Another approach to integrate FPGAs and DSPs into the SCA is using the commercial *ORBexpress* by Objective Interface Systems [19, 20]. This is a native implementation of CORBA for FPGAs and DSPs, where all CORBA handling is handled directly in hardware without the

requirement of a general purpose processor (GPP).

Many implementations of the SCA exist today, most of which are commercial or military systems. A freely available open source implementation is the Open Source SCA Implementation::Embedded (OSSIE), which is reviewed in the following. A brief overview of some commercial implementations is given thereafter.

1.1.2 Open Source SCA Implementation::Embedded (OSSIE)

OSSIE is a lightweight SCA implementation maintained at Virginia Tech [21, 22]. It is primarily intended to enable research and education in SDR and wireless communications. OSSIE includes a core framework for rapid development of SCA-based SDRs as well as a library of pre-built components and waveforms. It is licensed under GNU General Public License (GPL), making it free to use and open for modifications.

The OSSIE architecture is based on the JTRS SCA. It uses the free cross-platform CORBA implementation omniORB [23] as middleware.

All OSSIE processes follow the procedures given in the SCA [14]. However, not all of the SCA interfaces are implemented in OSSIE. The developers decided to neglect the implementation of the Aggregate Device, a Device which consists of a list of child devices, and some of the SCA exceptions [24]. This has not been considered necessary for the range of applications targeted by OSSIE.

OSSIE provides two graphical user interfaces to design radio applications. Designers can choose to use the Eclipse plug-in OSSIE Eclipse Feature (OEF) or the standalone graphical user interface (GUI) OSSIE Waveform Developer (OWD) [22]. Both provide the same functionality. Designers can browse the library of pre-compiled components, assign processing devices, compose waveforms, and set component parameters. If the library does not provide all the components needed, new components can be created using either GUI. The required source code templates and project files are generated automatically, so that the radio designers can start developing the component's core functionality immediately.

OSSIE has been ported to platforms different from x86 PCs, e.g., a Texas Instruments C64 DSP platform [21, 25, 26], or the embedded ARM processor portion of the Texas Instruments OMAP processor [27]. Significant modifications were required to deal with the limited resources and capabilities of the embedded platforms, e.g., the TI C64 platform lacks C++ STL containers, exception handling, and non-volatile storage. In [28], an architecture for integrating FPGA processing components with OSSIE is presented. It uses a software wrapper component and

memory-mapped input/output for hardware interfacing, similar to the method applied for the audio transmission demonstrator developed for Iris-FPGA, as presented in Section 3.1 and Appendix C.1.

1.1.3 Other Frameworks Implementing The JTRS SCA

Lyrtech

Lyrtech is a company that offers a variety of SDR development platforms, together with software and hardware development kits [29]. An SCA-based radio architecture is applied, which has been extended to support FPGA and DSP components [30]. Lyrtech's hardware platforms, e.g. the SFF SDR development platform, are modular systems consisting of antennas, a radio front-end module, an interface module, a baseband processor and optional expansion modules. The baseband processor is a hybrid system with a DSP processor, an ARM processor, and a Xilinx FPGA. A software development kit for DSP and processor development is provided, along with an extension of Xilinx System Generator [31] for model-based design.

PrismTech

PrismTech is a US software company specialised in productivity tools and middleware solutions for industry clients in telecommunications. Their product *Spectra* is a "complete package of new products, services, and partner programs for Software Defined Radio (SDR) OEMs, system integrators and independent software vendors." [32].

The *Spectra* software package allows for design, development and test of SCA compliant waveforms for various hardware platforms, including Rockwell Collins SCA WDS, the Spectrum Signal SDR-4000, and Gumstix miniature computers. PrismTech does not provide a radio hardware platform with radio front-end itself. It benefits from the portability of SCA-based SDRs by providing a development framework for software components then can be executed on a variety of hardware, including standard PC systems.

Spectrum Signal Processing by Vecima

The company *Spectrum Signal Processing by Vecima*, part of Vecima Networks Inc., develops SDR solutions for communications and intelligence gathering applications [33]. The company provides a set of hardware platforms and software solutions for SCA-based SDR development.

Among their products is the SDR-4000 family of black-side wireless modem solutions for tactical military communications systems. Powerful hardware systems are combined with the hardware abstraction layer and software library *quicComm*. This is a real-time operating system with integrated development environment and CORBA middleware.

1.1.4 Discussion

The SCA offers high levels of portability and connectivity to third-party components, which makes it an attractive architecture for industries developing or using radio subsystems. It is possible to sell parts of a software radio with SCA interfaces to other companies for integration into their radio systems.

On the other hand there are objections towards the overhead of CORBA for all communications, opposed to the direct communication links possible when software components are implemented in one system and one programming language, e.g., [34]. The default transport layer protocol for CORBA messages is TCP. This has a significantly higher overhead compared to passing pointers to memory between components, the approach used in single-machine applications without middleware. Furthermore, since SCA components are executed in separate asynchronous processes, scheduling their execution is handled by the operating system at run-time. For some radios it is possible to compute synchronous schedules before execution, avoiding the overhead of the run-time scheduler [35]. SCA implementations cannot avail of these techniques for optimised performance. The integration of SHPs is controversially discussed, as can be seen from the approach proposed within JTRS [17, 18], the incompatible proprietary extension of the SCA by Lyrtech [30], and Objective Interface Systems's proprietary implementation of CORBA for FPGAs and DSPs [19, 20].

The target application domain for the SCA is SDR. Therefore it lacks support for features required for cognitive radios, i.e., run-time reconfiguration of components and applications, a controller to implement cognitive decision-making, and mechanisms to obtain the state of the radio and the environment. It is possible to implement these features for a radio platform conforming to the SCA, but this requires to add components and interfaces which are not specified in the SCA, therefore diminishing the portability advantage.

1.2 GNU Radio

GNU Radio is an open source software development toolkit for software radio applications [36, 37]. It is designed to run on GPPs, allowing for easy, low-cost software radio experimentation and development. Radio applications are primarily written using the Python language, while the performance-critical signal processing components, provided, are compiled from C++. GNU Radio is licensed under the GPL and is therefore accessible, usable and alterable by everyone. Supported operating systems range from various Linux distributions to Mac OS X and Windows (in a Cygwin environment).

GNU Radio has a large number of academic, military, and private users, for instance the Cognitive Wireless Technology group of Virginia Tech [38] or the Institute for Defense Analyses (IDA) / Center for Communications Research (CCR) in Princeton, NJ, USA.

1.2.1 Architecture

The GNU Radio architecture consists of three main parts: the signal processing components, a framework to control and instantiate radios, and a scheduler controlling the execution. All performance critical blocks, i.e., the signal processing components and parts of the scheduler, are implemented in C++. They define an interface to Python, the language used to develop radio applications and control execution. A radio is composed by creating a graph of signal processing components, setting their parameters, and specifying the data path between components in Python. Hence, GNU Radio is a dataflow abstraction.

Each of the signal processing components has a number of defined input and output ports, as well as parameters to ease re-using them in different contexts. GNU Radio radios are built by connecting these port together to a graph. Each link between input and output ports is associated with a first in, first out (FIFO) buffer of a limited size. The dynamic scheduler executes components according to the state of the associated input and output buffers (input data available and enough space left in output buffer). In the current version of GNU Radio, two different schedulers exist, one for single threaded execution (one thread per graph), and one for multi-threaded execution (one thread per component). The multi-threaded version allows exploiting multi-core architectures. Hierarchical blocks, i.e., blocks that contain other flow graphs, are possible.

Recent additions include a message passing interface for each component to allow processing packet-based data associated with header information. This interface also allows to

exchange control messages with other components in the radio chain, or with Python objects. Another addition is a GUI called the *GNU Radio Companion*, which allows to browse available components, connect them to a flow graph and configure the component parameters graphically. This eases the radio design process significantly and lowers the barrier for new users.

To allow users to run GNU Radio applications without being reliant on Python, an aim for the next release is to allow users to build flow graphs entirely in C++. After compilation this results in a single application to run the radio. This is particularly attractive for embedded systems, where installing a Python interpreter may not be practical.

Run-time reconfiguration is supported by GNU Radio by pausing the radio chain, modifying the flow graph (component parameters, connections, etc.), and resuming the execution. All components that were not modified keep their internal states during reconfiguration. The reconfiguration can be triggered from within the radio chain itself, by setting up a message queue with the Python application as the message receiver and some processing component(s) are the message transmitters. For instance, if a component detects a change in the environment, it can send a message to a message queue. The Python radio application, on receiving that message, can then perform the necessary reconfigurations. It should be noted that this feature is rarely used and often requires modification of the C++ signal processing components to set up the message queue and send messages.

1.2.2 Radio Design

To design a new radio with the GNU Radio software, the user has to describe a dataflow graph. Traditionally this is done using Python. Since recently, the GNU Radio Companion GUI can be used as an alternative. Each of the components of the flow graph come with a set of parameters for configuration. A typical design flow involves the following steps:

1. Locate required components in the GNU Radio component library.
2. Decide the parameter set for each of these components.
3. Instantiate the components (in Python).
4. Connect the components (in Python).
5. Start the execution (in Python).
6. Wait for user input to stop the radio (in Python).

When using the GNU Radio Companion, the steps 3–6 can be done graphically. The GUI also comes with a library browser which simplifies Step 1 significantly.

To realise cognitive or adaptive radios, i.e., radios that change their behaviour at run-time triggered by some external event or by the radio itself, the user has to implement the reconfiguration routines in Python.

1.2.3 Discussion

GNU Radio is an extensive software radio toolkit allowing to build a vast variety of software radio implementations on standard hardware. It is flexible and easily usable at the same time, making it a powerful tool for software radio experimentation and research.

A major advantage is the extensive and well-maintained library of signal processing components, enabling fast prototyping. Many of the processing components are highly-optimised, utilising single instruction, multiple data (SIMD) vector processor features where possible (e.g. SSE, 3DNow!, AltiVec). A simple application programming interface (API) is provided for developing new processing components if required. Re-using and modifying existing processing components is allowed and encouraged by the GPL licensing scheme. Using Python to develop radio applications is advantageous due to its high flexibility and portability. The GUI, message passing interface, and multi-threaded scheduling enhances the usability, flexibility, and performance of GNU Radio.

GNU Radio was originally designed for SDR applications. Support for a cognitive engine and high run-time reconfigurability was not an original design goal. However, today cognitive radios can be implemented on GNU Radio. The cognitive cycle is managed by the Python script which also sets up the radio configuration.

To *observe* the radio and environment, the script can obtain information about the running radio (the flow graph and parameter settings) through the Python interface of the signal processing components. Message queues can be setup between a C++ signal processing component and the Python script, so that the component can inform the script of events and send information. In the *decide* stage, the Python script can perform whatever form of decision-making is required. The vast number of numerical and optimisation libraries available from Python may be used where required. To *act*, the radio is paused, components or the flow graph are reconfigured, and the radio is restarted again, all from the Python script.

The message queue setup required for the *observe* stage needs to be explicitly setup from the C++ component code in order to allow the Python script to receive this information. That is, often the source code of the components need to be modified to setup the required message queue, reducing the re-usability of the component in other configurations. This is less efficient

for a cognitive engine compared to publishing events, the mechanism used in Iris. Here, an event can be subscribed to multiple times, or never. The connection is more loose, allowing components to publish a whole series of events, enabling use without source code modification in different radio setups. Thus, it can be said that using GNU Radio for cognitive radios is more an add-on rather than a built-in feature.

GNU Radio is of limited use for embedded systems due to the large number of external library dependencies and the overhead of a Python interpreter. For embedded systems, it is desirable to minimise the dependencies and footprint of an application.

Another disadvantage is that it takes a considerable amount of learning and experimentation to be able to design efficient radios with GNU Radio. Designing radio flow graphs often requires an in-depth understanding of the implementation details of the components and GNU Radio in general. The documentation often does not provide a solution, and the user might be required to read the source code or consult the mailing list.

Thus, GNU Radio is a capable and feature-rich SDR framework with a large user and developer community. Although it can be used for cognitive radio implementations, its capabilities in this domain are limited and radio designers often need detailed knowledge of implementation details.

Designing a cognitive radio using the GNU Radio toolkit requires a good working knowledge of Python, since reconfiguration can only be handled through Python scripts. Reconfiguration and connection to a cognitive engine is performed using Python, with environmental information retrieved from the radio via message queues. This often requires changing the processing component's implementation, so that messages are sent to the message queue, hence, this feature is used rarely.

1.3 The Iris Cognitive Radio Framework

Iris has been under development at Trinity College Dublin since 1999 [39]. It is a flexible framework for developing physical layers of cognitive radios, including mechanisms for cognitive decision-making and run-time reconfigurability. The building blocks of a radio in Iris are signal processing components with a specific function, e.g., a pulse-shaping filter or a convolutional encoder. These may have parameters for adjusting their behaviour, for example, for setting the roll-off factor of a pulse-shaping filter. A radio is composed by connecting these processing components together, where data is passed between components along these connections. This

is the description of the processing part of the system. Additionally, every component can trigger events which a radio controller can subscribe to. Whenever such an event is triggered, the controller has the opportunity to reconfigure the running radio, either by changing component parameters or by replacing components.

Iris, in the version of 2006, is written in C++ for execution on the Windows operating system. Components are compiled into dynamically loadable libraries (DLLs) so that no recompilation is required for new radio designs from existing components. The radio controller is also written in C++, using a simple API to the running radio for subscribing to events and for reconfiguring the radio. Radios configurations are described in an Extensible Markup Language (XML) file, listing the used components with their parameter settings. This describes the *data plane* of Iris, i.e., the part where the data processing takes place. Optionally, a radio controller can be specified in the XML file, which together with the Iris run-time system forms the *control plane*, i.e., the part where the execution of the data plane is controlled. This XML file is parsed by Iris during setup, components are loaded, linked, and initialised, the controller is loaded and subscribes to events, and the radio is executed by continuously passing data between components and processing it.

The cognitive cycle of *observe*, *decide*, and *act* is executed as follows. Component events are triggered by a component to inform the controller of the state of the radio or the environment. This represents a *meter* of the radio from the controller perspective, read during the *observe* stage. A callback function in the controller is executed in response to the event, performing decision-making and possibly machine learning. This is the *decide* stage. The controller may then *act* by reconfiguring the radio, for instance, by tuning a component parameter or by replacing a component with a different one. This reconfiguration possibilities represent the *knobs* of the radio.

In the following we will take a closer look at the architecture and design flow of Iris. This information is largely based on [39].

1.3.1 Architecture

The Iris architecture is illustrated in Figure 1.1. As can be seen, the basic building blocks of Iris are the *Iris run-time system*, the *component library*, the *data plane* (described by the *XML radio description*), and the *radio controller*. In the following each of these parts is examined in more detail.

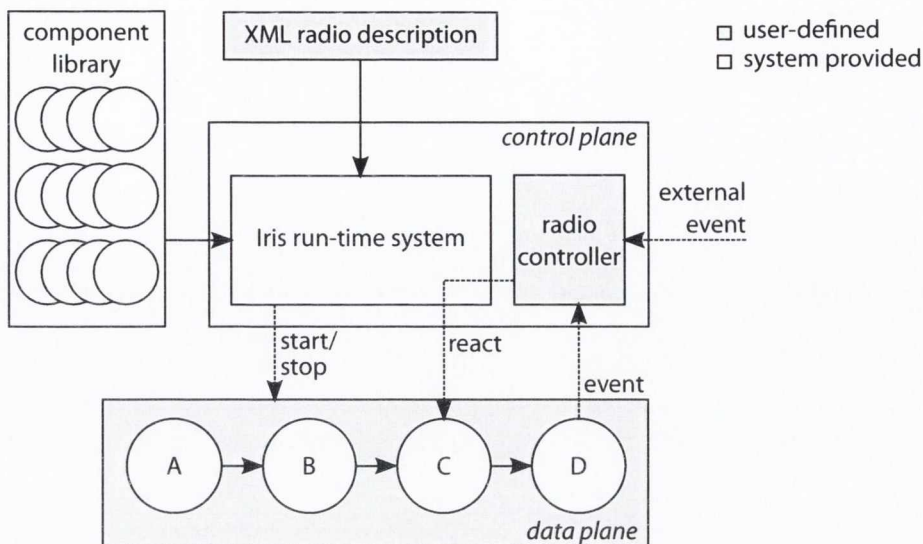


Figure 1.1: The Iris architecture. A radio with four components A, B, C, and D is shown as an example. The user describes the radio in an XML file and writes the radio controller in C++.

Component library

The *component library* holds a set of Iris components, used as building blocks for radio configurations. Each component is responsible for performing one specific processing step on a stream of data. Examples are a modulator or a pulse shaper component. Radios are constructed by connecting these components and configuring them. Components are written in C++ and compiled into dynamic libraries loadable by the Iris run-time system. They have input and output ports, for connecting them into a radio components, and parameters for adjusting the component behaviour without recompilation.

Events are used by components to trigger a call to the radio controller and provide information to it. For instance, a decoder component might estimate the current bit error rate (BER) and report it to the controller using an event. This event interface is key to achieve dynamic reconfiguration of the radio based on the current environmental and radio-internal conditions.

A wide variety of different components have been implemented for Iris so far, ranging from simple signal scaling components to complex orthogonal frequency division multiplex (OFDM) modulators. The component library holds all these compiled components, ready to be used in a cognitive radio.

Data Plane

The *data plane* of Iris is specified in an XML file (see Figure 1.1). It is composed from components contained in the component library, with all parameters set to specific values and all inputs and outputs connected to other components. This XML syntax allows to describe a wide range of radio configurations typically used in cognitive and software radios in a simple and efficient way. Various different radio configurations can be created easily by connecting and configuring the components of the component library; no source code compilation is required. The full syntax of the Iris XML format as well as some illustrative examples are described in Appendix A.

The radio's execution semantics, i.e., how the radio configuration is executed, is simple and efficient. The Iris scheduler, part of the run-time system, calls each of the components in a sequence from source to sink, once in each scheduling cycle. That is, the scheduler executes a simple loop with each component executing once. It provides each component an input block of data to process, and passes the computed output block on to the next component. Every block of data is associated with a valid signal; components may use this to signal other components down the chain that the output data is invalid and can be ignored.

This simple radio execution requires no buffers between components since there is no concurrency and output data blocks are immediately consumed by the next component. This allows fast execution of the radio, without reallocating memory, managing buffers, etc. Block sizes are fixed during normal execution and components have no control about the size of the input data block they are given. If a reconfiguration happens, either triggered by the radio controller or by a change in the XML file (see the following sections), the block sizes are recomputed and memory is reallocated as required.

Radio Controller

A *radio controller* is the decision-making part of the radio, i.e., it executes the cognitive cycle of *observe*, *decide*, and *act*. It can access information about the currently running radio as well as reconfigure it.

Information about the components currently executing and their parameter values can be accessed by the radio controller and are provided by the run-time system. This is one method to *observe* the status of the running radio. Further, radio controllers can subscribe to events that are published by components, in order to get informed about changes in the environment or the radio itself. Components can trigger these events during radio execution, sending information to the radio controller. For example, a spectrum sensing component might send estimated

spectrum occupation data to the controller. When the controller subscribes to an event, it registers a callback function which is invoked when the event is triggered. This is the second method for the controller to *observe*. Both the component information interfaces and the published events can be considered the *meters* of the cognitive radio.

For the *decide* stage, radio controllers are free to implement any decision-making algorithm they choose. For example, they can implement a simple state machine, switching different radio operation settings based on predefined conditions, or a genetic algorithm optimising a multi-objective function to determine optimal settings for the radio.

During the *act* stage, the radio controller can reconfigure the radio through various different methods. Parameters of components can be adjusted, e.g., setting the gain parameter of a scaling component. Components can be replaced with different ones from the component library, for example replacing one modulator for another.

The controller to be used with a radio is specified in the XML description (see Appendix A).

This simple mechanism allows for adaptation of the radio to changes in the environment, a concept central to cognitive radio. Controllers from simple decision tables or state machines to ones employing learning using artificial intelligence principles are possible with this architecture. Hence, it is a clear enabler for implementing cognitive radio systems.

Iris Run-Time System

The Iris *run-time system* is the core of Iris. It parses the radio XML description, loads and manages radio components, manages memory, handles radio execution, loads and initialises radio controllers, etc. It provides simple interfaces to the radio components, controllers, and the user and manages all low level tasks.

The lifecycle process of a radio is as follows. On start-up, the XML file describing the radio is parsed and the Iris run-time engine creates the radio. It instantiates, connects, and initialises the specified components. In the next step, it loads the radio controller, also given in the XML file, and invokes a function let it attach to the components (i.e., retrieve radio information and subscribe to events). Finally, the radio is started and blocks of data generated by the source component are processed by each of the components in the radio chain in sequence. When one of the events the radio controller has subscribed to is triggered by a component, the controller can react in many ways, as described above. The user can stop the radio execution, or pause and resume via a simple text-based interface.

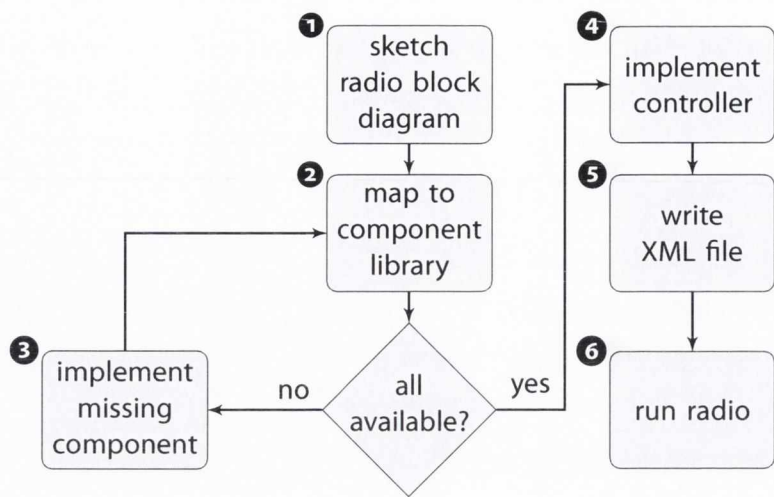


Figure 1.2: The Iris design flow.

In addition to the radio controller, the user can also reconfigure the radio by manipulating the XML file of the running radio and telling the Iris run-time system to reconfigure. The XML is parsed, differences are identified, and the changes are applied to the radio. This provides a simple mechanism for run-time reconfigurations without developing a controller which many users find convenient for development and experimentation.

1.3.2 Radio Design

Typically, a radio is designed in Iris following the steps illustrated in Figure 1.2. The first step in every radio design is to divide the radio into components and sketch a block diagram describing the radio’s functionality (1 in the figure). This step is independent of Iris and typically involves a sheet of paper or a white board. In 2, the components of the sketched diagram are mapped to the ones existing in the Iris component library. This mapping might require some variation of the original diagram, depending on component granularity and availability. If one or more components are not in the library provided by Iris, the radio designer has to implement them using the simple API described in Section 1.3.1 (3 in the figure). If a radio controller is required, the radio designer has to implement it as the next step (4), using the API described in Section 1.3.1. Now all the implementations are ready. All that is left to do is writing the radio description in XML format (5) and running the radio with Iris (6).

The Iris architecture and component library allows for high levels of re-usability. That is, components can be used in a variety of different radio configurations without recompilation. Modifications with a particular radio design can be made in the XML file, making it easy and

fast to experiment with a particular radio. Even if a component is missing in the library and must be implemented manually, this is only a small part of a full radio system, with simple interfaces. All other parts of the radio system can be used ‘out of the box’.

1.3.3 Discussion

Iris has proved very efficient and easy to use for various cognitive radio applications (see for example [40–42], where it has been used for dynamic spectrum coexistence experiments, for experimenting with cyclostationary signatures as an innovative rendezvous technique in dynamic spectrum scenarios, and for exploring various reconfigurability opportunities for multi-carrier CDMA). Its strengths are reconfigurability, ease-of-use, simple APIs, and rapid prototyping. On the other hand, it is bound to the Windows operating system running on GPP hardware, which severely limits its usability for setups requiring more compute power and smaller form factors. Other weaknesses include the limited flexibility of the scheduler and the lack of support for parallel hardware architectures. Nevertheless, it is a complete and ready-to-use cognitive radio framework with a clever architecture and concept.

1.4 Kansas University Agile Radio (KUAR)

The KUAR is a compact, powerful, and flexible software radio development and experimentation platform developed at Kansas University [43]. It is primarily aimed to enable advanced research in the areas of cognitive radio, cognitive networks, and dynamic spectrum access. The KUAR is an integrated hardware and software platform, including antennas, radio frequency (RF) components, digital baseband processing hardware, and a control PC. Due to its small form factor and the optional battery pack it is mobile and therefore allows experiments in the field.

1.4.1 Architecture

Hardware

The radio consists of a power supply, a digital board, and a RF board within a shielded box of approximately $17.8 \times 7.6 \times 15.2$ cm. The transmit and receive patch antennas are active (include amplifiers) and mounted outside of the shielded box. This is the default antenna configuration, however, other antenna configurations are possible. The power supply can be either battery

driven or mains driven. The digital board contains an embedded PC for diagnosis and control and a Xilinx Virtex-II Pro FPGA for high-speed signal processing. The embedded PC is equipped with a 1.4 GHz Intel Pentium M processor, 1 GB RAM and a 8 GB MicroDisk running the Linux operating system, which allows implementing sophisticated radio controllers using standard tools. To ease the integration of a KUAR node into an existing network it is equipped with a gigabit Ethernet port. To simplify development and debugging, the KUAR can be used as a standard PC with graphical desktop using the USB and VGA connectors. The FPGA can be accessed and configured from the Linux PC via PCI Express, PCI, or USB, depending on the required performance and available FPGA resources. The fastest connection is through PCI Express, but this requires the largest footprint in the FPGA. The slowest connection is via USB with the lowest resource requirements, and PCI is a solution providing average performance with average resource requirements. The RF board is responsible for tuning the frequency and power (maximum bandwidth is 30 MHz), up/down conversion, filtering, and analog-to-digital/digital-to-analog conversion.

Software

The KUAR platform is a hybrid system of heterogeneous processing elements. The FPGA logic provides high-speed parallel signal processing, typically used for all low-level baseband processing of a radio. The two embedded PowerPC processors in the Xilinx Virtex-II Pro FPGA can be used for radio and FPGA control functions and for basic signal processing. They also allow to run a secondary Linux for standard operating system features directly on the FPGA device. The powerful Pentium host processor runs the main radio control, i.e., it is the “glue” for all processing elements. Linux drivers and tools for accessing the other parts of the system are provided. The host processor can also execute signal processing and networking functions and run diagnostics and configuration tools. All these elements are programmable with standard tools, i.e., Xilinx tools (ISE [44], SystemGenerator [31]) for FPGA programming and standard compilers (GCC, etc.) for the PowerPC and Pentium processors. The software architecture is modular, using a custom middleware to interface between all software and hardware components of the system. An illustration is given in Figure 1.3.

A radio configuration is created based on a library of pre-compiled modules for the various processing devices available in KUAR. When booting the radio, these modules are loaded onto the appropriate subsystem of the KUAR platform, configured, and started. During execution, test data can be generated and the radio performance is measured and logged for offline access [45]. Modules communicate with each other via a custom middleware, which is composed

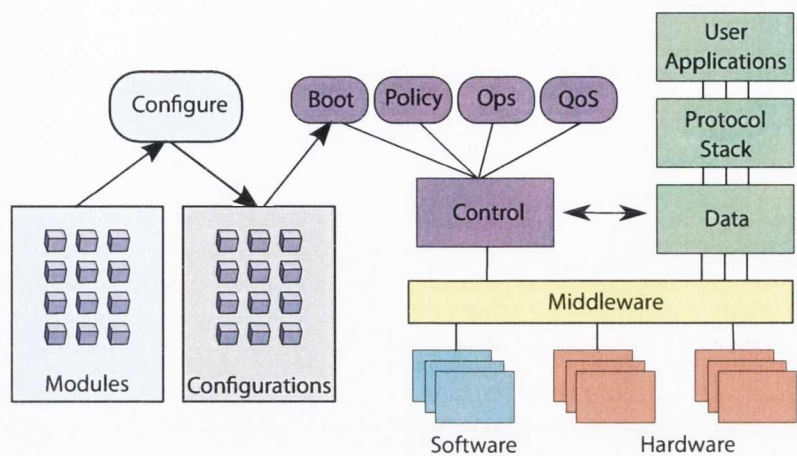


Figure 1.3: The KUAR software architecture (based on [43]).

of a set of software APIs and hardware interfaces. Software APIs are available for RF front-end control and for FPGA control and data access. The FPGA hardware interface is called the *KUAR Memory Interface*, which is a set of VHDL modules that provide an abstraction for interaction with software. These include registers, FIFO buffers, and memories shared between FPGA and software. Data to and from the RF board is always passed through the FPGA. Using these abstractions it is possible to freely decide which parts of the KUAR platform should be used for which parts of a radio design. Typically most physical layer signal processing is implemented on the FPGA while networking and control is implemented in software on the host processor.

1.4.2 Radio Design

A typical design flow for the KUAR is presented in [43]. Radio designers model and simulate their design with high-level tools, e.g., MATLAB and Simulink. Once convinced that the design is functionally correct, they browse the component library that is provided with KUAR for re-usable components. If required components are missing, they are implemented by the designers using standard FPGA design tools, e.g., VHDL, Verilog and Xilinx ISE [44]. Once these designs are loaded onto the FPGA, designers can use a set of test and debugging tools provided with KUAR. These support, for instance, signal generation, interoperability with MATLAB or Simulink, linking with other KUAR radios via Ethernet or wirelessly, or displaying diagnostic plots like eye diagrams or BER curves.

An abstraction layer for easy transfer of data to and from the FPGA allows implementing control, networking, and higher layer functionality from Linux without in-depth knowledge

of the FPGA implementation.

To provide maximum flexibility, no radio design toolkit is provided with KUAR, i.e., no tools for connecting flow graphs of components, execution scheduling, etc. are provided. Designers may run GNU Radio (Section 1.2), OSSIE (Section 1.1), any other software radio framework, or their own custom radio applications on the KUAR.

Reconfiguration and control is performed by the components Policy, Ops, and QoS (see Figure 1.3). The QoS component keeps track of changes in the signal quality and the environment and signals those to the Ops component. This adjusts the radio operation settings accordingly, e.g., by changing the modulation type, the frequency, or the forward error correction technique. All changes have to run through the Policy component, which ensures that the operation parameters are permitted [45].

1.4.3 Discussion

The KUAR platform is a complete and portable hardware solution for cognitive radio experimentation. Its capabilities are vast, enabling a wide variety of applications. On the other hand the library of pre-compiled radio modules and configurations is still small; only a small number of implementations for point-to-point communications are available thus far [45–47]. The software architecture and radio design methodology is handled at a low level, i.e., no high level toolkit for constructing radio dataflow graphs, monitoring its execution, and handling reconfiguration is provided.

Radio designers have to implement the scheduling and synchronisation techniques, the radio life-cycle management, and the control of the radio reconfiguration manually. This offers high flexibility, with the drawback of decreased portability of radio modules and increased development times. Most of the designs proposed in [45–47] are pure hardware designs using Xilinx FPGA design tools. The software API is used for loading the FPGA configuration, testing, debugging, and diagnostics only.

The KUAR platform is developed and built within Kansas University. It is not available for purchase, making it difficult for other researches to get access to the platform and use it for their own experiments.

1.5 Rutgers University WiNC2R

The WiNC2R is a cognitive radio platform developed by Wireless Information Network LABoratory (WINLAB) at Rutgers University [48]. It uses flexible hardware accelerators to achieve programmability and high performance at each layer of the protocol stack. The prototype consists of one or more baseband modules, each connected to an RF module, a networking module, and a central processing unit (CPU). The performance intense radio functions are executed on dedicated hardware accelerators (implemented in FPGA logic), while less processing intensive functions are performed by data processors (implemented as soft CPUs on the FPGAs). The data transfers between those elements are performed using fast direct memory access (DMA) techniques. A system scheduler manages the synchronisation of all the processing elements as well as the data transfers. The external CPU manages the system reconfiguration; it can reconfigure hardware accelerators, data processors, and the systems scheduler if required.

The WiNC2R is still at an early stage, i.e., the prototype is not fully developed yet. Early tests show encouraging results, but no reference designs have been demonstrated yet [48]. Thus, no information about approaches for designing radios for the WiNC2R framework could be found in the literature.

High level and easy-to-use design tools for the WiNC2R are key to allow other researchers using this platform for their experiments. According to the website [49], the WINLAB is currently working on such tools and software support.

1.6 Virginia Tech Stalion Processor

The Mobile and Portable Radio Group (MPRG) at Virginia Tech developed a layered radio architecture for software radios [50] and proposed the Stalion processor for the implementation of its physical layer [51]. The Stalion uses the *stream-based processing* concept, i.e., streams of packets containing either programming information or data are pipelined through functional blocks. If a programming packet arrives, the functional block is reconfigured with this new information. Data packets are processed with the current programming information. This architecture allows reconfiguration of hardware blocks on a per-packet basis. The Stalion consists of an interconnected array of functional blocks (an arithmetic logical unit (ALU) with some flow control logic), which can be reconfigured on the fly. A cross-bar switch connects all functional blocks and can be programmed to reflect the required data path through the

system. The ALUs support addition, subtraction, bit shifts and logic operations on the 16 bit inputs. Some of the functional blocks are multipliers. This architecture makes the device highly parallel, with a lower granularity than FPGAs (functional blocks instead of look-up tables and flip flops). The performance levels lie between those of GPPs and FPGAs.

The Station architecture is a powerful concept for on-the-fly reconfiguration of radio functionality.

However, the programming tools available are low-level, requiring in-depth understanding of the underlying hardware. In [51], a set of low-level graphical programming tools are described for the Station, which set the programming bits of each functional unit as well as switching cross-bar configurations directly. This developing cognitive radios for the Station a difficult task requiring considerable effort and in-depth architecture knowledge.

1.7 Rice University WARP

The WARP is a scalable, extensible, and programmable hardware platform using an FPGA as its baseband processor and up to four RF daughter boards [52]. The physical layer of a radio is implemented in the FPGA logic fabric, while medium access control (MAC) layer functionality can be implemented in C using the two embedded PowerPC processor cores (without an operating system). The first version of the WARP uses a Xilinx Virtex-II Pro FPGA with two embedded PowerPC processor cores for baseband processing. In October 2009 a new version of the WARP board was released, featuring, among other capabilities, the more powerful Virtex-4 FPGA and gigabit Ethernet connectivity [53].

Various reference designs and tutorials are available detailing a variety of radio implementations on the WARP board. Standard Xilinx tools are used to program the WARP board (typically System Generator [31]), allowing efficient software radio implementations. WARPLab, a framework for interaction with the WARP board from MATLAB, allows signals generated from MATLAB to be transmitted using the WARP board or analysis of signals received from the board. This can aid the design of radio applications and facilitates rapid prototyping.

To implement a full radio application on the WARP board, requires significant low-level hardware design experience. No framework for composing radios from functional blocks, e.g., modulators or filters, is provided. The Simulink-based System Generator tool provides a graphical methods for designing hardware, but designers still have to understand low-level details to be able to design radios for the WARP.

The WARP board is primarily designed for SDR experimentation. No standard methods or interfaces for implementing cognitive radios are provided. That is, implementing a cognitive decision-making process, obtaining information about the environment and reconfiguring the radio during run-time requires cumbersome low-level implementation from the ground up. Cognitive radios adapting the operation of the physical layer of a radio on-the-fly have not been demonstrated on the WARP board yet.

1.8 Summary

A number of different cognitive and software-defined radio frameworks and platforms have been reviewed in this chapter. They range from an abstract software architecture (SCA), over pure software implementations (OSSIE, GNU Radio, Iris), and combined hardware and software solutions (KUAR), to pure hardware boards and devices (WiNC2R, Stalion, WARP). The selection reviewed is believed to be representative for the full range of platforms available today.

Each of the discussed frameworks follows a different approach to designing and executing cognitive radios in a testbed, with advantages and disadvantages. Summaries of the strengths and weaknesses for each of the reviewed frameworks are given in tables 1.1 to 1.7.

Strengths	Weaknesses
• portable open royalty-free standard • many implementations available (proprietary [29, 32, 33] and open source [21]) • Different implementations can be combined • proprietary solutions for integrating DSPs and FPGAs available • widely adopted by the industry • high flexibility	• asynchronous communication between components (inefficient on a single processing device) • communication overhead due to CORBA middleware • often too complicated architecture for simple applications • target domain: SDRs, not cognitive radio • no mechanisms for cognitive cycle and run-time reconfiguration

Table 1.1: Strengths and weaknesses of JTRS SCA [14].

Strengths	Weaknesses
• accessible open-source toolkit • high performance on standard hardware • large number of optimised processing components • flexible radio application development in Python • easy of use through GUI	• not suited for embedded platforms • cannot use FPGAs or DSPs • mainly for SDRs • cognitive radio development cumbersome • poor documentation

Table 1.2: Strengths and weaknesses of GNU Radio [36].

Strengths	Weaknesses
• built-in support for cognitive cycle (radio controllers) • ease of use (component-based appraoch, XML configuration) • plug-in based approach simplifies adding new components	• Windows only • not suited for embedded platforms • cannot use FPGAs or DSPs • limited flexibility in radio configurations (branching/mergin/feedbacks are difficult or impossible) • simple sequential execution ignores parallel hardware

Table 1.3: Strengths and weaknesses of Iris [39].

Strengths	Weaknesses
• complete and portable hardware (PC, FPGA, front-end, antennas) • capable of advanced high-performance radio implementations • high flexibility through custom low-level design • APIs for common functionality provided	• low-level development, detailed platform knowledge required • limited availability to other researchers • small number of provided waveforms and reusable components

Table 1.4: Strengths and weaknesses of KUAR [43].

Strengths	Weaknesses
• complete hardware platform (networking, baseband, RF, and CPU modules) • network stack support • uses high-performance hardware (FPGA) • efficient parallel architecture, streaming data through processing elements • run-time reconfigurable processing elements • fast data transfers between elements	• still early prototype • no reference designs demonstrated • no information about design approach • non-standard hardware → requires custom design methods

Table 1.5: Strengths and weaknesses of WiNC2R [48].

Strengths	Weaknesses
• stream-based processing concept suits cognitive radios • reconfiguration of functional blocks on per-packet basis (highly reconfigurable) • highly parallel hardware device (performance lies between GPPs and FPGAs)	• low-level programming, requiring in-depth understanding of hardware • custom hardware design, i.e., standard development tools cannot be used • high development effort

Table 1.6: Strengths and weaknesses of the Stalion [51].

Strengths	Weaknesses
• complete hardware platform (including baseband processing and radio front-end) • support for higher layers (networking) • leverages high-performance hardware (FPGA) • a number of reference designs available • good documentation • available off-the-shelf • WARPLab, a Matlab-based tool, aids design of radios	• low level design, using standard FPGA design flow • FPGA design experience required • networking implementation on bare PowerPC without operating system requires advanced software programming skills • implementing cognitive cycle difficult

Table 1.7: Strengths and weaknesses of WARP [52].

2 An FPGA-based Cognitive Radio Framework

Contents

2.1 Hardware Platforms for Cognitive Radio Experimentation	39
2.2 Iris-FPGA	40
2.3 Summary	60

The previous chapter reviewed various existing frameworks and platforms for experimenting with cognitive radios. It could be seen that a variety of different approaches exists, all with strengths and weaknesses. None of the reviewed frameworks however provided an easy-to-use methodology for radio design and at the same time a hardware platform that is capable to run complex radios in real-time. The software-centric frameworks (Software Communications Architecture (SCA), Open Source SCA Implementation::Embedded (OSSIE), GNU Radio, Iris) cannot not deal with real-time hardware without major difficulties. The frameworks with real-time hardware (Kansas University Agile Radio (KUAR), WINLAB Network Centric Cognitive Radio (WiNC2R), Stallion, Wireless Open-Access Research Platform (WARP)) are using a low-level approach to design radios and are not built for cognitive radios with their requirement to support run-time reconfiguration. Therefore, this chapter proposes a framework for the implementation of cognitive radio prototypes for the purpose of experimentation than does not have these shortfalls. This provides mechanisms to implement the key aspects of cognitive radios, i.e., reconfigurable signal processing and the cognitive cycle of *observe*, *decide*, and *act*. It is computationally powerful, and yet, easy to use.

In order to perform experimentation, cognitive radio prototypes have to be physically built.

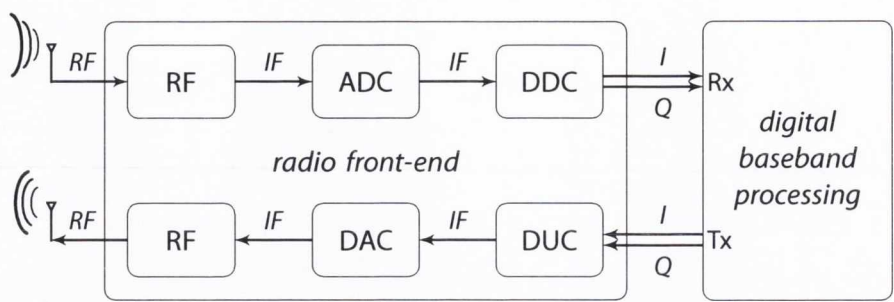


Figure 2.1: Components of a typical radio.
The Tx radio frequency (RF) component filters, amplifies, and converts to an intermediate frequency (IF). The analog-to-digital converter (ADC) digitises. The digital down converter (DDC) converts to baseband in-phase (I) and quadrature (Q) components. The digital up converter (DUC) converts to IF. The digital-to-analog converter (DAC) converts to analog. The Tx RF component up-converts, filters, and amplifies.

This involves an analog radio frequency (RF) part connected to an antenna, conversion to and from digital baseband signals, and digital baseband processing, as illustrated in Figure 2.1 This thesis focuses on the digital baseband processing part of the radio, specifically on the physical PHY layer. A radio front-end, consisting of the RF part, ADC/DAC, and DDC/DUC, can be purchased off-the-shelf and is not subject of this thesis (see Figure 2.1 for an explanation of these parts).

The digital baseband processing consists of the hardware platform it executes on and a software framework responsible for managing the execution and implementing radio functionality. As a hardware platform, a combination of a general purpose processor (GPP) and field-programmable gate array (FPGA) was selected since this combines high flexibility with high performance. As software framework, the existing cognitive radio framework *Iris* [39] has been chosen (see Section 1.3 for a review). It provides all the features required for cognitive radios, was especially tailored for the reconfigurability required, and was readily available within the research group. It is modified for the execution on an FPGA-based platform and combined with a hardware architecture in Section 2.2. This forms the proposed FPGA-based cognitive radio framework, termed *Iris-FPGA*.

2.1 Hardware Platforms for Cognitive Radio Experimentation

Wireless communications systems today are running more and more complex digital signal processing (DSP) algorithms. Examples are the different coding and interleaving schemes and the computationally complex orthogonal frequency division multiple access (OFDMA) access scheme in the IEEE 802.16 family of standards [54, 55], also known as Worldwide Interoperability for Microwave Access (WiMAX). Cognitive radios add even more complexity, potentially running advanced decision-making and machine learning algorithms in addition to the signal processing. Furthermore, they are required to be run-time reconfigurable in order to adapt to the current environment. Thus, the hardware platform for cognitive radio testbeds needs to be fast and reconfigurable at the same time.

In terms of power consumption and form-factor, one has to distinguish between testbeds purely for research and experimentation in a lab environment and mobile production-type systems for tests in the field. For the former, power consumption and form-factor is not of concern, as long as it is within reasonable limits. The latter is not so straightforward and poses significant challenges to keep the power consumption low while still providing the processing power needed for the application. For simplicity and practical reasons, the focus of this thesis is on systems for research labs.

A wide range of hardware platforms for digital computations is available today. Appendix B reviews a selection of them, namely application specific integrated circuits (ASICs), application specific standard parts (ASSPs), GPPs, DSPs, graphics processing units (GPUs), the Cell Broadband Engine (CellBE), and FPGAs. Their properties in terms of (re-)programmability, performance, form factor, and other criteria are evaluated in the appendix. It is shown that the platforms that are easy to develop for and offer high re-programmability are comparably slow, while the faster ones are more difficult to develop for.

Cognitive radio systems need the reprogrammability of a GPP and the performance and power efficiency of ASICs or ASSPs. However, no such platform exists. The trade-off between flexibility, power consumption, and performance must therefore be made carefully for a cognitive radio system.

The main decision factors are the type of the radio, the affordable engineering effort, and the price. For instance, if a highly mobile radio with low power consumption and a small form factor is required, ASICs, ASSPs, or DSPs are suitable platforms. Typically a combination of

these is used, sometimes on a single chip, to achieve maximum efficiency.

For a cognitive radio testbed in a research environment, mobility and power consumption are not of concern. Instead, tool support, development cost, flexibility, and reprogrammability are the dominating factors. The most suitable platform in this scenario is the GPP. However, if the performance of a GPP is not sufficient, more powerful platforms need to be considered. Suitable options are DSPs, GPUs, the CellBE, or FPGAs.

Often the best alternative is a combination of a GPP and one of the faster processing platforms. The GPP is used for managing the radio execution, executing higher layers in the network stack, and the cognitive decision-making part of a cognitive radio. The faster platform, connected to the GPP, executes the performance-critical parts of the radio, similar to a co-processor. These are typically the digital signal processing close to the antenna, where the sample rates are highest.

It is important for a testbed framework for research to be useable for a wide variety of different radio setups and experiments without much effort. That is, setups from simple narrowband point-to-point links to complicated OFDMA multi-user networks with forward error correction (FEC) and interleaving must all be supported by the testbed framework. Only very high performance platforms will support this wide range of test setups. For this reason, FPGAs were chosen as a hardware platform to complement a GPP as they provide the performance required for both simple and complicated setups. To provide the run-time reconfigurability required, the dynamic partial reconfiguration feature of Xilinx FPGAs has to be exploited.

With the hardware platform selected, the next section proposes a cognitive radio framework based on Iris (see Section 1.3), that uses a combination of GPPs and FPGAs.

2.2 Iris-FPGA

In an ideal world, a cognitive radio framework would combine high flexibility and reconfigurability with high compute performance, while allowing simple and straightforward design, development and testing. Unfortunately, this is difficult to achieve, as these goals often contradict each other. As follows from discussions Section 2.1, a promising hardware platform is a combination of a GPP and an FPGA. Some parts of the radio are more suitable for the GPP, e.g., radio execution management or decision-making, while other parts need the high processing power of the FPGA logic fabric. Both parts should work together seamlessly in order to provide the best possible performance and flexibility.

This section proposes a framework which allows to design and execute cognitive radios for an FPGA-based platform, seamlessly integrating hardware and software components. The Iris framework is adapted and extended to enable exploiting the FPGA's processing power. It manages the radio execution and integrates hardware components into the radio. The aim is to achieve a level of reconfigurability similar to a pure software implementation by exploring dynamic partial reconfiguration of Xilinx FPGAs, as explained in Section B.3.

For convenience, the framework developed in this section is named *Iris-FPGA* throughout the remainder of this thesis.

The architecture proposed in the following sections is in its majority the work of the author of this thesis. Help was provided by Juanjo Noguera of Xilinx in the selection of the FPGA board, the setup of the design tools and the board itself, handling the partial reconfiguration, and some other implementation issues. Together with Dr. Suhaib Fahmy he also developed the inter-component interface in the FPGA hardware and the hardware side of the software-hardware interface.

This section starts with outlining the goals and requirements for Iris-FPGA in Section 2.2.1, before presenting an architecture that fulfils these requirements in Section 2.2.2. Section 2.2.3 discusses selected implementation issues which are considered important for creating a functional system according to the requirements. It is difficult to handle and develop for FPGAs without considerable experience in hardware design. Therefore the proposed framework attempts to hide as many low-level hardware details from radio designers as possible. This is discussed in Section 2.2.4.

2.2.1 Requirements

From a high level view, two major goals can be set for Iris-FPGA. These are

1. to port and integrate Iris onto an FPGA-based hardware platform and
2. to abstract and hide low-level hardware details from the user.

The reason for the second goal is to allow radio designers to experiment with the platform without the need to be an FPGA design expert.

Based on these high level goals, the following sections develop a list of hardware and software requirements.

Hardware

There is a large number of different types of FPGAs available from different vendors today, all of which have different specifications and capabilities. Additionally, the development boards employing FPGAs are available in great variety. Therefore, to increase the versatility the proposed framework, it should be portable between different boards and chips without too much effort. Thus, in the following, specific development boards are abstracted from and a list of required features and capabilities for the hardware is compiled.

Iris is executed in software on a GPP on top of an operating system. Therefore, a capable GPP is required on the development board. For this framework, Linux is chosen as the operating system due to its high availability and unmatched portability between different processor architectures. Linux can be executed on almost any GPP architecture. The most common ones for embedded systems are PowerPC and ARM, both can be found on FPGA development boards. If there is no explicit GPP available on the board, it is possible to instantiate a soft GPP core in the FPGA logic fabric. Many different processor architectures are available as an FPGA core, e.g., the Xilinx MicroBlaze soft processor [56]. Linux has also been ported to the MicroBlaze architecture. Thus, it is required that the hardware platform provides *a GPP for which a Linux port is available*.

In order to boot Linux and run software applications, some basic hardware components are needed. One of which is *system memory* which is used to store the Linux kernel and applications during execution. This memory should be connected to the processor via a memory management unit (MMU) capable of managing virtual memory. This is required to support dynamic memory management and dynamic libraries, both of which are used extensively by Linux and Iris.

To allow remote access and easy data transfer to and from the development board, *Ethernet with IP networking* is needed. This simplifies transferring data between the board and a development PC and allows remote control of the radio operation. With Ethernet, no screen or terminal needs to be connected to the board as it allows using a remote shell over TCP.

The Linux root filesystem, where all the applications are stored, needs a mass storage device. Some embedded systems use a read-only memory (ROM) for permanently storing the kernel and the file system and load it into a random access memory (RAM) filesystem during boot. While this method requires the least resources, its disadvantage are the memory limitations of the ROM and that changes to the filesystem are not persistent between boots. Another method popular for embedded Linux systems is to provide the root filesystem over the network.

This is convenient for development but requires that the development board is continuously connected to the network. Users might design standalone cognitive radio systems which do not need a network connection. Therefore, *persistent local mass storage* is a requirement for this framework, e.g., a flash memory card.

In order to allow communication between these hardware items a *system bus* is required. Various different standards and solutions exist, for instance the IBM's Processor Local Bus (PLB), part of the Power architecture, which is used by Xilinx for embedded processor designs.

The above features are sufficient to boot the Linux operating system and to execute software applications such as Iris on the FPGA-based board. No direct user input and display devices need to be added to the requirements since this can be achieved through network terminal sessions or through automatic boot scripts. However, this may be added for convenience. Example user input/display devices would be a RS-232 serial port to connect to a terminal device or terminal emulation program on a workstation, or a VGA-port and keyboard combination.

The purpose of this framework is to integrate FPGA hardware components into the Iris-FPGA cognitive radio. This hardware portion of the system should be reconfigurable at run-time, either by changing operating parameters or by completely replacing the hardware components with different ones. They need to be connected to the system bus, so that Iris-FPGA can access and utilise them as part of a radio. Iris-FPGA must be able to control the configuration of this portion of the system, i.e., a way to reconfigure the FPGA from the Linux environment is required. Hence, *a software-reconfigurable FPGA logic fabric area* must be available in the system.

These features allow the implementation and execution of a wide range of cognitive radios, combining software and hardware as needed.

Software

The software environment for the proposed cognitive radio framework is managed by a modified version of Iris, i.e. *Iris-FPGA*. The existing version of Iris, as presented in Section 1.3, is only available for the Windows operating system. Thus, *Iris needs to be ported to Linux* to allow using it on the FPGA-based platform outlined above.

Linux is available in a huge number of different flavours and it has been ported to many different processor and system architectures. While this is an advantage and one of the reasons why it was chosen for the proposed framework, it also imposes difficulties for software applications. For instance, if one application can be compiled and executed on one flavour of Linux

for one processor architecture, this might fail for a different flavour/architecture combination. To avoid restricting the framework to only one processor architecture and flavour of Linux, Iris should be portable among a large variety of those. In fact, it would be beneficial to be able to compile the Iris-FPGA for common PC architectures as well, so that the software portion can be developed and tested easily. This also ensures that there is only one codebase for both the FPGA-based version and the PC-based version. Thus, *Iris-FPGA should be portable between different processor architectures and Linux flavours.*

A disadvantage of targeting FPGAs is that they are difficult to develop for without extensive experience in hardware design. It is even more difficult to manage the interface between hardware and software, as many low level details are required to manage this efficiently. For this reason, the second high level goal is to hide low level details from the user as much as possible. This means for Iris-FPGA that *the interface between hardware and software should be provided as part of the framework*, so that users do not need to know these details. Furthermore, *hardware components should provide the same interface to radio controllers as software components*, to allow experimentation with the cognitive part of the radio without special treatment of FPGA hardware components. And finally, *all FPGA configuration/reconfiguration should be managed by the framework*, to abstract the details of this process from the users.

Iris consists of many elements, as seen in Figure 1.1 on page 23. For Iris-FPGA, it has to be decided which parts should be executing on which hardware processing element to provide the highest efficiency. The following paragraphs assess the elements of the Iris architecture and evaluate their suitability for available processing resources.

Component library and data plane — A goal for porting Iris to an FPGA-based platform is to allow a combination of hardware and software processing components to work together, exploiting the vast processing power of FPGAs while keeping the flexibility of software where needed. The software component library is a set of directories containing shared libraries in the existing version. In the Linux port, it is straightforward to keep this feature, using Linux shared objects instead of Windows dynamic link libraries. Hardware components are effectively configurations of the FPGA logic fabric, represented by bitstream files for Xilinx FPGAs. These can also be held in a library, i.e., a directory on the Linux filesystem, so that Iris-FPGA can avail of them and configure the FPGA as needed. The data plane of the proposed framework will therefore *run on both the GPP and the FPGA logic fabric*, depending on the radio configuration.

Iris run-time system — It is convenient during radio development and testing to be able to control the execution of Iris and to edit the Extensible Markup Language (XML) configuration files from a development PC, connected to the board over the network. This avoids the necessity

to install an XML editor and parsing library on the embedded Linux system, which saves disk space and memory – valuable resources in embedded systems. The XML parser for Iris utilises the Xerces XML library (part of the apache project), which has a relatively large memory footprint. Additionally, maintaining all required XML radio configuration files on one central development PC ensures data safety in case of crashes of the development board. The intermediate representation generated by the XML parser as well as radio control commands are sent over the network to the FPGA board. Thus, the *launcher and XML parser are to be executed on the development PC*.

Radio controller — Typically radio controllers contain code that is difficult to implement in FPGA logic fabric. The controller events and parameter reconfigurations are easier to manage in software than in hardware. For instance, an iterative optimisation algorithm uses many conditionals and complicated control flow, something which is difficult to implement in hardware. For these reasons, the radio controllers should be implemented in software for execution on the GPP. If some advanced cognitive engine implementations need FPGA processing resources, they can still avail of through a separate custom interface to hardware. For instance, artificial neural networks can be implemented efficiently in FPGA logic. However, this is not anticipated for the near future, hence, mechanisms for using hardware in a cognitive engine not defined further.

All other software elements of Iris are not performance critical, i.e., it is not necessary to implement them in FPGA logic. They are therefore implemented on the embedded GPP.

Requirements summary

A summary of all hardware and software requirements for later reference is given in the following lists.

The chosen FPGA board needs to provide

1. a GPP with Linux port available,
2. memory with MMU,
3. Ethernet,
4. persistent local mass storage, and
5. an area of software-reconfigurable FPGA logic fabric.

The tasks and requirements for the software part of the framework are:

1. to port Iris to Linux,
2. to ensure source code portability between processor architectures and Linux flavours,

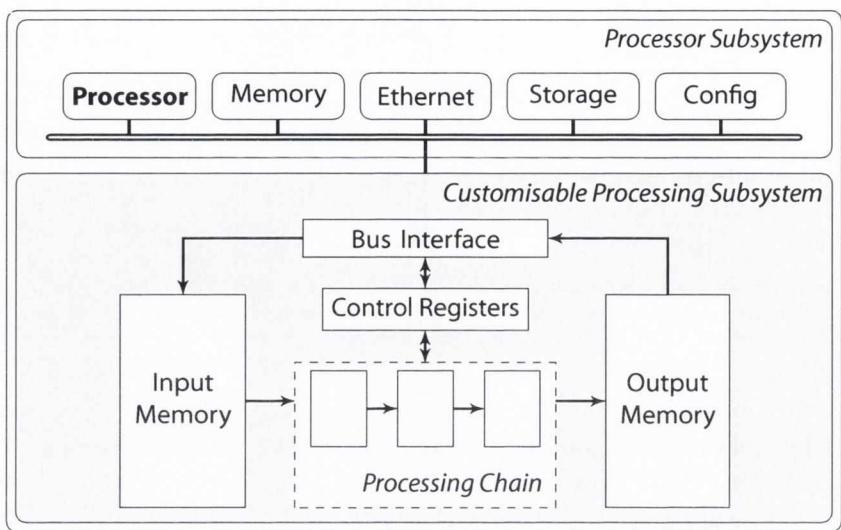


Figure 2.2: Virtual architecture for FPGA-based platform. All elements are concepts which have to be mapped to specific hardware elements on a real board.

- 3. to provide a hardware/software interface with Iris-FPGA,
- 4. to ensure an identical interface to radio controllers for hardware and software components,
- 5. to manage all FPGA configuration/reconfiguration within the framework,
- 6. to allow components to execute on either the GPP or the FPGA logic fabric,
- 7. to target the launcher and XML parser for the development PC, and
- 8. to target the radio controller and all other Iris parts to the GPP.

2.2.2 Virtual Architecture

In order to achieve hardware portability, it is beneficial to abstract from real hardware and define an architecture on a higher level. Therefore, it was decided to define a *virtual architecture* that can be mapped to various FPGA boards with ease. The major elements and concepts required for the proposed framework are contained in the virtual architecture, but not their specific implementation. For porting the framework to a specific board, the elements of the virtual architecture have to be mapped to the physical elements of the board. For instance, the *processor* element could be mapped to an embedded PowerPC 440 core. Figure 2.2 illustrates the proposed virtual architecture.

The two main tasks of the architecture are to run the Linux operating system for the execution of the software part of the framework, and to provide an area to hold radio components

in FPGA logic fabric. The virtual architecture was therefore designed to consist of two major parts, the processor subsystem (PSS) and the customisable processing subsystem (CPSS). The PSS is the part of the architecture which is responsible for running the Linux operating system and execute the software parts of the system. It is static hardware, but programmable in software. The CPSS is an area in the FPGA logic fabric which is used for hardware radio components or chains thereof. Its functionality can be changed during radio execution, utilising the dynamic partial reconfiguration capability of Xilinx FPGAs. To accommodate for different reconfigurable sections of a cognitive radio, the virtual architecture is designed to allow multiple CPSS connected to one PSS. Then each CPSS can be reconfigured independently, as might be required for a transceiver setup where the transmit and receive path are reconfigured individually.

As a special case for a mapping of this virtual architecture to a real system, the whole PSS might be outside of the FPGA device and one CPSS might incorporate one entire FPGA. That is, a development board could provide a GPP device separate from the FPGA itself, effectively having two separate processing devices on the same board. Then the external GPP can be used to execute Linux and the whole FPGA serves as a CPSS, holding hardware radio components. In that case, dynamic partial reconfiguration is not required since the device will always be reconfigured as a whole. This configuration does not violate the proposed virtual architecture which demonstrates its flexibility. However, in the designs and experiments for this thesis, both the PSS and the CPSS were on the same FPGA chip.

In the following, each element of the virtual architecture in Figure 2.2 is described in more detail.

The processor subsystem (PSS)

The PSS is designed to contain all the necessary hardware to run the full Linux operating system. The *processor* is the master of the system, it executes all the software code. To meet the requirements in Section 2.2.1, it may be any processor for which a Linux port exists. Access to memory is required in order to run Linux, thus, the PSS contains a *memory* element. It includes a MMU and some form of external memory, for instance, the DDR RAM situated on many FPGA development boards. To transfer data to other machines, execute a remote shell, and perform other networking, the PSS contains an *Ethernet*. This abstract element includes the PHY and MAC hardware for Ethernet, so that Linux can implement the remainder of the network stack for networking. To hold the root filesystem for Linux, a local mass storage device is required. To abstract the type of device used, the element of the virtual architecture has

simply been named *storage*. On a real device, this could be, for instance, flash memory or a hard drive. The PSS runs Linux, and on top, Iris. Iris-FPGA needs to be able to reconfigure the hardware in the attached CPSS portion of the system, in order to (re)configure the hardware components as needed. This is why the *config* element is part of the PSS. This abstracts a hardware element that is capable of (re)loading an FPGA bitfile (containing the configuration bits for the FPGA's logic fabric) into one of the attached CPSSs at run-time. On Xilinx FPGAs, this functionality is provided by the Internal Configuration Access Port (ICAP) core [57], capable of reconfiguring partially reconfigurable regions from within the FPGA. Some type of system bus allows communication and data transfers between all the elements of the proposed PSS architecture.

The customisable processing subsystem (CPSS)

Hardware processing components to be used with Iris, e.g., a modulator or a filter, are loaded and executed within one or more CPSSs of the proposed virtual architecture. After Linux is booted, these areas are empty. Configurations may be loaded as required during Iris-FPGA radio execution by utilising the *config* element of the PSS.

For using hardware components within Iris, they need to be accessed by software, control information needs to be exchanged, and data needs to be transferred. Transferring data between software and hardware is a slow process compared to direct data transfers in between hardware components, independent of transfer method chosen. Therefore, the architecture was designed to allow the connection of multiple subsequent hardware components into chains directly in hardware. This reduces the data transfer overhead significantly. Note that it is possible nonetheless to include only one hardware component per CPSS.

The communication between hardware and software can be accomplished in several ways. One way to exchange control and status messages is through the use of registers which can be accessed by both hardware and software. This method is ideal when small amounts of information are to be exchanged. The *control registers* shown in Figure 2.2 follow this approach, as controlling the hardware behaviour, e.g., starting and stopping the execution, only requires the exchange of small amounts of control information. Transferring the data to be processed is different, as this is performance-critical and typically involves large amounts of data. This transfer could be performed through registers as well, but using shared memory or shared first in, first out (FIFO) buffers are more suitable approaches. This is since with registers individual data items need to be transferred at a time, while the above concepts allow large blocks of data to be moved. Utilising FIFO buffers is efficient if direct memory access (DMA)

is used for the data transfer and interrupts are utilised for signalling the processor when the FIFO is full, empty, or when transfers are completed. Without these relatively complicated mechanisms, the use of FIFO buffers is inefficient since the software has to poll the hardware for the above information before and after transferring each data item. This is a significant overhead. Therefore shared memory was chosen as a transfer method, which is the simplest to implement while still providing good performance.

The shared memories are embedded memory blocks in the FPGA fabric and can be accessed by software as well as hardware (the *input memory* and *output memory* blocks in Figure 2.2). The highest performance can be achieved by using DMA hardware to manage the data transfers from main memory to the embedded memories and back, so that the processor is free for other processing jobs. Alternatively, data can be transferred by the processor, copying the data between both memories directly. This proposed virtual architecture does not restrict the transfer method, the decision whether using a DMA is suitable can be postponed until mapping the architecture to a real board.

Once the input data has been transferred to the input memory, the hardware chain needs to be signalled that it can now process the data. This is achieved through writing a start signal into a control register. The hardware then starts processing the data and stores its output data in the output memory. Another control register is set by the hardware, to signal the software that processing is finished. Now the data can be read back. A requirement stated that hardware and software component should provide the same interface (see Section 2.2.1). This means, hardware components must also be allowed to expose parameters and to trigger events. This is solved by using other control registers for this information exchange. Some control registers can be used to set hardware parameters and others can be written by the hardware to signal an event.

In order to connect the CPSS to the system bus in the PSS, the architecture contains a *bus interface* element. It implements the bus protocol required and routes the data transfers between the processor and the hardware. Often this is available as a standard core provided by the FPGA vendors.

Note that the virtual architecture proposed here is not restricted to using interrupts or DMAs on a real board. Rather this can be decided depending on the target board, code availability, and the affordable implementation effort.

2.2.3 Full System Implementation

With the requirements (Section 2.2.1) and virtual architecture (Section 2.2.2) in place, the design was ready to be implemented on a real system. This section illustrates the main issues associated with the implementation of Iris-FPGA and outlines the chosen solutions.

Porting Iris to Linux

Although an official standard for the C++ programming language and the provided standard library is available [58], few compilers implement it in full. Additionally, many compilers and standard library implementations add to the standard and implement additional functionality. The standard does not define language elements or library functions for many common tasks, for instance, threads, shared libraries, filesystem access, etc. This functionality is therefore offered by operating system application programming interfaces (APIs), in a different way for each operating system. The POSIX family of standards [16] defines a standard operating system interface and environment, including an API that can be called from C++. Most Linux and Unix flavours, MacOS X, and many other operating system conform largely to this POSIX standard. Windows does not. All the above implies that porting C++ code written for the Windows operating system to Linux is not straightforward. In the following, the major issues during porting are outlined, giving solutions where possible.

The Windows API defines many non-standard data types, e.g., BYTE, WORD, or DWORD, which have been used extensively throughout Iris. Additionally, some simple functions defined in Windows do not exist in the standard or in Linux, e.g., `itoa` for converting an integer number into a string. As a simple solution, a header and source file was implemented which defines the Windows data types and functions in Linux, replacing them with the standard or POSIX equivalents or implementing them from scratch. This approach saved a large amount of time and allowed for gradual source code migration to fully portable code.

Some parts of Iris use threads to accomplish concurrent execution of different sections of the program. For example, the user interface uses a separate thread to allow a fast response to user inputs while the radio is executing. The threading API is very different in Windows and POSIX systems, although the concepts behind the functions are similar. Therefore, all Windows threading functions had to be replaced with their POSIX equivalents.

The Iris software components are implemented as shared libraries (`.dll` in Windows, `.so` in Linux). They are loaded from a repository, i.e., a directory in the filesystem, at run-time. Windows and POSIX use a completely different way of exporting and importing functions

from external libraries, and for loading and unloading them from an application. Thus, the library management functions and interface definitions had to be reworked completely.

The Windows version of Iris defined an interface to Java, for implementing controllers or for calling Iris as a whole from within a Java application. For simplicity and also due to the fact that this feature was rarely used, it was decided to omit this part in the Linux port.

One requirement for Iris-FPGA is that the codebase is portable between different flavours of Linux and different processor architectures, as stated in Section 2.2.1. In fact, the Linux port of Iris should be compilable and executable on a standard PC as well as on an embedded GPP on an FPGA board. The flavours of Linux use different versions of libraries, different compilers, and might differ in other points. To overcome these issues, it was decided to use the GNU Autotools [59] to manage the build system. This is a set of tools which generates a script to test for the capabilities of the build environment, including compilers, libraries, and other features. The script generates specific build files for the current environment automatically, based on generic rules specified by the developer. Furthermore, it can be set up for *cross-compilation* with ease. Cross-compilation is the process of compiling sources on one architecture for execution on another. For instance, the Iris-FPGA sources can be compiled on a standard PC for execution on the embedded processor on the FPGA board. This avoids native compilation directly on the target system, which is a very slow process due to the limited resources and processing power. The Autotools are very common in the Linux/Unix world as they allow to write portable source code that can be compiled for various architectures and operating systems. This portability allowed to port Iris to Linux using a standard PC; using the embedded GPP on the FPGA board was not necessary in the process.

With all the above issues solved, it was possible to create a working version of Iris for the Linux operating system. As an aside, this port also enabled the exploration of other hardware platforms and operating systems with Iris by fellow researchers, for instance, the CellBE (using Sony's Playstation 3 with Linux), or executing Iris on MacOS X.

Hardware interface

To allow to connect hardware components together to a chain within a CPSS, a hardware processing component interface had to be defined. The following simple interface has proved efficient for the task. All components define a data input and output wire, with a data type and a width (the number of bits) defined. For flow control, i.e., to be able to slow down or stop the computation, additional signals similar to a FIFO are required. A 1 bit valid signal

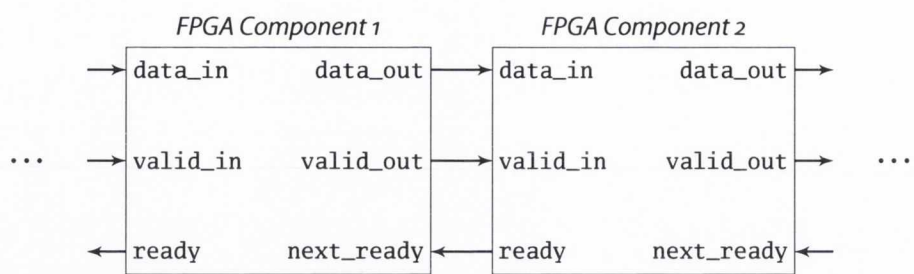


Figure 2.3: The Iris-FPGA hardware component interface with data and flow control signals.

travels with the data to inform components down the chain whether the incoming data is valid and can be processed. Additionally, to allow slower components to be connected to faster components, each component provides a 1 bit ready signal to the previous component. The previous component must make sure not to pass on any data if the ready signal of the next component is low. Thus, it might be required to set the previous component's own ready signal low, so that previous components cannot feed new data in. This simple interface is illustrated in Figure 2.3.

A simple example is a scaling component. When it receives a valid input sample, i.e., when the `valid_in` signal is high, and when the next component signals that it is ready to receive data, i.e., when the `next_ready` signal is high, it multiplies the input sample by a scaling factor and provides it at the output, along with a high `valid_out` signal.

As another example to demonstrate the use of the flow control signal, consider an upsampler component with an upsampling factor of 4. No actual processing in the component is necessary, the function can be realised by manipulating only the flow control signals. When the `next_ready` is high, i.e., the next component down the chain is ready to accept new data, the component consumes one element from its input and produces it directly on the output. Then it sets its own `ready` signal to low for three clock cycles to ensure the previous component does not produce new data. During these three clock cycles, it outputs zeros. Then this procedure is repeated. This results in upsampled output data with three zeros inserted in between original samples.

Two standard hardware components are provided for reading from the input memory and writing to the output memory. There are several variants of these components, for example for reading 16 bit integer values, or for reading the data bitwise. One variant is used in every hardware chain to provide access to the input and output memories.

Hardware / software interface

As a first working implementation for the hardware / software interface, it was decided not to use DMA or interrupts, but use a simple data copying and polling method. To allow that, it is necessary to map the memory address range of the input and output memories and the control registers into the address space of Iris-FPGA (a method called memory mapping). This allows accessing the memories directly. In Linux, the generic memory access driver `/dev/mem` can be used to achieve this. It allows accessing the whole physical memory of the system by mapping areas into the application. As a result, the software is given a pointer to the embedded memories/control registers in the FPGA fabric, which it can read from and written to as if it was standard memory.

If data is to be written to the hardware for processing, it is copied to the mapped address of the input memory. Then the hardware processing is enabled by setting the corresponding control register accordingly. The software then polls another control register which indicates whether the hardware is ready, while the hardware is performing its computation. Once ready, the software can read back the data from the output memory into a buffer in the system memory.

Using polling instead of an interrupt to find out when the processing is finished turned out in various tests to be an insignificant overhead, not affecting the overall performance. This is due to the hardware processing being typically orders of magnitude faster than processing on an embedded GPP. In our experiments, the software never polled the hardware more than once before the hardware indicated that it was ready. When highly complicated hardware designs are used, the processor might need to poll multiple times before the hardware is ready. In that case, interrupt-driven signalling can be implemented. However, this is very unlikely since the hardware is several orders of magnitude faster than the embedded processor.

The performance bottleneck is in the transfer of data in and out of hardware. The processor is involved moving every data item from main memory into the hardware and back. It is far more efficient to use a hardware DMA block to move the data, but also more complicated to implement. It was therefore decided to leave this as a future improvement in favour of testing the architecture and framework with a few experiments before.

Selecting an appropriate FPGA development board

To make the system more compact and to demonstrate the use of dynamic partial reconfiguration of FPGAs, it was decided to use a single system-on-a-chip (SoC) design using a Xilinx

Virtex FPGA for this framework. That is, all elements of the virtual architecture shown in Figure 2.2 are mapped onto a single chip. The Virtex line FPGAs are the high-end solution offered by Xilinx, compared to the low-cost Spartan line, and the design tools to utilise dynamic partial reconfiguration are readily available for Virtex FPGAs. Some types of the Virtex line of FPGAs provide a PowerPC GPP on the chip, targeted for combined hardware and software systems. The PowerPC is a perfect choice for an embedded GPP for this framework since Linux ports are readily available for the Power architecture and the PowerPC is a faster processor than the MicroBlaze, a soft core which could be instantiated on any Xilinx FPGA.

Any development board with a Virtex FPGA containing a PowerPC as well as the hardware for Ethernet, external memory, and local mass storage is suitable for this framework. Practical reasons (e.g., cost), and the availability of a reference design for Linux led to the choice of the Xilinx University Programme (XUP) Virtex-II Pro development board [60], which is available to academia at a low cost. This was meant as a proof of concept for the framework; once several experimental setups were demonstrated (see Chapter 3), the framework was moved to an ML507 board with a much larger and much faster Virtex 5 FPGA [61]. For the remainder of this chapter, the focus is on the XUP board, although most of the concepts and issues mentioned also apply to the ML507 board.

Both the XUP Virtex-II Pro and the ML507 fulfil the hardware requirements mentioned in Section 2.2.1. Both have DDR memory, Ethernet (100 Mbit/s on the XUP, 1 Gbit/s on the ML507), local mass storage provided by a compact flash card, the Xilinx ICAP core for configuring the FPGA, and a system bus. Thus, they are suitable platforms for this framework.

The system bus is IBM's PLB bus, part of the Power Architecture and readily available from Xilinx. For the XUP board, the Virtex-II Pro FPGA uses an additional type of bus, the On-Chip Peripheral Bus (OPB). This is bridged to the PLB bus and connects to the compact flash card and the CPSS. Interfacing with this bus is significantly simpler, saving considerable amounts of FPGA area for the *bus interface* element of the CPSS (see Figure 2.2). Due to the limited area on the Virtex-II Pro FPGA and the simpler implementation, it was decided to connect the CPSS to the OPB bus on the XUP board. The performance difference between PLB and OPB was irrelevant for the first experiments performed on this board. When the framework was moved to the ML507 board, with its faster and more spacious Virtex 5 FPGA, the CPSS was connected directly to the PLB bus.

Table 2.1 shows the mapping of the virtual architecture elements from Figure 2.2 to actual hardware components on the XUP and ML507 FPGA boards.

With the reference design for the XUP board, the PowerPC processor executes at a clock

Virtual Architecture	XUP	ML507
Processor	PowerPC 405	PowerPC 440
Memory	DDR2 SDRAM	DDR2 SDRAM
Ethernet	10/100 Ethernet MAC	10/100/1000 Ethernet MAC
Storage	SystemACE Compact Flash	SystemACE Compact Flash
Config	ICAP	ICAP
System Bus	PLB/OPB	PLB

Table 2.1: Mapping of virtual architecture elements (Figure 2.2) on XUP and ML507 boards. SystemACE is a controller for a Compact Flash card, ICAP is the Internal Configuration Access Port controller, used for dynamic partial reconfiguration of the FPGA, PLB is the Processor Local Bus, and OPB is the Onboard Peripheral Bus.

speed of 400 MHz, while the system bus and all peripherals are clocked with 100 MHz. This means that all the hardware components to be used within the CPSS are required to support a clock speed of 100 MHz.

Booting Linux on the FPGA board

For booting Linux on the FPGA board, it is required to cross-compile the kernel with the right configuration and drivers and to setup a root filesystem. A cross-compiler toolchain is needed, which can be used to compile code for the PowerPC architecture but execute on a standard PC.

A cross-compiler was built using Kegel's crosstool [62], a commonly used tool for building cross-compilers for various architectures. The kernel sources and configuration options were determined with help from Xilinx, various web resources, and through testing. A Xilinx-modified kernel was used, version 2.6.22, obtained from the Xilinx git tree [63]. This a standard kernel with extra drivers for Xilinx cores and devices. As a reliable root filesystem with support for many tools and libraries, Debian Linux was chosen, which comes with a convenient package manager (aptitude) for installing software as needed. The storage space on the Compact Flash card was chosen to be 2 GB, relatively large for an embedded system, to be able to install many diagnostic tools and libraries for development and debugging. If the framework was to be optimised for size in the future, the filesystem could be replaced with a smaller alternative, e.g., Busybox [64].

Connecting a radio front-end

A radio front-end is responsible to transform the digital baseband data to analog RF signals and transmit them wirelessly, as illustrated in Figure 2.1 on page 38. There is a large variety of front-ends available today, however, many of which are very expensive and offer features that are not needed for the first version of this framework. To demonstrate the capabilities of this framework and to run a few experiments, it is enough to transmit relatively narrow band signals (below 10 MHz) across distances of a few meters. Noise figures are not very important, as long as the signal quality allows reliable reception in a lab environment.

For these reasons it was decided to use the cheap and off-the-shelf radio front-end *Universal Software Radio Peripheral (USRP)* [65]. For transmission, the digital baseband data is transferred to the USRP using Universal Serial Bus (USB) 2.0, where it is interpolated to a higher sampling rate, filtered, converted to analog, up-converted to the desired transmission frequency band, and transmitted over an antenna. At the receiver, the signal received from the antenna is down-converted, filtered, converted to complex digital baseband data, decimated, and sent to a connected computer via USB 2.0. The digital processing and USB interface of the USRP is performed on the digital baseband board and the analog part on a daughterboard. This daughterboard is targeted to a specific frequency range and can be replaced if another band is required. This high flexibility, the simple connectivity, and the low price (below USD 1,000) led to an extensive use of the USRP in the cognitive radio research community. It is also used as the default front-end for this framework.

A new version of the USRP was released in 2009 which uses gigabit Ethernet instead of USB for connecting it to a host computer. This increases the maximum transmission/reception bandwidth of the device. The USRP 2 was not available at the time this cognitive radio framework prototype was implemented and its higher bandwidth was not needed for the experiments conducted (presented in Chapter 3).

Unfortunately, connecting the USRP to the XUP FPGA board is not easily possible since the board does not offer USB 2.0 connectivity. Ideally, a custom FPGA hardware component could be developed, which communicates directly with the USRP for configuration and data transfer. However, this involves a considerable amount of development time and has an uncertain outcome. Given the fact that a transition to a more recent FPGA board was planned (the ML507), which supports both USB 2.0 and gigabit Ethernet, it was decided to use a workaround for connecting the radio front-end in the first prototype.

The USRP is connected to a USB port of a laptop which routes the data to/from the FPGA

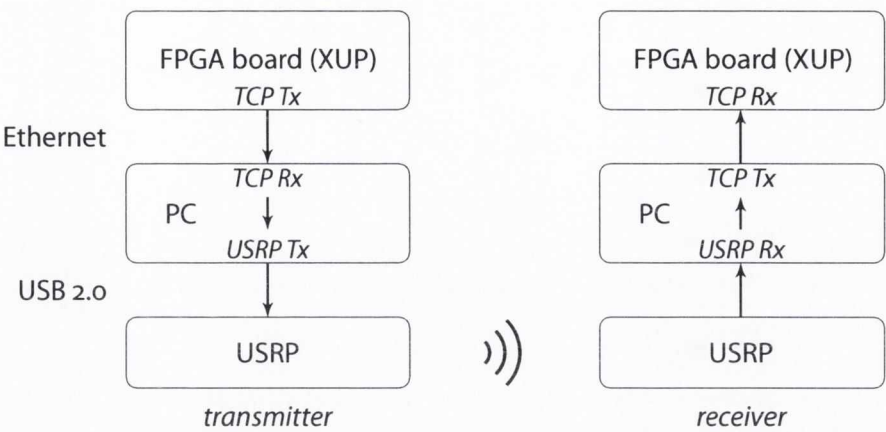


Figure 2.4: Illustration of the workaround to connect the USRP front-end to the FPGA board.

board via the Ethernet connection. Iris is used on the laptop, with a software component for sending the data over a Transmission Control Protocol (TCP) socket connection to the board, where a TCP socket receiver component is the first in the Iris-FPGA chain. The reverse is done on the transmitter side. The PC controls the USRP and acts as a router for the digital baseband data to and from the FPGA board. The setup for this workaround is illustrated in Figure 2.4. This workaround is easy to accomplish and it allows to focus on the framework aspects on the FPGA board.

2.2.4 Abstracting Low-Level Details

The second high-level goal of this framework is to allow radio designers without experience in FPGA design to build and design cognitive radio systems (see Section 2.2.1). This is accomplished by hiding implementation details and providing simple interfaces and components which can be used “out of the box”. The two primary enablers for this are the concept of a *software wrapper* component, and high-level tools which assist with the composition of hardware components to radio chains and the generation of software wrapper code. In the following sections, the details of these enablers are outlined and possible future improvements are discussed.

Software wrapper component

As a solution to manage the interfacing with the FPGA hardware, the concept of a *software wrapper* is proposed. This is a standard software component for Iris-FPGA which manages all interfacing to the FPGA logic fabric. All parameters and data inputs and outputs of the

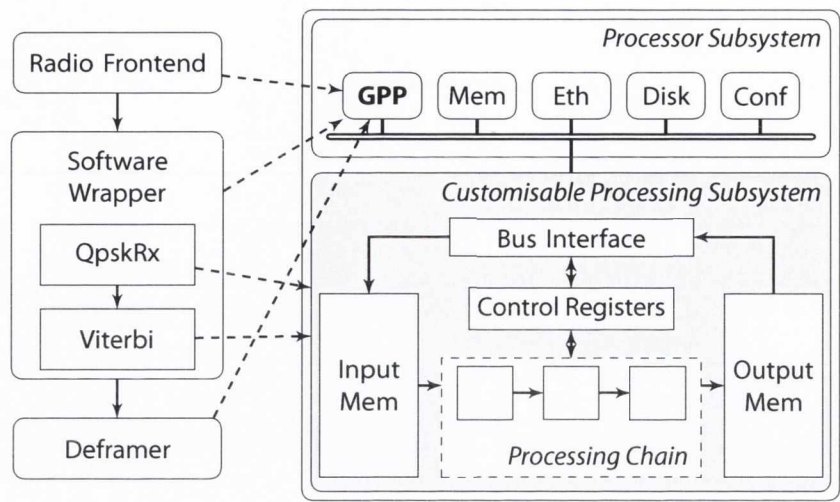


Figure 2.5: The software wrapper concept for an FPGA-based narrowband receiver. The QPSK Rx and Viterbi decoder components are executing in hardware (FPGA logic fabric). A software wrapper component manages the hardware/software interface.

software wrapper component are routed to hardware, encapsulating all low-level tasks. The software wrapper component behaves like an ordinary software component on the outside but performs computations in the FPGA fabric. Figure 2.5 illustrates the software wrapper concept using the example of a simple narrowband receiver system.

One software wrapper manages one CPSS on the FPGA. During initialisation, the CPSS is configured with the appropriate bitfile using the *config* component in the PSS. The wrapper component exposes parameters as every other Iris-FPGA component and handles the settings of the appropriate hardware control registers to reflect the parameter values. When data arrives at the input of the wrapper, it is copied into the input memory in the CPSS and hardware processing is enabled (through a control register). A control register is polled to find out when the hardware is finished, so that the data can be read back into software from the output memory. Furthermore, the control registers are checked for hardware events aimed for the radio controller. If such an event occurred, a corresponding standard Iris-FPGA software event is triggered by the software wrapper, enabling the controller to obtain information from hardware. If the controller wants to reconfigure a hardware component, this can be achieved by changing a parameter value of the software wrapper. This translates the parameter change into a change in a hardware control register, or, when required, reconfigures the whole CPSS with a new FPGA configuration file.

Thus, the proposed software wrapper component fully encapsulates the interface to the FPGA logic fabric. The remainder of Iris-FPGA is completely ignorant of hardware and treats

the wrapper as a regular software processing component. This not only reduces the implementation complexity, it also ensures that the interface of a radio to the radio controller remains the same, no matter whether components run in software or in hardware. Thus, radio designers can compose their radios from hardware and software components from the component library and write radio controllers to manage the decision-making about reconfigurations without the need to understand hardware details. Their experience is mostly consistent as with a pure software implementation.

Hardware composition tool

Due to the fact that one CPSS can hold a chain of hardware components, software wrappers might represent more than one component. To allow easy connection of hardware components from the component library into chains and to assist the creation of appropriate FPGA configurations, high-level tools are beneficial.

The hardware components in the component library are available in source code, in some hardware description language (HDL). They implement the interface shown in Figure 2.3. Each component has some meta information about the data types at its input and output ports and the supported parameters and events. This information can be used by an automated tool to generate the HDL code for a full CPSS based on a high-level block diagram of radio components. Furthermore, the software wrapper code can be generated automatically as well, ensuring that the right events and parameters are supported and the inputs and outputs have the right data type. The rest of the software wrapper code is similar for most FPGA chains and can therefore be generated automatically.

Fragments of this automatic hardware composition tool have been implemented, using XML to describe the component meta information and the hardware radio chain, and a tool written for transforming that into Verilog code for the hardware, and C++ code for the software wrapper. The radio configuration is specified in an XML format similar to the format used for Iris, and the composition tool reads this to find out about the required components and their connections. The generated Verilog instantiates all hardware components and connects their input and output ports together, as illustrated in Figure 2.3. It can then be given to the Xilinx synthesis tools to generate a bitfile for the configuration of the FPGA's CPSS. The software wrapper is generated by using a template file and substituting code with information taken from the radio chain description and component metadata. Extensible Stylesheet Language Transformations (XSLT) was chosen as a language to implement this, as this is a language designed to transform XML files into some other textual representation. It is therefore ideal for

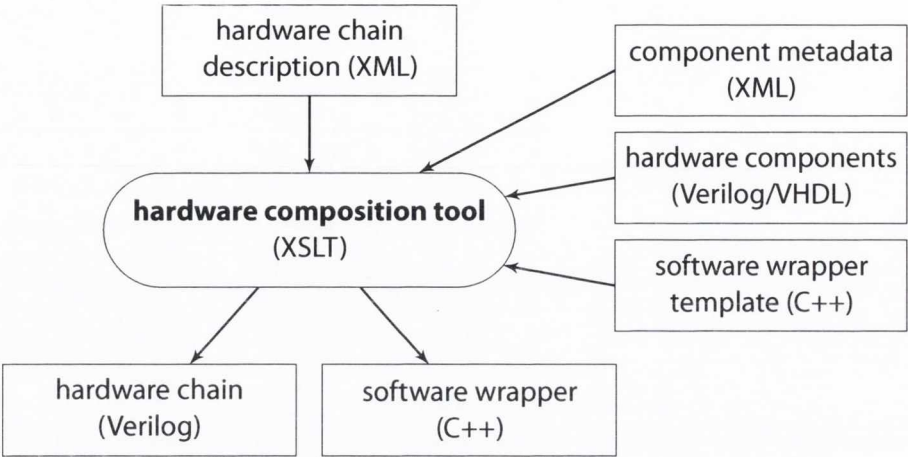


Figure 2.6: XSLT hardware composition tool. The component metadata, component code, and software wrapper template are provided with the composition tool. It takes an XML description of the radio chain to be implemented in hardware and generates the Verilog and Iris-FPGA software wrapper code automatically.

the tasks of this composition tool. Figure 2.6 illustrates the hardware composition tool along with the inputs and generated outputs.

This tool has been used during the design of the experiments described in Chapter 3. It is still work in progress, which is why some manual modifications to adapt the generated code were required. Once it is finished, radio designers do not need to write any HDL code or software wrapper code and are still able to compose arbitrary radios based on the components available in the library, no matter whether they are software or hardware components.

2.3 Summary

In order to develop a capable and flexible cognitive radio experimentation framework, this chapter has evaluated potential hardware processing platforms for its implementation. Typically, flexibility contradicts performance with these platforms, so trade-offs have to be carefully chosen. As a solution, a platform combining a GPP for software and an FPGA for high performance hardware accelerators has been proposed.

This platform requires a combination of a software framework and a hardware architecture to function as a union. For the software part, the Iris cognitive radio framework has been chosen. This has been reviewed in detail as it forms a basis for the framework developed.

The FPGA-based cognitive radio framework Iris-FPGA has been proposed which combines the flexibility of software with the performance of FPGAs. The concepts and ideas of Iris have

been ported to this platform, and extended to include processing components implemented in the FPGA's logic fabric. Software and hardware components can be combined in a single radio, leveraging the advantages of both domains. Through the use of the dynamic partial reconfiguration feature of Xilinx FPGAs, both hardware and software can be reconfigured in the same way by radio controllers. These controllers implement the cognitive cycle of observe, decide, and act. No special treatment for hardware components is required, simplifying the controller implementation significantly for the user. The low-level implementation details for handling hardware are completely handled by the Iris-FPGA, hidden from the radio designer. Furthermore, thanks to defining a virtual architecture for the proposed framework, it can be ported to various different FPGA development boards without difficulty. Thus, the proposed framework allows implementing cognitive radios requiring both high performance and high flexibility with relatively little effort.

In the next chapter, several experiments are conducted to test the proposed Iris-FPGA framework in various setups. Important aspects of cognitive radios are investigated to evaluate the capabilities of the framework.

3 Experiments with the FPGA-based Framework

Contents

3.1 Simple Audio Transmitter Design	65
3.2 Video Streaming Link with Spectrum Sensing	69
3.3 Video Streaming Link with Adaptive Coding	79
3.4 Combining Adaptive Coding and Spectrum Sensing	84
3.5 Summary	91

The field-programmable gate array (FPGA)-based cognitive radio framework Iris-FPGA, proposed in the previous chapter, is employed in a series of tests and experiments in this chapter, evaluating its capabilities and usability for real cognitive radio setups. The purpose of the experiments is to test various different aspects of Iris-FPGA using example radios. The radios are chosen to show key aspects of cognitive radios in relatively simple configurations, i.e., demonstrating the capabilities of the proposed Iris-FPGA framework.

Recall the definition of cognitive radios from the introduction. They can be viewed as radios with meters and knobs that execute the cognitive cycle of *observe*, *decide*, and *act*. Therefore, these concepts must be supported by a cognitive radio experimentation framework. The demonstrators developed in this chapter are designed to show these aspects for the Iris-FPGA framework, as well as the capability to run basic radios.

The first demonstrator consists of a simple software defined radio (SDR) running on the Iris-FPGA platform without any meters, knobs, or decision-making process. Although this is not a cognitive radio according to the definition, this functionality is a basic requirement for

all other setups and should be tested first. Thus, the first demonstrator developed is a basic narrowband radio transmitter on the proposed platform, streaming audio data across to a receiver. This is implemented using the Iris software on a Windows PC to confirm that the data can be received correctly. This demonstrator is presented in Section 3.1.

In the second demonstrator, both the transmitter and receiver are implemented using the proposed framework, including meters, knobs, and a decision-making process. Video data is streamed from a narrowband transmitter at a carrier frequency unknown to the receiver. This employs spectrum sensing techniques to search for the signal in order to receive it and display the video. In this setup, the adjusted knobs are parameters of software components, not affecting the FPGA configuration. Section 3.2 presents this demonstrator.

To demonstrate the dynamic reconfiguration capability of the proposed Iris-FPGA platform, a demonstrator that adapts components implemented in FPGA hardware was developed. It consists of a video streaming narrowband link with convolutional coding whose coding strength can be adapted to the current channel conditions automatically. This involves a replacement of the decoder hardware component at the receiver. This demonstrator is discussed in Section 3.3.

To integrate all the previous demonstrators and to show a more complex radio configuration, the final demonstrator combines the adaptive coding and the spectrum sensing experiments. This demonstrator changes its coding scheme according to the current channel conditions and detects the transmitter's frequency without prior knowledge using spectrum sensing techniques. The decision-making process is more complex than for previous demonstrators, and a wider variety of FPGA configurations have to be managed. This setup demonstrates multiple meters and knobs, both on hardware and software components, and a more complex decision-making process. It is presented in Section 3.4.

These demonstrators were implemented with the help of two other researchers. Suhaib A. Fahmy, a post-doctorate researcher, was mainly concerned with the implementation and tests of the hardware processing components on the FPGA. Juanjo Noguera, a member of the Xilinx Research Labs, was mainly concerned with the FPGA setup and configuration, and integrating the various hardware elements into a working system. The majority of tasks for designing and implementing these demonstrators, including the concepts, radio chains, algorithms, controllers, and software architecture, were mainly performed by the author of this thesis.

The presented demonstrators have been presented at international conferences to obtain valuable feedback [66–71]. This and the experience and knowledge gained from develop-

ing the demonstrators helped to identify common challenges for cognitive radio design for experimentation in Chapter 4.

3.1 Simple Audio Transmitter Design

In order to test the basic radio transmission capability of the platform, without meters, knobs, and decision-making, the first experiment is a simple narrowband transmitter. This is not strictly a cognitive radio but all cognitive radio setups require functional baseband processing.

The transmitted data is uncompressed audio. The receiver side is implemented fully in software, using the Windows version of Iris, which has been tested for several years. Since no error correction, buffering, interleaving or similar methods are applied, the received signal is likely to be effected by bit errors. However, the audio should still be easily recognisable. The experiment is considered successful if the audio is played back on the receiver laptop in a reasonable quality.

This section describes the experiment setup, the signal processing chains running in the transmitter and receiver radios, and summarises the results.

3.1.1 Experimental Setup

The transmitter of this demonstrator is built on a Xilinx University Programme (XUP) Virtex-II Pro FPGA development board, using one of the two embedded PowerPC processors for running the Linux operating system as well as the Iris software. The processing intensive radio components are implemented in the FPGA's logic fabric and the remaining components are software, executed on the PowerPC. The Universal Software Radio Peripheral (USRP) [65] radio front-end is used to transmit the data to the receiver. To keep the first demonstrator simple, the receiver is implemented fully on a standard PC running the Windows operating system and the Windows version of Iris. Figure 3.1 illustrates the experiment setup.

The USRP had to be connected to a PC due to the lack of a Universal Serial Bus (USB) 2.0 connector on the XUP board (see Section 2.2.3 for details). The PC is routing the baseband data from the XUP board to the USRP using a simple Iris radio. This workaround has no effect on the demonstrated cognitive radio concepts, it is a method to avoid the considerable implementation effort for a direct interface from the FPGA to the USRP.

Since the purpose of this experiment is to demonstrate that Iris-FPGA can indeed be used to transmit simple waveforms, a simple pulse-shaped differential binary phase shift

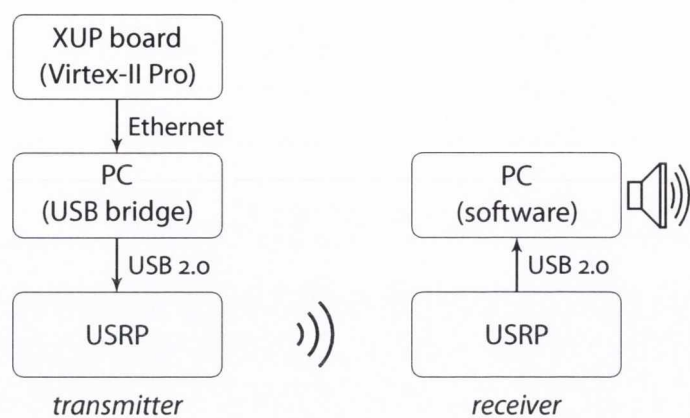


Figure 3.1: Setup for the audio transmission experiment. All digital baseband processing is performed on the XUP board, the PC is only routing the baseband data from the XUP to the USRP.

keying (DBPSK) waveform with simple framing was chosen. This modulation is real-valued in the digital baseband, which simplifies the hardware implementation significantly at the transmitter. Differential modulation simplifies synchronisation at the receiver and avoids training symbols. Only physical layer functionality without forward error correction (FEC) is implemented. This has the advantage that the quality of the physical channel becomes immediately apparent when listening to the received audio.

The audio itself is encoded as simple 16 bit pulse code modulation (PCM) samples, with the sampling rate adjusted to exactly fit the physical layer data rate. This ensures that no buffering is required and the streaming is real-time.

3.1.2 Radio Configuration

The radio configurations applied at the transmitter and receiver are shown in Figure 3.2.

First, the transmitter has to read the audio data from a file existing in the Linux filesystem (on the compact flash memory card). The *file reader* component was chosen for this task. Then, the data is embedded in a frame structure by a *framer* component. The simple frame structure chosen for the initial case study starts with a frame check sequence, so that the receiver can synchronise with each frame, followed by the payload length, the whitened payload and a checksum field. The payload is whitened by computing a bit-wise exclusive or with a pseudo random sequence, known at the receiver. This ensures that the data sent over the air appears random. This simplifies synchronisation at the receiver. To allow the receiver to detect whether there were errors in the reconstructed data, the frame contains a checksum field. With this, the receiver can output some diagnostic output, including the frame error rate (FER). To simplify

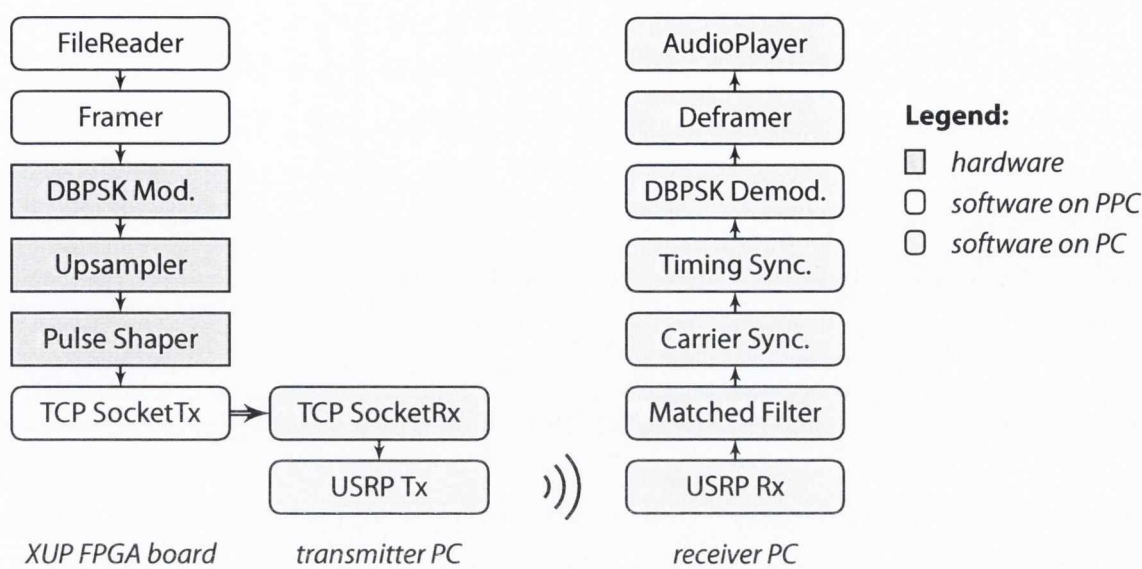


Figure 3.2: Radio configurations for the audio transmission demonstrator.

demodulation, DBPSK modulation was chosen for this demonstrator. This is performed by the *DBPSK modulator*, encoding a binary ‘1’ as 180° phase shift and a ‘0’ as no phase shift, relative to the previous output sample. To limit the spectral footprint of the signal, and, more importantly for this demonstrator, to enable higher signal to noise ratio (SNR) values at the receiver, each modulated symbol is shaped by a *pulse shaper*. This involves upsampling the data by a factor four (inserting 3 zeros between samples) by the *upsampler* and then filtering it with a finite impulse response (FIR) filter with an approximate root raised cosine impulse response. The filter is in fact an interpolation filter, smoothing the symbol transitions and therefore reducing the occupied spectrum. The data is then transferred to the transmit PC using TCP sockets over Ethernet, which routes it to the USRP via USB. This converts the baseband data to analog, up-converts it to the desired carrier frequency, and transmits the signal over the attached antenna.

The computationally intensive parts of the transmitter radio chain are the modulator, upsampler, and pulse shaper. Therefore it was chosen to implement these in the logic fabric of the FPGA. Accessing the Linux filesystem from hardware is a complex task, but this can be performed easily in software. The same is true for sending the data over a Transmission Control Protocol (TCP) socket. Therefore these components in software were chosen to be implemented in software. The framer component was implemented in software as well since its tasks are simple in software and there would not be much benefit if it was realised in hardware. Additionally, the data after the framer can be treated as a continuous stream of bits while

the framer itself operates on larger blocks of data. Streams of bits can be easily and quickly processed in hardware, while block-wise processing is more complicated and is typically slower.

The receiver configuration is fully implemented using Iris on a Windows PC. The components existed in the Iris component library before and were extensively tested in a pure software setup (where both the transmitter and receiver were running on Windows laptops). The receiver is used to validate the transmitter setup; its configuration is shown on the right in Figure 3.2.

After receiving signal samples from the radio front-end by the *USRP receiver* component, the *matched filter* filters out the desired signal, suppressing noise and therefore increasing the SNR. Typically the local oscillator in the receiving USRP is out of tune by some margin, introducing phase and frequency shifts into the signal. The *carrier recovery* reverses these effects by employing a Costas loop-like phase-locked loop (PLL) [72, 73]. To recover the modulation symbols from the signal samples, the optimal sampling points are estimated by the *timing recovery* component, employing a modified Mueller and Muller algorithm [74]. The *DBPSK demodulator* converts the complex baseband samples into bits, depending on the phase transitions between samples. In order to synchronise frames, the *deframer* correlates the bits with the frame check sequence. When the beginning of a frame is found, the payload is de-whitened, i.e., another bitwise exclusive or is computed with the same pseudo-random sequence used by the framer in the transmitter to obtain the original data. This is then converted to 16 bit audio samples, and a checksum comparison is performed to detect whether the frames have been received correctly. Finally the data is transferred to an audio player and the music is played.

The quality of the received audio can be assessed in two different ways. As a numerical metric, the frame error rate reported by the deframer component can be monitored. Additionally, the quality of the played audio can be assessed by listening to the music, which is a subjective measure but nevertheless useful for a demonstrator.

More design and implementation details of this demonstrator are given in Appendix C.1.

3.1.3 Summary and Discussion

An audio transmitter was successfully designed using the proposed FPGA-based cognitive radio framework. It was successfully demonstrated during the CISDR workshop [66], and presented at the ISSC conference [67]. This shows that the platform is capable of running simple SDRs incorporating software and hardware processing components in a single radio.

Most of the FPGA resources are still available, leaving enough space for more complicated setups.

The software wrapper was implemented manually for this demonstrator. From analysing the code became apparent that most of the same code is needed for all software wrapper implementations, with the only differences in the used data types, parameters, events, and the number of items consumed and produced during one execution of the hardware chain. Thus, most of the software wrapper implementation can be automatically generated, which abstracts low-level hardware details from the user of the framework. This led to the development of the composition tool outlined in Section 2.2.4.

3.2 Video Streaming Link with Spectrum Sensing

The second demonstrator was chosen to test the platform's capabilities for cognitive radios. That is, it *observes* by retrieving information about the radio's state and environment (meters), *decides* by performing rule-based decision-making, and *acts* by adjusting the radio's operating parameters at run-time (knobs). Thus, this demonstrator executes the cognitive cycle.

It was decided to design a radio link employing a simple dynamic spectrum access (DSA) technique. *Dynamic spectrum access (DSA)* is a paradigm which focuses on new and dynamic methods for managing and accessing spectrum. This is in contrast to the traditional *command and control* scheme in which spectrum bands are statically assigned to specific services and operators by an authority. This traditional scheme results in inefficiently used spectrum with low utilisation. DSA techniques promise greater spectral utilisation and allow technologically and economically innovative uses of spectrum.

When DSA is used, the transmitter may change the frequency band, for example when another user appears on the used band, and the receiver must follow to maintain the established link. Furthermore, when a receiver wants to join the network, it does not know which frequency is currently used by the transmitter. Determining the transmitter's frequency in order to establish a communication link is commonly referred to as *rendezvous*. The demonstrator developed in this section involves a transmitter that can change its carrier frequency and the receiver is designed to establish rendezvous and to maintain the communication link. This is achieved without a known control channel to exchange information and without prior knowledge of the transmitter's frequency.

The receiver of this demonstrator is designed to use spectrum sensing techniques to scan

the spectrum for the transmission signal, change its frequency accordingly, and establish the link. Real-time video data is transmitted over a coded differential quadrature phase shift keying (DQPSK) link. The transmitter is configured to change its carrier frequency randomly at regular intervals to emulate an opportunistic channel change. When the receiver loses the connection, it searches the spectrum for the transmission signal. Once found, it establishes a link and continues to play the video. All baseband signal processing for both the transmitter and receiver is executed on the proposed Iris-FPGA platform, using the XUP development board with a Xilinx Virtex-II Pro FPGA.

This section first outlines some theoretic background for the sensing algorithm used at the receiver, presents the experimental setup, details the radio configuration at the transmitter and receiver, and summarises the results.

3.2.1 Sensing Algorithm

A large variety of spectrum sensing algorithms for various different applications exists in the literature [75]. The most common and widely-used are based on energy detection, also known as radiometry or periodogram techniques [76]. This requires no information about the signals to be sensed and it can be implemented with relatively low computational complexity. Cyclostationary feature detection [77] is another popular technique that can be applied since most telecommunications signals are inherently cyclostationary random processes. This method requires prior knowledge about the cyclostationary properties of the signal to be sensed. Energy detection is more generic but less accurate at low SNRs, while cyclostationary feature detection has the ability to detect signals with power levels below the noise level.

The sensing algorithm developed for this experiment is based on energy detection, with some modifications since the transmitter's signal properties are known. The algorithm works as follows (illustrated in Figure 3.3:

- ❶ Apply the fast Fourier transform (FFT) to the received discrete-time data samples over a certain observation window.
- ❷ Estimate power spectral density (PSD) by averaging the squared magnitudes of the FFT over a number of FFT windows.
- ❸ Filter the PSD with a moving average filter in frequency domain, with a width matching the transmission signal. This roughly correlates the spectrum with the expected footprint, creating a triangular shape with the peak at the carrier frequency. It also suppresses other signals with different bandwidths.

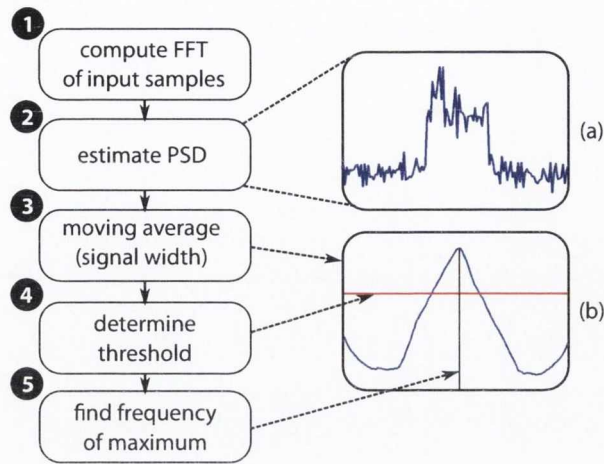


Figure 3.3: Sensing algorithm to find the transmission signal.

- 4 Determine a detection threshold in the filtered PSD, using the method described in [78]. That is, estimate the mean μ and standard deviation σ across frequency and set the threshold δt to $\mu + t\sigma$, where t is a configurable constant between 0.5 and 5.
- 5 The frequency of the maximum value of the area above the threshold is the estimated carrier frequency of the transmitter.

The algorithm has a number of parameters, namely, the bandwidth to scan, the FFT size, number of windows used for averaging, the width of the expected transmitter signal, and the threshold factor t . By carefully tuning these parameters in the demonstration setup, a reliable detection of the transmission signal could be achieved. The purpose of this experiment is to test the capabilities of the proposed cognitive radio platform, not to design and test a state-of-the-art sensing algorithm. Therefore, the choice of the above parameters is of low importance.

Since generally the USRP is not able to process a bandwidth wide enough to cover the whole spectrum band of interest, a mechanism for dividing the band into subbands is needed. It was decided to use overlapping subbands, as illustrated in Figure 3.4, to avoid missed detections at the edges. The overall frequency band of interest is divided into overlapping subbands where f_i denotes the centre frequency of the i th subband. Overlapping subbands are employed to prevent mis-detection at the edges when only part of the signal is received. Note that the sensing algorithm itself is capable of analysing the whole frequency band at once; the subbands are necessary due to the bandwidth limitation of the radio front-end and the processing capabilities of the system (higher bandwidth imply higher sampling rates).

Referring to the numbers in Figure 3.4, the receiver functionality can be summarised as

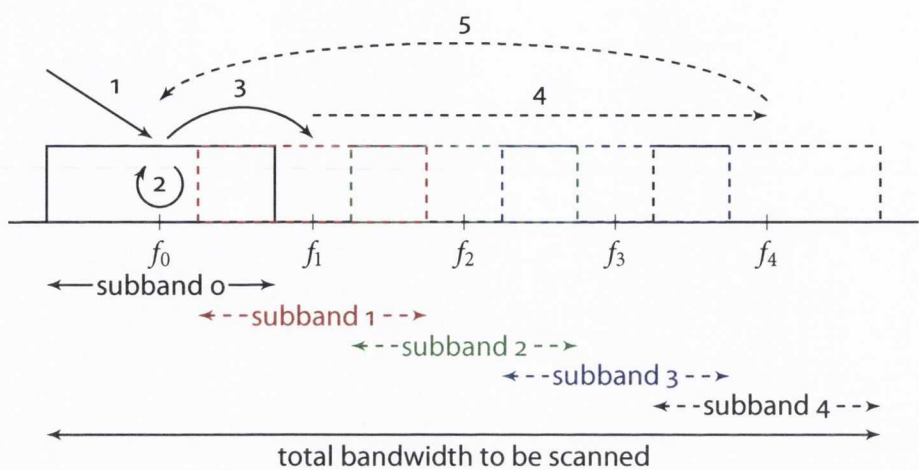


Figure 3.4: Carrier frequency acquisition mechanism. Overlapping subbands are scanned in a sequence until the carrier has been found.

follows:

1. The receiver frequency is set to f_0 and scanning starts at subband 0.
2. The receiver searches for the signal to determine the carrier frequency. If a signal is detected, the receiver estimates the carrier frequency by exploiting the sensing algorithm described above, tunes the frequency of the radio front-end accordingly, and tries to receive the signal. A deframer component checks for a known preamble to determine if the signal was correctly received, and the receiver continues to play the video.
3. If no signal is found or if the signal is lost, e.g., due to a sudden change of the transmitter's frequency, the deframer detects this and the receiver scans in the next subband.
4. The search continues across all subbands until a correct signal has been found.
5. The receiver frequency is changed back to f_0 after scanning the full frequency band of interest.

3.2.2 Experimental Setup

In this application, both the transmitter and the receiver are implemented using the proposed FPGA-based platform outlined in Section 2.2. Two XUP Virtex-II Pro development boards are used on both ends of the link, both connected to a PC. Two USRPs are used as radio front-ends, connected to the PCs via USB 2.0 due to the lack of such connectivity on the XUP boards. The PCs run an Iris radio which acts as a router and transfers the signal samples to or from the development boards via TCP sockets over Ethernet. Furthermore, the PC on the transmitter side runs an instance of the VideoLan VLC Player, which is used to stream video data to the

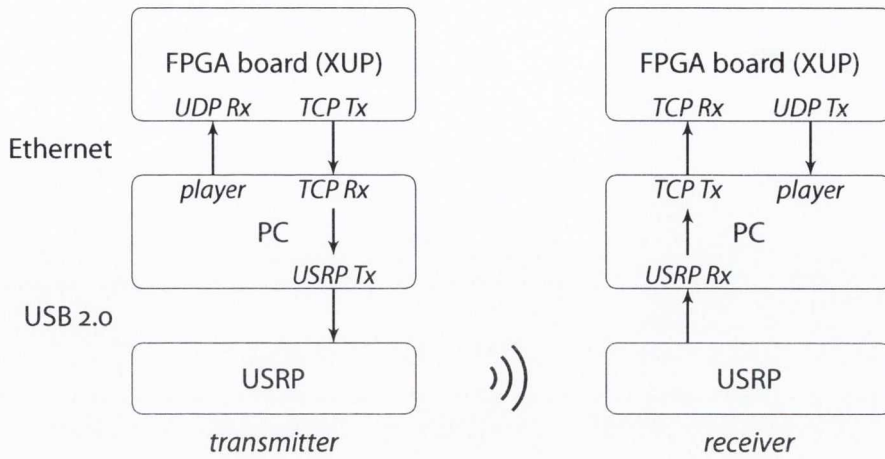


Figure 3.5: Setup of spectrum sensing rendezvous experiment.

transmitter XUP board via User Datagram Protocol (UDP). That is, the first component in the radio chain at the transmitter is a UDP receiver component which obtains the video streaming data from the PC. At the receiver side, the last component in the radio chain on the board is a UDP transmitter which sends the data to a VLC Player instance running on the PC at the receiver side. This player displays the video. In other words, the PCs at the transmitter and receiver end are used for routing the front-end data and for streaming and displaying the video. The experimental setup is illustrated in Figure 3.5.

In order to achieve the higher data rates and lower bit error rates (BERs) required for video streaming, the waveform had to be adapted from the previous experiment in Section 3.1. The modulation scheme is now DQPSK to increase the data rate of the link, and convolutional codes are applied for FEC before modulation to allow the receiver to correct bit errors. This made the implementation more complex as well as more realistic than the previous experiment. The sampling rate over the USB connection to the USRP was increased to 2 MHz¹. This increased the overall data rate of radio to 489 kbit/s.

The video was encoded using a 256 kbit/s MPEG-4 codec, the audio using a 32 kbit/s mono MPEG-4 codec, and both were encapsulated in an MPEG-TS container and streamed using UDP. These bitrates are averages since the video codes are of variable bitrates. This choice allows safe transmission of the video over the 489 kbit/s radio channel, even when the peak video datarate is considerable higher than the configured average. The codecs and streaming were provided by VLC Player.

¹The 100 Mbit/s Ethernet connection to the PC, used to route the signal to and from the USRP, limits the maximum achievable sampling rate to 2 MHz. Each sample is a complex number consisting of two 16 bit integers. Thus, the 2 MHz sampling rate gives a 64 Mbit/s data rate over the Ethernet.

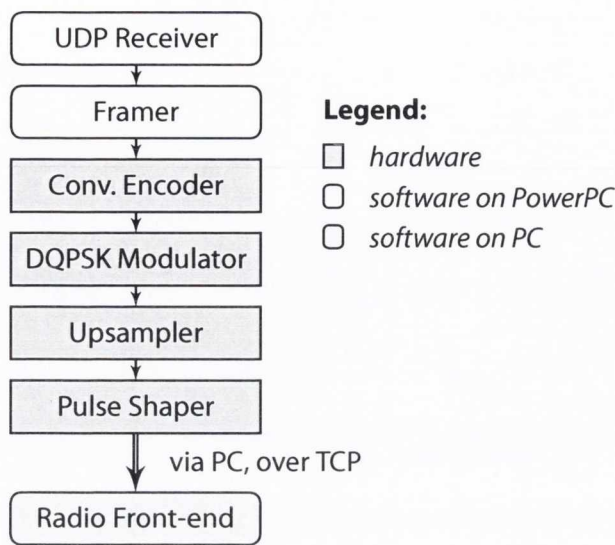


Figure 3.6: Transmitter configuration for sensing rendezvous experiment. The ‘radio front-end’ component combines the link via a PC using TCP and the connection to the USRP via USB 2.0.

3.2.3 Radio Configuration

Transmitter

The transmitter configuration in this experiment is similar to the previous experiment; its block diagram is shown in Figure 3.6. The first component is a *UDP receiver*, which is used to obtain the video streaming data generated by the VLC Player. This is passed into the *framer*, identical to the previous experiment. Then the data is encoded with a convolutional code for FEC by the *convolutional encoder* component. The code rate is 1/2, with polynomials of constraint length 7. That is, each bit is encoded by two bits, introducing redundancy to allow for the correction of bit errors at the receiver. Next, the data is modulated using DQPSK by the *DQPSK modulator*, encoding pairs of two bits with a corresponding phase shift. This generates one signal sample for every two input bits. To limit the occupied spectrum, a root raised cosine FIR filter combined with an upsampler converts each input sample into four filtered output samples. This is performed by the *pulse shaper* component. The data is then transferred to the radio front-end by a *TCP transmitter* through the transmitter PC which serves as a router to the USRP.

The Iris component controlling the USRP radio front-end, running on the PC, is configured so that it randomly changes its transmission frequency approximately every 30 seconds, within a

configurable frequency band. This is to emulate a sudden change of the transmission frequency, for example caused by the appearance of a different communication system on the same frequency. The receiver is designed to detect this change and find the transmission band automatically.

For this experiment, the transmitter does not require a radio controller, it simply keeps transmitting the video stream at changing frequencies.

The computationally intensive parts of the transmitter radio chains are the convolutional coder, the modulator, and the pulse shaper. Those are easily implemented in FPGA hardware, as was done for this demonstrator. The software components were readily available in the Iris-FPGA component library.

Receiver

The receiver is more complex in the demonstrator. It incorporates two modes, reception and detection. When the receiver is in reception mode, it demodulates and displays the video stream. The detection mode is used when the radio is trying to join the network, and when the signal is lost in the middle of reception, possibly due to a change of frequency at the transmitter. A radio controller is needed to decide when it is required to switch between the modes, as well as to tune the parameters of the radio front-end for sensing or reception. An illustration of the receiver configuration is given in Figure 3.7.

When the reception mode is active, the *matched filter* obtains the data from the radio front-end and filters the signal with a root raised cosine filter matched to the pulse shape to increase the SNR. The synchronisation components *carrier recovery* and *timing recovery* are hardware implementations of the same algorithms detailed in Section 3.1.2. Then the samples can be demodulated by the *DQPSK demodulator*. The *Viterbi decoder* corrects bit errors due to the convolutional code applied at the transmitter. The *frame correlator* correlates the known frame access code with the incoming data stream to find the start of each frame. It only returns valid frames to the *deframer* software component, which extracts the payload and compares the checksum to identify frame errors. If no frames are found within a parametrised number of samples, the *deframer* raises a **no signal** event to the radio controller to allow it to switch to detection mode. The recovered video stream data can be sent to a VLC Player by the *UDP transmitter* component.

The detection mode applies the sensing algorithm described in Section 3.2.1. This was separated into two components, the *PSD estimator* to estimate the PSD in the currently frequency

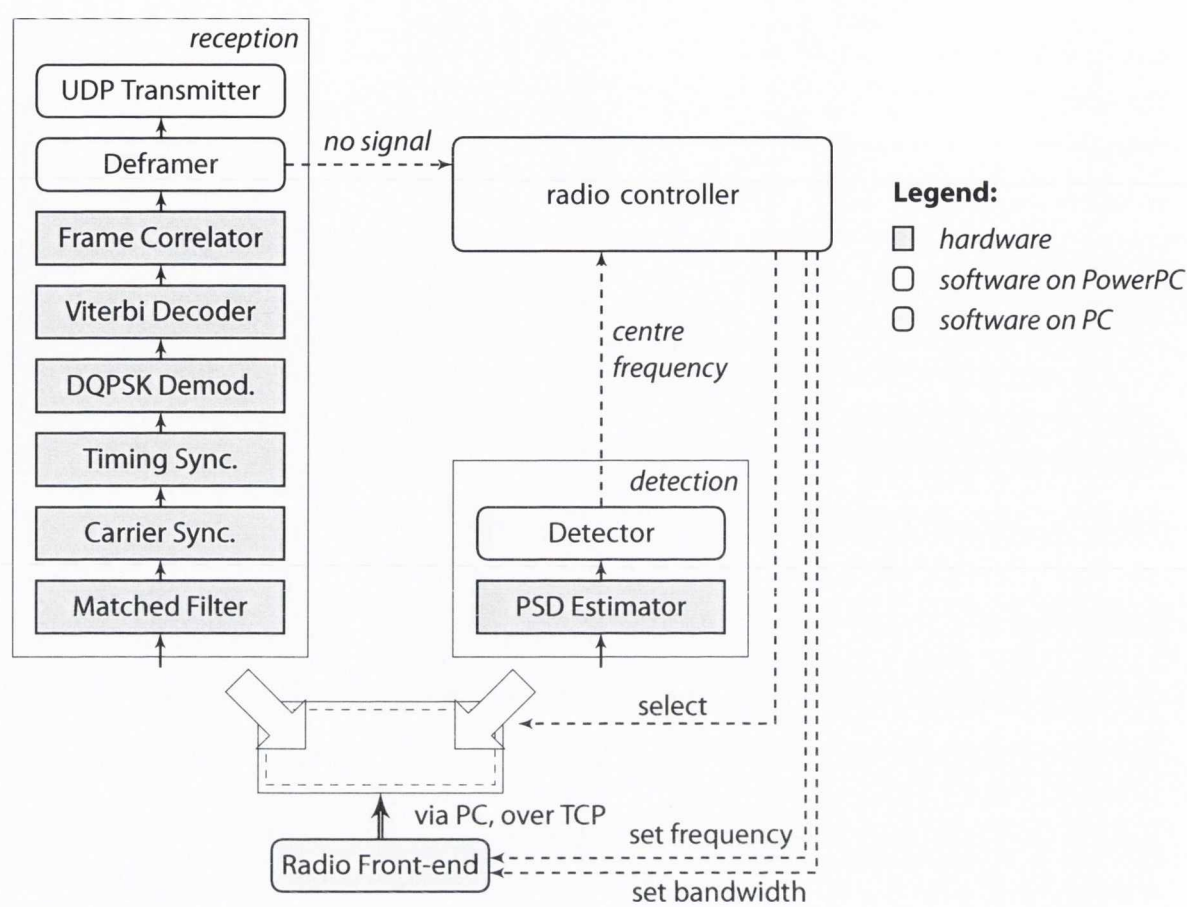


Figure 3.7: Receiver configuration for sensing rendezvous experiment. The radio controller and corresponding control signals are shown. It selects which of the two modes are active and connected to the radio front-end.

band (performing the algorithm steps 1 and 2), and the *detector* to decide whether and where the transmission signal is present in that band (performing steps 3 to 5). The estimated centre frequency of the transmission is reported to the radio controller through an event.

The computationally complex part of the reception chain is everything between the matched filter and the deframer. Note that deframing is implemented partly in software and partly in hardware (the frame correlator and the deframer components in Figure 3.7). This was necessary due to the high complexity of correlating the data with the frame access code, something that can be performed easily in hardware. The hardware part performs this correlation, so that only valid frames are output to software. The software then extracts the data, de-whitens, and compares the checksums. This reduces the amount of data that has to be copied between hardware and software significantly and frees the PowerPC from computing the correlation.

The computationally complex part of the detection chain is the estimation of the PSD since it involves several applications of the FFT and averaging the result, both costly operations. This part of the chain has to be processed in real-time, which is impossible if the embedded PowerPC processor on the Virtex-II Pro FPGA is used. Thus, this part of the chain is implemented in the FPGA's logic fabric. The detector component on the other hand is not processing intensive and only runs once in every detection cycle, after the PSD estimation is complete. Therefore it is implemented in software on the PowerPC.

Radio controller

The radio controller defines which of the two modes is active in the receiver, based on the events it receives from the radio, i.e., the meters of the radio. It subscribes to the `no signal` event of the deframer component as a request to be notified when the signal is lost. It also subscribes to the `centre frequency` event of the detector component, so that it can be notified of the estimated carrier frequency of the transmission. The 'knobs', i.e., the actions taken to change the radio operation, are the active mode and the centre frequency and bandwidth of the USRP radio front-end.

For decision-making, the controller implements a state machine at its core. The states represent the active mode of the radio. The outputs are the active mode, the centre frequency, and the bandwidth. The transition from reception to detection mode is made when the `no signal` event is triggered. Once the `centre frequency` event is triggered, the state machine transits back into reception mode. The entry action in the detection mode is a change of the bandwidth of the radio front-end to the wider bandwidth used during detection a new configuration of the centre frequency according to the subband scan technique discussed in Section 3.2.1. It continues scanning the band and changing the centre frequency until the transmission signal has been detected and its centre frequency is obtained. The entry action in the reception mode is to tune the USRP to the estimated centre frequency and switch the bandwidth as required for reception.

More design and implementation details of this demonstrator are given in Appendix C.2.

3.2.4 Summary and Discussion

The radio designed in this section has been successfully demonstrated using a live video stream. The receiver is able to find the transmission frequency without prior information, at startup or if the transmitter changes frequency during transmission. The receiver radio can adapt its

mode of operation to the current environmental conditions, i.e., the transmission band of the transmitter, as a result of a decision-making process. This fundamental concept for cognitive radios has been demonstrated with this experiment, showing that the proposed framework is capable to develop and run simple cognitive radios.

The high demands of real-time video streaming led to some changes in the architecture of the platform and the radio configurations. The virtual architecture for the FPGA was modified from the experiment presented in Section 3.1, using shared memories for the input and output data transfers. This increases the speed of the data transfer between software and hardware and back, as was required for this demonstrator. Further, convolutional coding had to be used to lower the BER at the receiver since video streaming using MPEG codecs can only tolerate a low rate of errors (the raw audio stream used for the demonstrator in Section 3.1 can tolerate higher error rates). Additionally, it was found that splitting deframing into two parts and perform the frame correlation in hardware greatly increases the performance of the system. The correlation does not have to be computed by the PowerPC and, more importantly, only valid frames are passed back to software. This avoids costly data transfer overhead for data that is dropped later anyway.

The radio itself as well as the decision-making process are simple and arguably not a real cognitive radio setup. However, this radio can be extended using exactly the same architecture and mechanisms without difficulty. The PSD estimator and detector components can easily be modified to detect unused spectrum for use by the transmitter. This demonstrator can also be extended to include advanced signal classification algorithms to inform the radio controller how to configure the receive chain, e.g., setting the modulation type or data rate. The different settings might result in different FPGA configurations loaded into the customisable processing subsystem (CPSS) but the radio controller is not concerned with this level of detail.

This setup has been demonstrated to an international audience at the IEEE DySPAN conference 2008 [68], in an earlier version where the detection mode was implemented in software on the PC. A photo of that demonstration is shown in Figure 3.8. The version described here was used as a case study for a publication at the IEEE GLOBECOM conference 2009 [69], where it was presented and discussed.

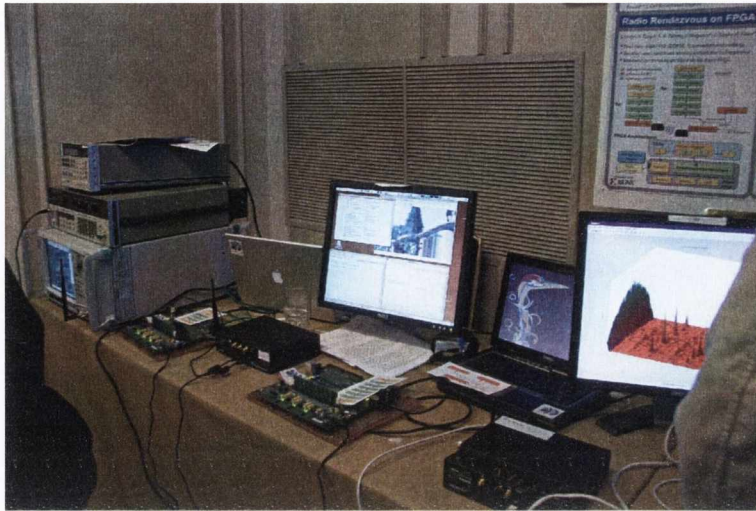


Figure 3.8: Sensing demonstration at DySPAN 2008.

3.3 Video Streaming Link with Adaptive Coding

This demonstrator was designed to show how a radio can adapt its operating configuration to the current state of the wireless channel autonomously. The quality of the received data is continuously monitored and the radio adapts to the conditions by changing the applied coding scheme.

The waveform used is a DQPSK narrowband signal, similar to the one used for the demonstrator in Section 3.2. The adaptation is implemented by allowing coding to be reconfigurable. A wireless channel with high noise levels can cause high BERs at the receiver. Using FEC reduces these errors to varying extends. The strength of the error reduction with convolutional codes is higher with longer code polynomials, i.e., higher constraint lengths. A convolutional coder with reconfigurable constraint length can *optionally* be inserted prior to the modulator in the transmitter, and a matching Viterbi decoder after the demodulator in the receiver. The radio controller monitors the current BER and FER (the ‘meters’), makes decisions based on theoretical BER curves, and adapts the coding strength accordingly (the ‘knobs’). The application data is a video stream.

The experimental setup, the radio configuration at the transmitter and receiver, and experimental results are given in this section.

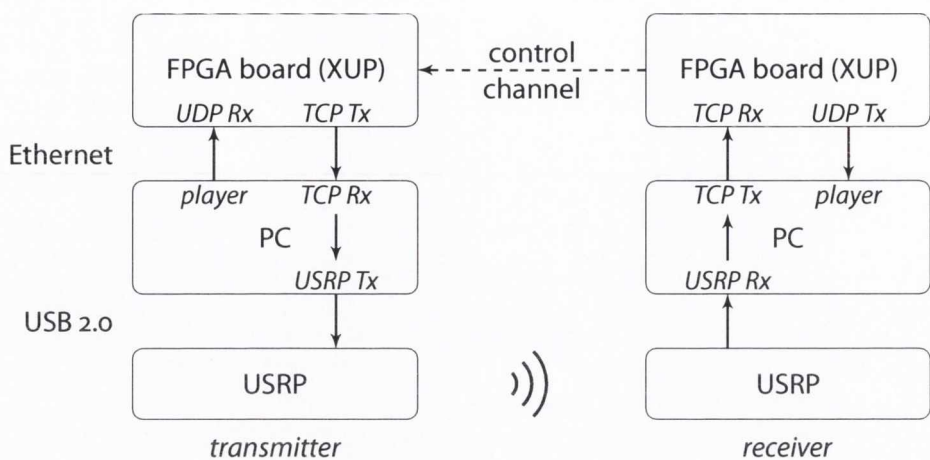


Figure 3.9: Setup of the adaptive coding demonstrator.

3.3.1 Experimental Setup

As for the previous demonstrators, the implementation platform used is the XUP board which hosts a Xilinx Virtex-II Pro FPGA. The USRP radio front-end [65] is used, connected through a TCP socket via a PC bridge. To allow change of the coding strength at the receiver, it has to inform the transmitter of this change. For simplicity, this information is transmitted through Ethernet in this experiment, although this could be done wirelessly. Figure 3.9 shows the demonstrator setup.

The transmitter receives UDP packets, through Ethernet, containing video data, streamed from a PC using the VideoLAN VLC Player. These are processed through the transmission chain on the FPGA, then transmitted by the USRP. At the receiver, the signal is processed by the reception chain in the FPGA, then passed, through UDP, to a VLC Player window on a PC which displays the video. As for the previous demonstrators, the connection to the USRP radio front-end is routed through the PC due to a lack of a USB 2.0 connector on the XUP board.

3.3.2 Radio Configuration

Transmitter and receiver configuration

The radio configurations for the transmitter and receiver are shown in Figure 3.10. Both are similar to the ones for the previous demonstrator (see Section 3.2), with the exception that the encoding and decoding components are optional and parametrised with the constraint length of the used code polynomial. Convolutional codes with code rate 1/2 and polynomials

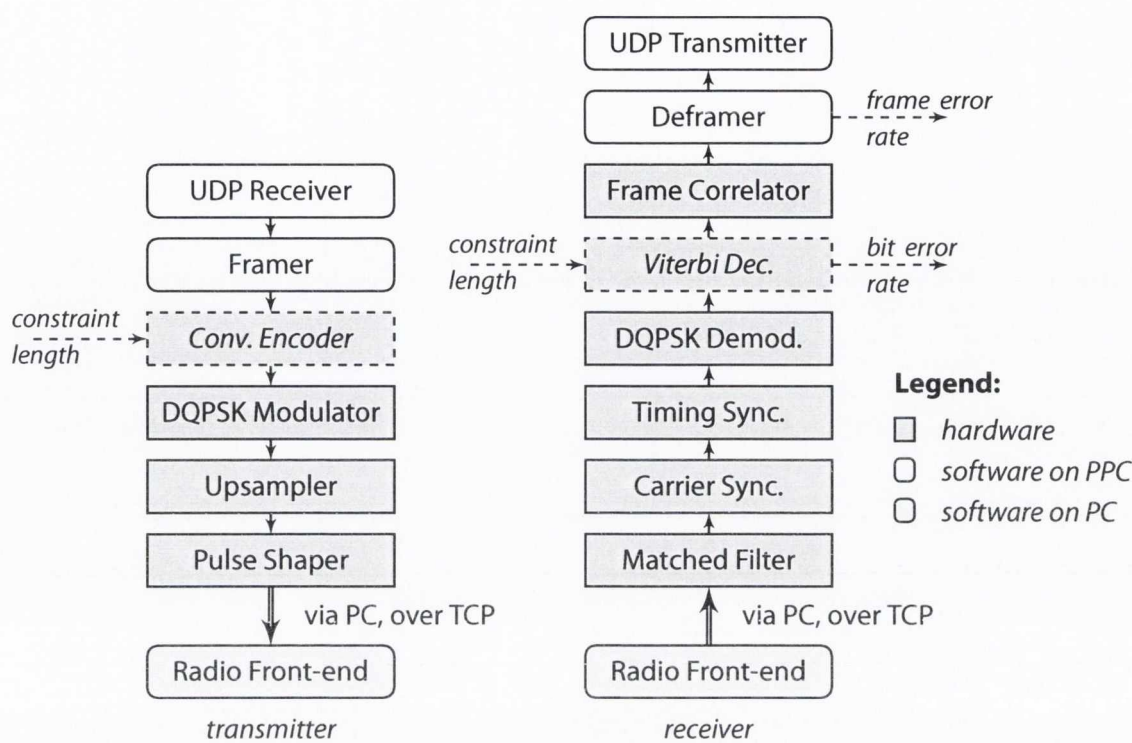


Figure 3.10: Transmitter and receiver configurations for the adaptive coding experiment.

of constraint lengths 6 and 8 are implemented, although this could be extended to include other codes. To obtain an estimate of the current BER, the Viterbi decoder is configured to re-encode the decoded data and compare to the original input. This functionality is provided by the Viterbi decoder intellectual property core from Xilinx [79]. When no coding is present, and thus no BER estimate from the Viterbi decoder is available, the controller has to rely on the FER. This was achieved by modifying the deframer component to provide an estimate of the current FER in an event, based on the number of missed and erroneous frames detected using the checksum field.

All other components are identical to the ones used in the previous experiment. For details the reader is referred to Section 3.2.

Radio controllers

As can be seen from the simulated BER curves in Figure 3.11, the BER at the receiver depends significantly on the coding scheme used. On the other hand, lower constraint length codes or no coding result in considerable area and power savings on the FPGA (the power consumption and area requirements for this setup are given in Appendix C.3). To resolve this trade-off

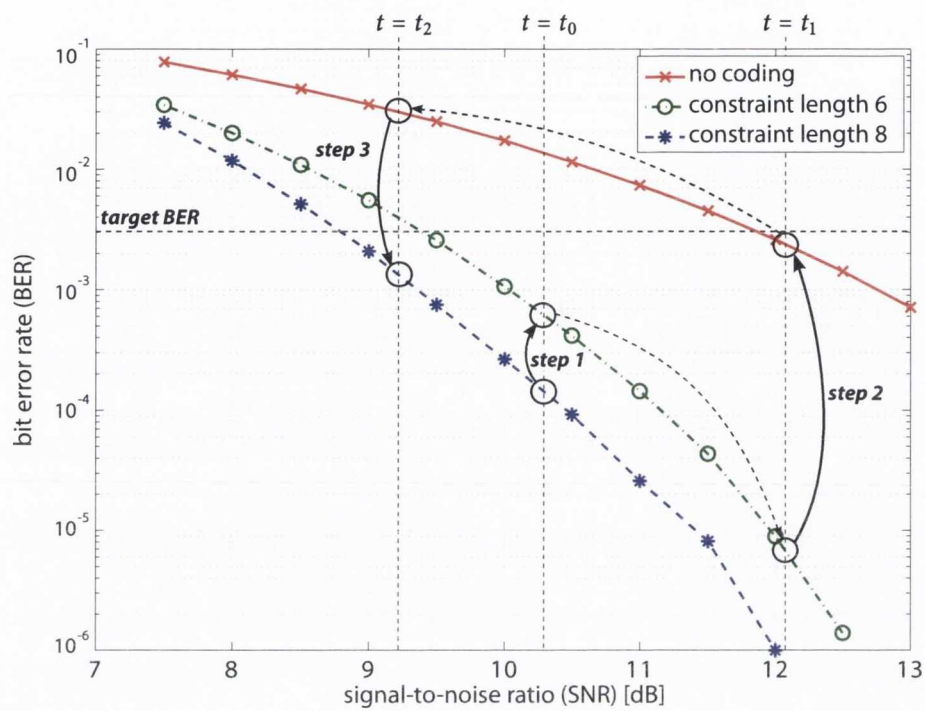


Figure 3.11: BER vs. SNR for different codes, and an example code switching sequence.

a radio controller which changes the coding scheme depending on the perceived BER was designed.

Setting a target BER, as shown in Figure 3.11, the adaptation algorithm implemented in the radio controller at the receiver decides when a switch of the coding scheme is required². Decisions thresholds are programmed into the radio controller, based on the simulated BER versus SNR curves shown in the figure. Every coding scheme has different thresholds, tuned to the target BER and the curves shown in the figure. If the current BER is higher than the target, a reconfiguration to a higher constraint length code is triggered. If the BER is low enough that a less complex code is sufficient to achieve the target, the controller lowers the constraint length or switches the coding off completely. To allow the transmitter to respond to an event at the receiver and reconfigure to the same constraint length, a feedback channel from the receiver to the transmitter is needed. In this implementation, Ethernet is used to connect the receiver controller to the transmitter controller through a TCP socket, though this could be done wirelessly.

² The target BER in Figure 3.11 is chosen for illustrative purposes only.

An example adaptation sequence is shown in Figure 3.11. The system starts in the most robust coding state ($CL = 8$) at time $t = t_0$. The controller monitors the BER output of the Viterbi decoder at the receiver. The hardware reports this value to the software wrapper using the register interface (see Figure 2.2), which in turn triggers an event to the controller. The controller compares the BER to the threshold for switching to less robust coding ($CL = 6$), indicating that it remains below the target BER, so it initiates a reconfiguration of the *constraintLength* parameter of the decoder component to 6 (step 1 in Figure 3.11). A control packet is sent to the controller at the transmitter to adjust the constraint length of the encoder component accordingly. The software wrapper at both the transmitter and receiver reconfigures the CPSS to the new configuration at run-time, using the dynamic reconfiguration feature of the used FPGA.

At time $t = t_1$, the signal quality increases. When it is high enough that the target BER can be achieved without coding, which the controller knows based on the programmed threshold, the controller removes the decoder from the receiver radio chain (step 2 in Figure 3.11) and informs the transmitter to do likewise with the encoder. Again, the software wrapper on both sides initiates a reconfiguration of the CPSS in the FPGA's logic fabric at run-time, using dynamic partial reconfiguration.

Now the FER output by the deframer is monitored. If frames are lost completely, i.e., the frame access code was not found, this is an indication that the channel has worsened significantly and a reconfiguration to include coding is triggered. For frames detected but containing errors (as detected by the deframer through comparison of the checksum), a threshold for the maximum tolerable FER is programmed into the controller, which is an indication of the current BER.

At time $t = t_2$, the channel conditions have worsened considerably and the BER has risen above the target, indicated by the FER event of the deframer. The encoder and decoder are reinserted with a constraint length 8 code (step 3 in Figure 3.11). This initiates a partial reconfiguration of the FPGA on both sides, managed by the software wrapper component.

In this manner, the system continues to use the most suitable type of coding for the current channel conditions.

This demonstrator has been evaluated for the FPGA resource utilisation, the communication performance, and the associated power consumption in order to show the benefits of adapting the coding scheme using the proposed framework. The details are given in Appendix C.3.

3.3.3 Summary and Discussion

A radio demonstrator was designed that adapts the used coding scheme to the current channel conditions based on the perceived quality of the received signal, so that energy and resources could be saved. The benefits of this setup have been shown, with details given in Appendix C.3.

This demonstrator shows how the proposed Iris-FPGA framework can be used to design a radio that makes best use of its resources. It does so by minimising the required resources while maintaining a target signal quality, i.e., it follows an objective function. It implements a decision-making controller that obtains information about the environment ('meters') to inform its decision about the radio's operation parameters ('knobs'). Both these meters and knobs are exposed from FPGA hardware but can be accessed and adjusted at run-time without the knowledge of low-level hardware details. That is, this experiment demonstrates the capability of the proposed framework for building cognitive radios which include FPGA hardware and general purpose processor (GPP) software seamlessly.

From the experience of this and the previous demonstrator, it became apparent that connecting hardware radio chains from individual components and writing the software wrapper code can be automated. This led to the development of the hardware composition tool outlined in Section 2.2.4. It was applied for this demonstrator, with some manual adjustments of the generated code. It is envisaged that if the tool is developed further in the future it can automate these task completely, freeing the developer from performing these low level implementations manually.

During the FCCM conference 2009, this experiment was demonstrated to an international audience and a paper was published and presented [70]. In contrast to the conferences chosen for the other demonstrations in this chapter, FCCM is a hardware-centric conference.

3.4 Combining Adaptive Coding and Spectrum Sensing

This experiment combines the previous two experiments into one, building a radio link that can establish communication rendezvous without prior knowledge of the transmission frequency and without a dedicated control channel, and can adapt the FEC used to the current channel conditions at run-time. A new and more up-to-date FPGA development board is used, introducing many new features and more resources for more complicated radios.

The purpose of this experiment is two-fold. First, it shows that a more complicated and more realistic radio system can be created using the proposed framework. Second, it shows that

in applying the virtual architecture introduced in Section 2.2.2 moving to a new development board is possible without difficulty.

In this application, the transmitter starts transmitting a video stream at an unknown arbitrary frequency, using DQPSK modulation and convolutional coding. The receiver wakes up, and enters detection mode, where it uses spectrum sensing to locate transmitted signals within the frequency band. Once a transmission has been found, the radio reconfigures into reception mode and attempts to demodulate the signal. If this fails, the receiver switches back to sensing mode and searches for another signal. Once a correct transmission is found, the receiver continues to demodulate.

The transmitter can change frequency during transmission. When the receiver detects that the signal is lost, it switches back to sensing mode and searches for the transmit signal again, as described above.

During reception, the receiver monitors the BER, and based on this determines when to change the strength of the code, and when coding can be switched off entirely. This adaptation is continuous, based on channel conditions. It results in power and resource savings while maintaining the link quality.

The receiver in this application uses multiple meters to get information about the environment and the radio's state. It monitors whether the signal is received correctly, the centre frequency estimated in detection mode, the current BER, and the current FER. The knobs it can adjust are the mode it operates in (i.e., detection or reception), the centre frequency and bandwidth of the radio front-end, the constraint length of the code used, or whether coding is used at all. Thus, it is a cognitive radio system with multiple meters and knobs, where the objective of the decision-making is to maintain the link and minimise the used resources while maintaining a target signal quality.

This following section first discusses the process of moving the framework to a new development board and the experimental setup, radio configuration, and some design details are discussed in the subsequent sections.

3.4.1 Moving to a New FPGA Board

Due to the fact that the hardware on XUP development board is outdated (the board was released in 2005, the Virtex-II Pro FPGA in 2002) and is limited in space and performance, it was decided to move to a more up-to-date board. At the time of development, the most recent line of Virtex FPGAs was the Virtex-5. Several development boards were on offer by

Xilinx, one of which featured a Virtex-5 FXT FPGA, i.e., a version with an embedded PowerPC processor, and came with a reference design to boot Linux. For this reason it was decided to use the Xilinx ML507 development board [61].

The ML507 comes with a Virtex-5 FX70T FPGA with 11,200 Virtex 5 slices, a PowerPC core, a hard gigabit Ethernet core, and many special features for digital signal processing (DSP) and embedded processing designs. The Virtex-II Pro on the XUP board has 13,696 Virtex-II slices and did not provide many of the valuable special features for DSP and embedded processing. Virtex-II and Virtex-5 slices cannot be compared directly since a Virtex-II slice consists of two 4-input look-up tables and two flip-flops, while a Virtex 5 slice consists of four 6-input look-up tables and four flip-flops. Xilinx provides an estimate of the number of logic cells (AND gates, OR gates, etc.) that fit on the device as an indicator of capacity. The Virtex-II Pro on the XUP has 30,816 logic cells and the Virtex 5 on the ML507 board has 71,680. Thus, there is a vast amount of extra space on the new FPGA, crucial for more complex cognitive radio configurations. For more details the reader is referred to the data sheets [60, 61].

All components of the virtual architecture as stated in Section 2.2.2 can be mapped to a component on the ML507 development board. The processor is a PowerPC 440 core, which includes a memory management unit (MMU) to control external DDR2 memory. Ethernet is provided by a gigabit Ethernet MAC in the FPGA device's silicon (i.e., it is not instantiated in the FPGA logic, giving higher performance and saving logic area). A Compact Flash card holds the Linux root filesystem. The main system bus is a 128 bit wide Processor Local Bus (PLB) bus, which connects all the components together.

Although dynamic partial reconfiguration is possible on the Virtex 5 FPGA, the Xilinx development tools support was not finished by the time the development of this demonstrator commenced. The Virtex 5 partial reconfiguration feature is supported from the Xilinx ISE Design Suite version 12 [44], released in April 2010. This release was too late for the development of this demonstrator. However, the previous three demonstrators on the XUP board prove the functionality of the concept and make it clear that this feature can easily be added to the new board once the new design tools are used. For this demonstrator, the Virtex 5 is large enough to hold all possible FPGA configurations at once (approximately five times larger than the Virtex-II Pro). The software can select which configuration to use by setting a register value, instead of using partial reconfiguration. Thus, the *config* element of the virtual architecture is mapped to a switch in the FPGA.

The first step when moving the framework to this board is to synthesise the processor subsystem (PSS) and boot Linux on the board. A reference design for Linux is available for

this board from Xilinx [80]. The Xilinx open source wiki [81] provides official support to boot Linux using this reference design and gives instructions how to configure the kernel and create a root filesystem. The Linux kernel, version 2.6.29, was successfully booted, using the same Debian root filesystem as for the XUP board.

To map the full virtual architecture, the reference design had to be altered to include the CPSS. It was added as a slave on the PLB bus, with all configuration possibilities present in parallel. At run-time, the active configuration can be selected using a register. This is due to the lack of support for partial reconfiguration discussed above.

With all parts of the virtual architecture mapped, the new development board is ready for use with Iris. All the experiments explained earlier can be used with the new board without difficulty, all that is needed is to synthesise the hardware chains for the new FPGA.

3.4.2 Experimental Setup

The ML507 development board is used for this experiment, with the USRP as radio front-end. VLC Player, running on a PC connected through Ethernet, is used as the source and sink for the video transmission.

Although the ML507 has a physical USB 2.0 connector, and the FPGA hardware core to access it is available from Xilinx, no Linux driver for a high-speed USB host was available at the time of development. Therefore, the workaround to route the baseband data through a PC had to be used again, but this time through gigabit Ethernet. It is anticipated that the radio front-end will be connected directly to the board in future experiments.

An overview of the demo setup is shown in Figure 3.12. The transmitter receives video streaming data from a VLC player on the PC through UDP, passes it through a radio processing chain using DQPSK modulation and, optionally, convolutional coding, and sends it to the radio front-end via a bridge through the PC. In receive mode, the receiver FPGA obtains the data from the radio front-end through a PC, demodulates the video stream data, and sends it to a VLC Player for display using UDP. When the receiver is in sensing mode, it scans the given frequency band for the carrier frequency of the transmitted signal tunes the radio front-end to it, and switches to receive mode.

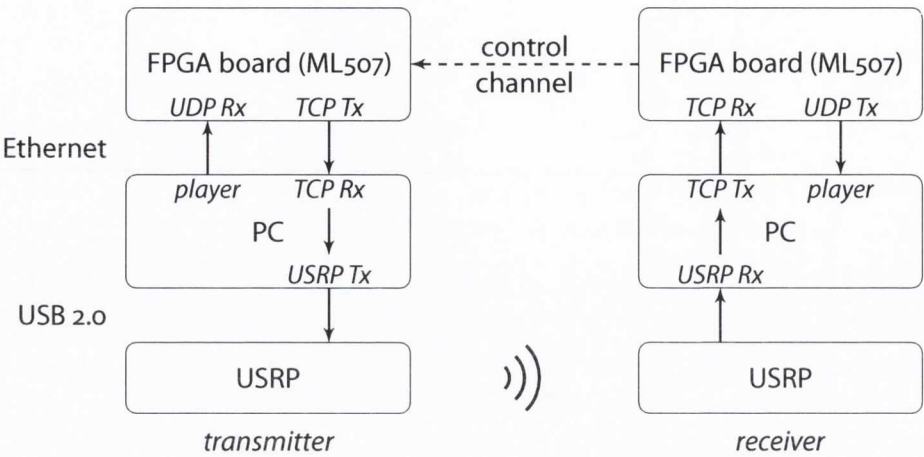


Figure 3.12: Setup of the combined sensing and coding experiment.

3.4.3 Radio Configuration

The transmitter configuration is the same as already explained for the adaptive coding experiment in Section 3.3.2, with the only difference that the radio front-end is configured to change its frequency periodically to emulate the effect of an opportunistic frequency change.

The receiver is a combination of the sensing (Section 3.2.3) and adaptive coding (Section 3.3.2) experiments. That is, it consists of two modes, detection and reception. The detection mode is identical to the sensing experiment. The reception mode is identical to the adaptive coding experiment, allowing four different configurations: no coding, and with a Viterbi decoder with three different coding strengths. Due to the large number of resources on the Virtex 5 FPGA, convolutional codes with constraint lengths 5, 7, and 9 could be used (the constraint length 9 decoder was too large for the Virtex-II Pro FPGA).

The receiver radio configuration is shown in Figure 3.13. The transmitter configuration is identical to Figure 3.10. For a detailed explanation of the individual radio components in Figure 3.13, the reader is referred to sections 3.3.2 and 3.2.3.

The radio controller at the receiver in this demo was chosen to be implemented as a hierarchical state machine. At the top level, it has two states representing the operating modes: sensing and reception. Transitions between these states are triggered if the signal is lost, or if possible transmission frequencies are found, respectively. When the receiver is in reception mode, it may be in one of four coding configurations. Thus, the reception state contains an internal states for each coding method. Transitions between these states are triggered when the BER or FER is above or below predefined thresholds. These are determined using simulated

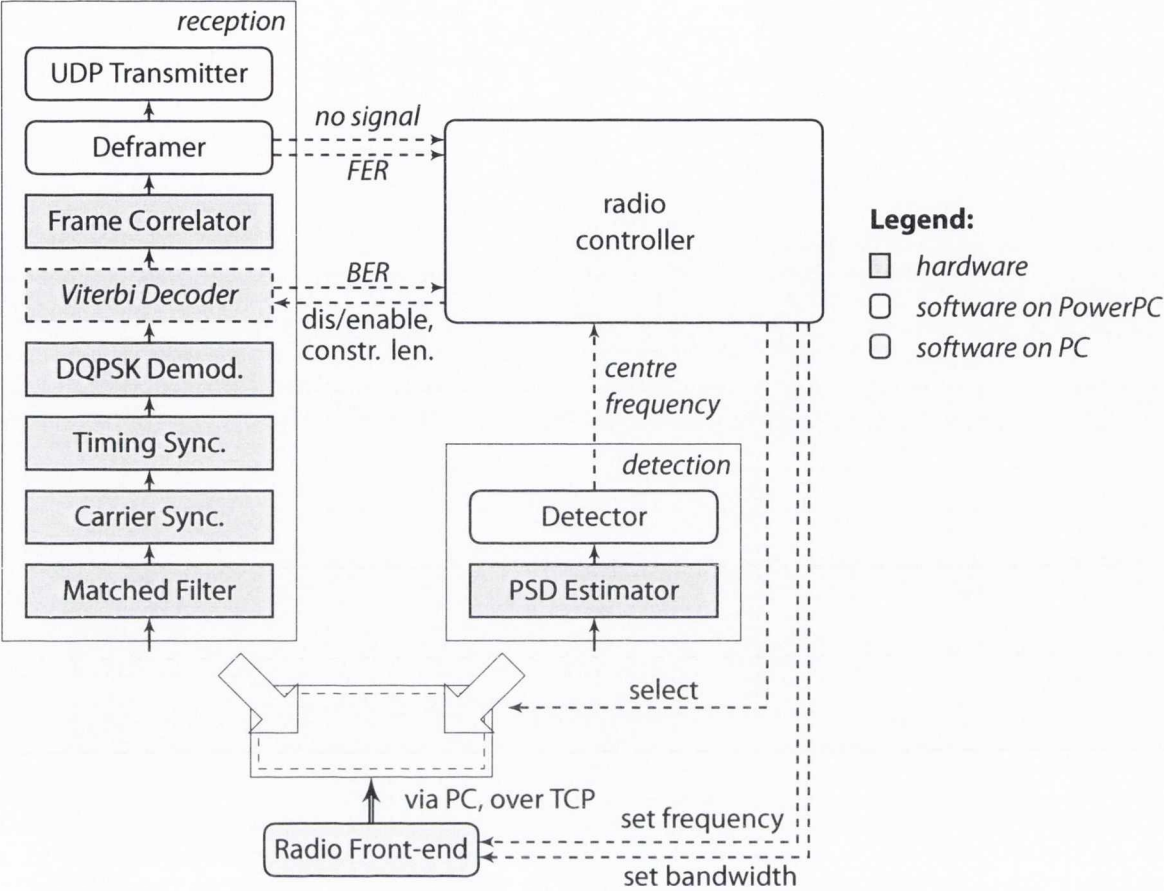


Figure 3.13: Receiver configuration for the combined adaptive coding and sensing experiment. The controller and control signals are shown. The controller selects which of the chains are loaded into the system and connected to the radio front-end as well as the decoder and radio front-end parameters.

BER versus SNR curves as explained in Section 3.3.2. This full state machine is illustrated in Figure 3.14. When the receiver loses the signal, and therefore switches to sensing mode, it has to remember the previously used coding scheme. Therefore the reception state has history, i.e., when switching to detection state it remembers the internal state that was active before the switch and returns to that when reception is activated again.

Implementation details of this demonstrator are given in Appendix C.4.

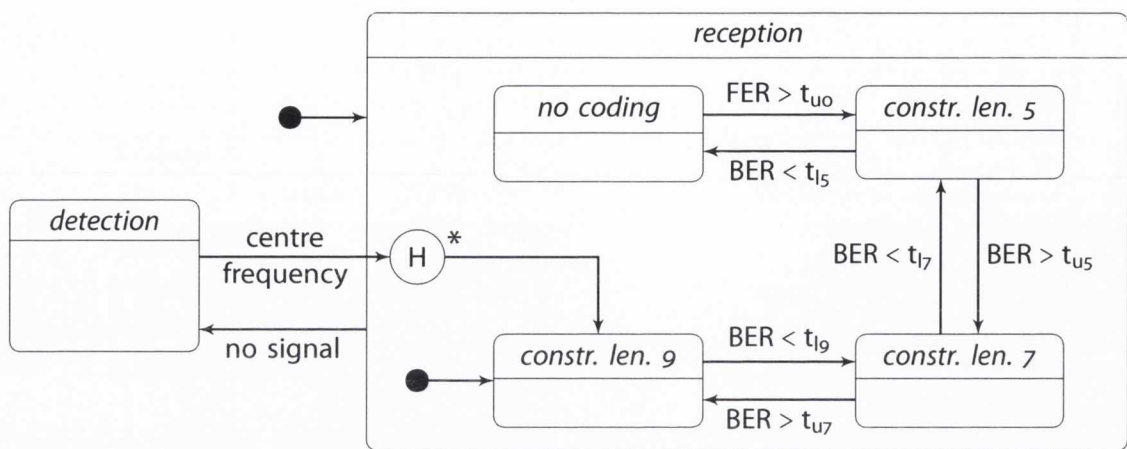


Figure 3.14: Controller state machine for combined adaptive coding and sensing experiment. The UML state chart notation is used for this diagram.



Figure 3.15: Setup of the sensing and adaptive coding demonstrator at SIGCOMM 2009.

3.4.4 Summary and Discussion

A frequency agile video streaming demonstrator which adapts its coding scheme to the current signal conditions was built using the proposed FPGA-based cognitive radio framework Iris-FPGA. All adaptation and switching between operating modes is handled by the software wrapper component, the radio designer does not need to concern himself with the details. The benefits of defining a virtual architecture, as presented in Section 2.2.2, show when Iris-FPGA is moved to the newer ML507 development board.

This demonstrator was presented to an international audience during a demonstration session at the ACM SIGCOMM conference in 2009 [71]. Figure 3.15 shows the setup during this conference.

This experiment shows a complex radio setup involving several operating configurations. The radio controller performs decision-making using a hierarchical state machine, based on information retrieved by the radio ('meters'), and effects its decisions through reconfigurations

of the radio (‘knobs’). The meters and knobs involve FPGA hardware as well as software, working together seamlessly and transparent to the controller. This demonstrator shows all functionality required to build complex cognitive radio systems based on the proposed cognitive radio frameworks.

3.5 Summary

The FPGA-based cognitive radio framework Iris-FPGA, proposed in Chapter 2, has been tested and evaluated thoroughly with respect to its suitability to build real cognitive radio systems. A series of demonstrators have been developed, showing that

- 1. the framework can be used for a simple software radio configuration,
- 2. simple decision-making can be done, based on meters in software, and actions that reconfigure the FPGA’s logic fabric can be taken,
- 3. a radio that can adapt itself to the current channel conditions can be developed, based on hardware meters and following an objective function,
- 4. more complex cognitive radios, involving different modes and configurations and using meters and knobs in both software and hardware can be developed, and the architecture is portable to different development boards.

Furthermore, it has been shown that FPGAs can be used as a platform to overcome the performance issues of the GPPs in standard PCs without the need to be a hardware design expert.

Table 3.1 lists the concepts demonstrated by the different experiments. As can be seen, all the fundamental concepts of a cognitive radio, according to its definition, can be implemented on the proposed platform.

	SDR	decision-making		meters		knobs		board
		simple	complex	SW	HW	SW	HW	portability
Audio Tx	✓							
Sensing	✓	✓		✓		✓		
Coding	✓	✓		✓	✓	✓	✓	
Sensing+Coding	✓		✓	✓	✓	✓	✓	✓

Table 3.1: Concepts demonstrated by the different experiments. The experiments are presented in sections 3.1 to 3.4. (Notation: SDR = software defined radio; SW = software ; HW = hardware)

Furthermore, the computational power of FPGAs can be easily exploited, which enables radio configurations that would be impossible using a PC. For example, Viterbi decoding is a highly complex algorithm that can be parallelised easily.

To illustrate this fact, the highly optimised Viterbi decoder with constraint length 7 and code rate 1/2 by Phil Karn [82] has been used³ on an AMD Phenom II X6 processor clocked at 2.8 GHz. It achieved a data rate of 57 Mbit/s. An equivalent implementation on a Virtex 5 FPGA achieves rates of 272 Mbit/s, according to the datasheet of the intellectual property core provided by Xilinx [79]. However, the real power of FPGAs shows when multiple components are implemented in hardware. The speed of the Viterbi decoder on the FPGA stays the same, no matter how many other components are present in the radio chain, due to the parallel architecture of the device. On a PC, all computation shares the same processor (with potentially multiple cores) with limited compute power. This results in considerably decreased performance when the processing components in the radio are increased or become more complex, as more compute tasks compete for the same number of compute resources. This is not an issue for FPGAs, where one component is not affected by others running concurrently.

The experiences gained and challenges discovered during these experiments led to valuable insights into the realities of cognitive radio design. These are discussed in Chapter 4.

³The Viterbi performance with Phil Karn's library was measured by executing the speed test `vtest27` provided.

4 Observations and Analysis

Contents

4.1 Iris-FPGA: Reflections	93
4.2 Approaches to Cognitive Radio Design: An Analysis	105

Following on from the experiments presented in the previous chapter, this chapter looks at what can be learned in order to enable and simplify cognitive radio experimental research.

It begins with a reflection of the key strengths and weaknesses of Iris-FPGA in Section 4.1. Together with the strenghts and weaknesses identified for existing frameworks and platforms in Section 1.8, this forms a basis for further analysis.

It is found that, more than the capabilities of the execution platform itself, the simplicity and effectiveness of the radio design process is key for enabling cognitive radio experimental research. Cognitive radio researchers do not wish to concern themselves with the low level engineering aspects of building a demonstrator, they want to test their algorithms and ideas in the real world. Therefore, Section 4.2 compares the design approaches taken by the discussed platforms and frameworks and identifies a set of characteristics for an ideal cognitive radio design process. This serves as a motivation for the novel design approach proposed in Part II of this thesis.

4.1 Iris-FPGA: Reflections

The field-programmable gate array (FPGA)-based cognitive radio framework Iris-FPGA, proposed in Chapter 2, was tested extensively by building a series of demonstrators with

various degrees of reconfigurability and cognition in Chapter 3. Based on this experience, the key strengths and weaknesses of this framework are identified and discussed in this section. This critically evaluates Iris-FPGA and serves as a basis for comparisons with other existing frameworks in the subsequent section. It also gives some key insights and learned lessons about the implementation of cognitive radios for experiments.

4.1.1 Strengths

The following strengths of the proposed FPGA-based platform have been identified:

- Block diagram for radio configuration is efficient
- Support for cognitive cycle
- Ease of use
- Portability between platforms and architectures
- Efficient combination of hardware and software
- Software suitable for embedded systems
- Exploitation of dynamic partial reconfiguration of FPGAs

Each strength is explained in more detail in the following sections.

Block diagram for radio configuration is efficient

A generally accepted and widely-adopted approach to design any type of system is to break the design into smaller manageable units. The breakdown is chosen to achieve a low amount of coupling, enabling independent development and unit testing of individual components before integration into the full system. The interfaces between components should be well-defined in advance to the detailed design to allow easy integration later in the design process. If the interfaces are general enough, components can be reused in different systems without much difficulty.

For radio systems it is natural to break down the signal processing into individual tasks, e.g., modulators, filters, amplifiers, or coders. The full system can then be described by a diagram of these components, connected by channels that represent the flow of data. This approach is followed by Iris-FPGA.

In fact, this block diagram of components is a directed graph, where the vertices are components and the edges represent the dataflow. These graphs are called *dataflow graphs* in the literature, and designing systems based on these graphs is often described as *dataflow-based*

design [83, 84]. This is an intuitive and established approach for describing signal processing system and is used by many design and simulation tools outside the field of cognitive radio (e.g., [85–87]). It decouples the signal processing implementation from the system composition and allows for the composition of systems without detailed platform knowledge. If all components implement a standard interface, they can be used individually and as part of many different setups without modification. This approach has many advantages, for instance:

- independent development and test of individual components is easy,
- decoupled signal processing implementation from system composition,
- for radio composition it is not necessary to know how components are implemented, only what is their function ('black box' components),
- system composition is implementation-independent, and
- components can be placed in a library for re-use in different setups.

Using dataflow graphs to compose digital signal processing (DSP) systems, e.g., the data path of cognitive radios, is common. For instance, signal processing systems can be designed using dataflow graphs in Matlab Real-Time workshop [88], Xilinx System Generator for DSP [31], or Ptolemy II [85]. It is therefore a natural choice to partition a cognitive radio system into dataflow components and to use dataflow graphs for system composition.

Since the Extensible Markup Language (XML) description of components and their links in Iris-FPGA is essentially a dataflow graph, composing a radio for the proposed FPGA-based cognitive radio framework Iris-FPGA benefits from the advantages mentioned above. Thus, *using dataflow graphs for radio composition is an efficient and effective approach to cognitive radio design.*

Support for cognitive cycle

The simplified cognitive cycle consists of three phases: *observe*, *decide*, and *act*. The proposed framework supports these phases directly, providing mechanisms and interfaces to implement them with ease. The entity executing the cognitive cycle is the cognitive engine, realised in Iris-FPGA by the radio controller.

During the *observe* phase, the controller reads the meters of the radio. These may be information about the status of the radio dataflow graph, i.e., which components are present, how they are connected, how they are configured (the parameter settings), and what events they are publishing. These events represent the second type of meters. When the controller subscribes to them and they are triggered by the component information about the current

state of the component or radio can be conveyed. For example, the current bit error rate (BER) can be sent to the controller. These mechanisms allow a radio controller to observe the radio in order to inform its decisions.

During the *decide* phase, the controller reasons about possible actions in order to achieve some objective, using the observed meters. Iris-FPGA supports this by allowing radio developers to implement arbitrary decision-making functions, i.e., the decide phase can be implemented as needed.

The decided actions are affected to the radio during the *act* stage. Iris-FPGA supports reconfiguring parameters of running radios, replacing components with others, invoking commands in components, or sending asynchronous data to components. These allow to bring the radio into a state that better suits the current environmental conditions according to a given objective.

If FPGA hardware is used in the radio, the above methods are still supported in the same fashion. Meters can be read and knobs can be tuned also for hardware components, through the software wrapper concept (see Section 2.2.4).

The experiments from Chapter 3 demonstrate these concepts on example radios, as can be seen from Table 3.1 on page 91. Hence, *Iris-FPGA implements all aspects of the cognitive cycle.*

Ease of use

The proposed Iris-FPGA framework provides many mechanisms to simplify radio design and development for radio designers.

The dataflow-based radio composition is the first, providing an intuitive approach to radio composition as discussed earlier in this section. Using XML for describing these graphs is simple and has the advantage of being both human-readable and machine-readable.

The interface for implementing software components is easy to use. Only the functions of interest need to be implemented, everything else is automatically generated. Specifying parameters, events, commands, and ports only requires adding special comments into the code. The C++ code of the functions required to support those (the Reflection interface) is generated automatically by a parser/code generator (see Section 1.3.1).

Hardware component implementation is relatively simple. The components need to implement a simple interface (see Figure 2.3 on page 52) and can provide adjustable parameters and events through a register interface. They can be implemented using any hardware design flow, e.g., VHDL, Verilog, Xilinx System Generator for DSP [31], etc.

Radio controllers use a simple interface to the radio and the Iris-FPGA run-time system. They can subscribe to events and register callback functions which are called whenever the corresponding event is triggered. The software wrapper concept (see Section 2.2.4 on page 57) ensures that the controller uses the same mechanisms for hardware and software components; low-level details are abstracted.

Iris-FPGA is therefore an easy-to-use cognitive radio framework that supports FPGA hardware components as well as standard software components. It allows radio designers to focus on the important parts, everything else is handled by the framework.

Portability between platforms and architectures

Iris-FPGA can be compiled for different processor architectures and operating systems from a single codebase, through the use of portable C++ for programming and the GNU Autotools [59] building. This makes it easy to maintain the code, develop new code, and execute the same algorithms on different architectures. When designing radios for Iris-FPGA, it is possible to develop and test the software components of the radio on a standard PC. This can reduce the development time and simplifies testing.

Thanks to designing a virtual architecture (see Section 2.2.2 on page 46) which can be mapped to a variety of FPGA platforms, the proposed framework can be ported to different boards without difficulty. The elements of the virtual architecture can be mapped to the real hardware elements existing on the board of interest and the framework can be executed on this board. This proved beneficial when moving from the Xilinx University Programme (XUP) Virtex-II Pro board [60] used for the demonstrators in sections 3.1 to 3.3, to the ML507 Virtex 5 board [61] used for the demonstrator in Section 3.4.

Thus, the proposed Iris-FPGA framework can be executed on a wide range of platforms.

Efficient combination of hardware and software

The proposed framework provides a full standard Linux operating system to the radio designers, easing file access and allowing to exploit other software tools as part of a radio from software components. When fast signal processing is required, FPGA hardware can be integrated into system and reconfigured at run-time without difficulty; the low-level aspects are managed by the framework. This combination proved powerful, as seen from the discussions in Chapter 3. Thus, *Iris-FPGA combines the advantages of the hardware and software worlds in one system.*

Software suitable for embedded systems

Embedded systems, such as the FPGA-based development boards used during the experiments with the proposed framework, are often constraint in processing power, storage, and memory size. Typically, the operating system is configured to contain only a minimum set of tools and libraries to reflect that. This means for a cognitive radio software that it should make careful use of resources and should not be dependent on many software libraries and applications.

The software library support required to run Iris-FPGA is minimal, only the standard C/C++ libraries are needed. Parsing the XML files can be performed on a host PC and transferred to the target system in an intermediate representation which can be parsed without external libraries (see Section 2.2.1 on page 41). Iris-FPGA is a relatively small software package (less than 8 MB, including all available components).

Thus, the software required for Iris-FPGA is ideally suited for embedded systems.

Exploitation of dynamic partial reconfiguration of FPGAs

Given its limited usage in industry, dynamic partial reconfiguration is certainly considered an advanced feature of Xilinx FPGAs. It requires specific expertise and tool support to use this feature efficiently. The proposed framework allows exploiting this without requiring expertise by the radio designer; it is managed automatically.

FPGA hardware components in Iris-FPGA can be reconfigured on-the-fly using dynamic partial reconfiguration of FPGAs. The low-level actions required for this reconfiguration are handled by the software wrapper component, which can be generated largely automatically (see Section 2.2.4). This component provides the same interface as any other software component, enabling seamless access and reconfiguration of hardware from the radio controller.

When the framework was moved to the newer ML507 FPGA board [61] for the demonstrator presented in Section 3.4, this feature was temporary disabled due to insufficient tool support. However, with the new version of the Xilinx tools released in April 2010 [44], enabling this feature is not difficult. This was proved by the other demonstrators from Chapter 3, implemented on the XUP Virtex-II Pro development board [60].

Thus, the dynamic partial reconfiguration feature of Xilinx FPGAs can be exploited in cognitive radios, adding run-time reconfigurability to high-speed hardware components.

4.1.2 Weaknesses

Through the development of the demonstrators in Chapter 3 a number of weaknesses of the proposed framework could be identified. They are:

- Dynamic behaviour not captured by radio description
- Limited flexibility in dataflow graphs
- Simple sequential execution
- Limited reconfigurability
- High latency
- Performance bottleneck in hardware/software interface
- Mixed design and execution process
- Testing and debugging is difficult

Each of these weaknesses is explained in more detail in the following.

Dynamic behaviour not captured by radio description

While dataflow graphs can be applied to describe static radio systems, they do not capture the dynamic nature of cognitive radios well. That is, they can only represent the state of the radio at one moment in time but not how it can be reconfigured. In order to fully describe a cognitive radio system with all its configuration options, many flow graphs have to be used.

The XML radio description of the proposed framework only represents the start configuration of the cognitive radio. When radio controllers are allowed to modify it in many different ways during execution, designs cannot be fully tested. This might result in unforeseen run-time errors which in the worst case can cause a crash of the radio system. It is therefore desirable to find a method to describe the radio's configuration options during the design phase, so that the radio can be thoroughly tested.

Limited flexibility in dataflow graphs

Due to the structure of the XML files, the internal representation of the graph within Iris-FPGA, and the simple scheduler, only simple types of flow graphs are supported. That is, all data flows from a source to a sink component, feedbacks (cycles in the graph) are not permitted. Components can have multiple input and output ports, but only if all are of the same type and support the same block size of data per run. This is due to the fact that the type and size information of the input and output ports is not differentiated for each port, it applies to

all ports at once. This also limits branching and merging. Branches are only possible if the branching component has multiple output ports with the same type and block size; merging is only possible if the merging component has multiple input ports with the same type and block size.

Due to the simple first in, first out (FIFO) like interface (see Figure 2.3 on page 52), hardware blocks can only be connected together in a linear chain. Branches, merges, and feedbacks are not possible for hardware chains represented by a single software wrapper (i.e., executed in one customisable processing subsystem (CPSS)).

Simple sequential execution

The Iris-FPGA scheduler is very simple. It executes all components in a graph in a round-robin fashion, calling each component once per scheduling cycle. This avoids buffering of data between components (only one block of data can ever exist per link), allows static memory allocation before the radio is executed, keeps the execution overhead low, and avoids synchronisation issues.

However, components cannot change the size of the data blocks they output per call, exactly one input block has to be processed and exactly one output block created, and the block sizes must be known before execution begins. Furthermore, this scheduler is inherently sequential. Only one component is active at a time while the rest of the graph is idle. This is efficient on a single-core general purpose processor (GPP), but becomes inefficient when parallel processing resources are available, e.g., a multi-core GPP or when FPGA hardware is involved. For instance, while FPGA hardware is processing, the processor is idle and waits for the execution to finish before it can proceed invoking other components.

This limits the types of radios which can be created using Iris-FPGA, as some radio components require flexible block sizes. For instance, a deframer component that can handle different frame sizes depending on the `payload_length` field in each frame's header cannot be supported by the Iris-FPGA framework. Additionally, the processing resources available are not used efficiently due to the lack of parallelism in the scheduler.

Limited reconfigurability

Controllers in Iris-FPGA can only efficiently reconfigure parameters of running components, or replace components with different ones. The replaced component must have the same input/output block sizes and data types as the component it is replaced with. Components

cannot be added or removed from the radio during execution. Further, parameters of running components can only be reconfigured if they do not have an effect on the data sizes produced during one process invocation. For instance, reconfiguring the upsampling factor of an upsampler component is not allowed, as this translates to a different amount of data produced at the upsampler's output.

This is due to the simple static scheduler in Iris-FPGA (see above) which does not support reconfiguring the sizes of data blocks once the radio is executing. Note that this has been achieved for some demonstrators, e.g. [89], but only through “hacking” the Iris run-time system for this demo. It is not naively supported.

High latency

When running the demonstrations described in Chapter 3 which use video streaming for the application data, it was discovered that latencies of around one second appear between playing the video at the transmitter and at the receiver. Part of this delay is caused by the connection of the radio front-end (Universal Software Radio Peripheral (USRP) [65]) through a PC bridge via TCP due to a lack of USB 2.0 connectors on the used FPGA boards. The TCP implementations use large buffers to accomplish high throughput, which increases latency. Further buffering is introduced by the USRP itself, to maintain continuous transmission. The video player and streaming also uses buffers to allow a smooth display of the video. However, some of the latency is also caused by the time it takes to process the data in Iris, transferring it between components, to hardware, and back to software, etc. This is difficult to avoid in a software implementation without sacrificing throughput.

These high latencies make it difficult to implement efficient MAC layers due to the short response times required. Furthermore, the latencies vary due to asynchronous scheduling of the operating system, which complicates real-time MAC implementations further. A comprehensive discussion about the latency issues present in software implementations of radios in general and their implications for MAC layer implementations is given in [90].

Performance bottleneck in hardware/software interface

Processing data in FPGA hardware is in many cases orders of magnitude faster than doing the same in software on a GPP. However, if hardware and software are combined, the performance bottleneck lies in the interface between both elements.

In the demonstrators built in Chapter 3, approximately 90 % of the processing time on the FPGA board was spent in the transfer of the data from software to hardware and back. The PowerPC moves the data from the external memory into the embedded memories in hardware which takes considerable time.

For the demonstrators, this performance bottleneck is partly due to the fact that the USRP radio front-end is connected to the board using a PC bridge, rather than a direct connection of the front-end to hardware. The highest data rate in the radio chain is directly before or after the front-end. For example, if rate 1/2 convolutional coding, differential quadrature phase shift keying (DQPSK) modulation, and pulse shaping with 4 samples per symbol is used, each bit is converted into 4 complex signal samples at the transmitter. If each sample is encoded as 2 16 bit fixed-point numbers, the data rate is increased by a factor 128 towards the radio front-end. This illustrates that the majority of the time at the hardware/software interface is spent on the front-end side, which could be avoided if the front-end interfaced directly with the hardware chain.

Another reason for the relatively low performance of the demonstrators is that the data is transferred element by element by the PowerPC processor. This is comparably slow. If direct memory access (DMA) was used, these transfers could be accelerated significantly. If a DMA block in hardware is responsible for the data transfer from external memory into the embedded memories and back, all the processor has to do is set up the DMA with the memory addresses and initiate the transfer. The processor is then free to perform other tasks while the transfer is in progress, and a dedicated DMA can work considerably faster than the processor in transferring the data.

Note that the lack of front-end connectivity and DMA is not a limitation of the Iris-FPGA framework architecture, it is merely an implementation issue that can be fixed with some effort without architecture violation. However, even when both methods were implemented, data transfers between hardware components and external memory would still be considerably slower than direct transfers between hardware components. For this reason, switching between hardware and software should be kept at a minimum, chaining hardware components together as much as possible.

Mixed design and execution process

In Iris-FPGA, the execution and design environments are identical. That is, constructing and checking of the radio flow graphs has to be performed directly on the target system.

This consumed considerable amounts of time during development of the experiments from Chapter 3, which could have been avoided if the radio design was managed from a separate environment on a PC and deployed to the run-time system later in the process. Note that if the target system is a standard PC, this separation is not necessary as the run-time system is readily available and checks can be performed without delay.

The design environment should consist of tools for constructing and checking radio configurations and for developing radio controllers. These tools should verify that the radio is correctly constructed, i.e., that it is a valid graph executable by the run-time system. Being able to run simple tests on the design system is an advantage. The output of the design tools should be a configuration for the target run-time system that has already been checked for correctness. Generally, as many tests as possible should be done by design tools, to allow early detection of errors without the overhead of running the radio on the target platform.

Ideally, the performance on the target system could be estimated in the design environment. This would allow the radio designer to not only check for correct radio composition, but also if his design will meet the target performance.

During the development of the experiments discussed in Chapter 3, it was often the case that a radio setup had performance issues on the FPGA platform which were not foreseen during the design. Software profiling methods had to be used to identify which part of the setup is the bottleneck and needs tuning. It would be beneficial if the performance could be estimated during the design phase, in a separated design environment.

This could be achieved by thoroughly measuring the performance of the individual components on the target platform and using these measurements during design. Many iterations of testing on the target system and fixing performance problems in the design could be avoided if the performance could be estimated at design time.

Testing and debugging is difficult

This weakness is only partly a weakness of the Iris-FPGA architecture itself. In fact, it is caused by the decision to combine hardware and software into a hybrid system. Although this has many advantages, as outlined in Section 4.1.1, it makes testing and debugging more difficult.

Testing and debugging either software or hardware on its own is a relatively simple matter. For testing software, data can be read from a file, processed by the software component under test, stored in another file, and analysed using tools like Matlab. Debugging can be done using standard debuggers, e.g., the GNU Debugger (gdb) in Linux. They allow stepping through

the code and checking the component's behaviour. Hardware can be tested through hardware simulation using testbenches, i.e., simulator modules that generate data, send it through the hardware component under test, and analyse the outputs. Simulators allow to analyse all hardware signals at any time during the simulated execution, enabling the hardware designer to find and correct errors before configuring the target FPGA. Another approach to test hardware is called 'hardware in the loop testing', where data is generated in software, sent to actual FPGA for processing, and retrieved back for analysis.

Testing and debugging a combined hardware and software system is not so straightforward. The most commonly used method is to run the software part, store all the data passed to hardware in files, and develop a hardware simulator testbench which reads these files and passes the data through hardware in simulation. This approach has been followed for the development of the demonstrators presented in Chapter 3. Although this method works well for the functional tests of the system, it does not allow to test the performance of the system. Additionally, the interface between hardware and software is not tested. System developers have to test the full system on the target platform itself to obtain test results, possibly using ChipScope [91] or similar tools to record hardware signals at run-time. For the experiments described in Chapter 3, rather than setting up ChipScope, the hardware was modified to write diagnostic data to registers which can be accessed from software. This data was read from software and stored for later analysis.

For radio systems, a difficult part to test and debug is the receiver behaviour under realistic channel conditions. For example, the synchronisation components are designed to reverse the effects of the channel and the radio front-end. Those effects are changing and generally unknown; they cannot be easily reproduced or simulated. One method to overcome this is to record raw signal samples at the receiver directly after the radio front-end in a number of representative scenarios and setups. Then, the receiver can be tested offline using this data. This requires that the transmitter has been fully tested and can be assumed to be working correctly. The difficulty involved in this approach is selecting scenarios that are representative. For instance, in some of the experiments described in Chapter 3, it turned out that a receiver working perfectly with the recorded real-world data did not function reliably in the final demonstration setup. More raw data had to be recorded to reproduce and correct the problems experienced.

During the development of the experiments from Chapter 3, the majority of time was spent in the test and debug phase, often detecting errors late in the process due to bad testing strategies. As a summary it can be said that *special care should be taken in testing and debugging*

the radio system.

4.2 Approaches to Cognitive Radio Design: An Analysis

The reviewed frameworks from Chapter 1 and also Iris-FPGA are very different in nature (some are software frameworks, some are hardware platforms, and some are a combination of both). In fact, it would be possible to execute GNU Radio on the Kansas University Agile Radio (KUAR) platform, or Iris-FPGA on a Wireless Open-Access Research Platform (WARP) board. A direct comparison of the provided features is therefore not possible. However, what can be compared is the approach taken to radio design. This is very relevant to enabling cognitive radio experimental research, as sets the complexity of building demonstrator radios.

It becomes clear from the discussions in Chapter 1 that the design and development approaches used for existing cognitive and software radio frameworks differ greatly. This section suggests a way of categorising that captures the experiences with Iris-FPGA and the analysis of the other frameworks¹. It further identifies the ideal characteristics of a cognitive radio design process in order to enable and simplify experimental research.

4.2.1 Level of Abstraction and Model of Computation

One way to categorise the frameworks is based on the level of abstraction of the design approach. A high level of abstraction means that radios can be designed without detailed knowledge of the platform’s implementation details. When designing radios for platforms providing a low level of abstraction considerable knowledge of the implementation details is essential. Table 4.1 shows which frameworks fall into these categories.

¹As Iris-FPGA is an extension of Iris in the version from 2006 (discussed in Section 1.3), Iris is not listed separately in the following sections.

High-Level	Low-Level
Iris-FPGA [69]	KUAR [43]
GNU Radio [36]	WiNC2R [48]
JTRS SCA [14]	Stalion [51]
	WARP [52]

Table 4.1: Frameworks categorised by level of abstraction during radio design.

It is greatly beneficial to use a framework or platform which provides a high level of abstraction since this reduces the development time, simplifies testing, and provides easier access for new users. Analysing the high-level design frameworks it becomes apparent that all provide a library of re-usable signal processing components and compose radios by connecting these together, i.e., they apply *dataflow-based design*. This intuitive and established approach for describing signal processing systems (see for example [85–87]) decouples the signal processing implementation from the system composition. It allows to compose radio systems without detailed platform knowledge.

However, in order to *efficiently* use dataflow graphs for design, the dataflow model of computation (MoC) must be known to the designer. The execution semantics, i.e., how components are executed and how data is passed between components, must be clearly defined in the MoC, ideally in terms independent of low-level implementation details. This MoC must be known when the system is composed. Some systems, for instance, might not permit feedbacks (cycles) in the graph, some might support parallel execution of components, and others might only support fixed data rates on each link. It is impossible to efficiently construct radios without knowledge of the underlying MoC.

The three frameworks providing a high level of abstraction apply dataflow-based design techniques, but do not define the applied MoC in an implementation-independent way. Thus, radio designers need detailed platform knowledge in order to design efficient radios.

The platforms shown on the right in Table 4.1 require expertise in low-level design. This allows for a high degree of freedom, allowing almost any radio system to be implemented. However, the considerable implementation effort required is often impractical for research and experimentation.

Ability to Exploit Real-Time Hardware

As discussed in the review of hardware processing platforms in Section 2.1 on page 39, GPPs are the most flexible platform that is easiest to access. However, it is also the slowest for signal processing applications, compared to the other platforms discussed. To achieve the processing performance and power efficiency required for realistic, cutting edge, real-time cognitive radio implementations, it is important that high performance computational resources, e.g., DSP processors or FPGAs, are available to the radio designer.

Most of the discussed frameworks support the integration of such hardware, as can be seen from Table 4.2. Note that all existing low-level platforms (according to Table 4.1) support

Real-Time Hardware	
supported	not supported
Iris-FPGA [69]	GNU Radio [36]
	JTRS SCA [14]
KUAR [43]	
WiNC2R [48]	
Stalion [51]	
WARP [52]	

Table 4.2: Frameworks supporting real-time hardware. JTRS SCA partly supports real-time hardware, using proprietary extensions.

Software Defined Radio	Cognitive Radio
GNU Radio [36]	Iris-FPGA [69]
JTRS SCA [14]	KUAR [43]
WARP [52]	Stalion [51]
	WiNC2R [48]

Table 4.3: Frameworks categorised by application domain.

real-time hardware, while the high-level ones provide no or limited support. The original version of Iris, without the FPGA extensions (see Section 1.3 on page 21), would also fall into this category. However, Iris-FPGA enabled access to high performance real-time hardware. Thus, there appears to be a correlation between a low level of abstraction for the design and support for real-time hardware.

4.2.2 What Application Domain is Targeted?

Another criterion to categorise the frameworks is based on the application domain. Some frameworks are primarily designed to allow software defined radio (SDR) implementations, i.e., radios that can be programmed and re-programmed using software but not necessarily reconfigure themselves at run-time, based on information about the environment and the radio itself. Frameworks for cognitive radios on the other hand include mechanisms for cognitive decision-making and run-time reconfiguration of the radio, i.e., methods to implement the cognitive cycle of *observe*, *decide*, and *act*. This does not necessarily mean that cognitive radios are impossible to implement using frameworks from the first category, but the required mechanisms are not provided and need to be implemented manually. Table 4.3 shows the categorisation by the application domain.

This thesis is concerned with cognitive radios, hence the platforms listed in the right column in Table 4.3 are of interest. However, several cognitive radio applications have been demonstrated using GNU Radio or Open Source SCA Implementation::Embedded (OSSIE). This was achieved by modifying the source code in several places of the frameworks; mechanisms for connecting a cognitive engine and for monitoring the radio are not built-in features.

4.2.3 Ability to Cope with Reconfiguration During Design

None of the frameworks discussed, including Iris-FPGA, defines a structured way of expressing reconfigurability on a high level of abstraction. The frameworks for the cognitive radio application domain (see Table 4.3) provide mechanisms for reconfiguration, but these vary greatly between frameworks and require in-depth understanding of the low-level architecture. The dataflow graphs used for high-level design (see Table 4.1) can only represent the state of the radio when it is started or at any one time in the radio’s lifetime. They do not express reconfigurability. This makes it difficult for radio designers to experiment with and reason about cognitive radios and networks.

4.2.4 Platform Summary

Table 4.4 gives a feature summary of the frameworks discussed in this chapter. It summarises the discussions from this section. The abstraction level and, for high level, the applied dataflow MoC were discussed in Section 4.2.1, the application domain in Section 4.2.2, and whether the framework can cope with reconfigurable on a high level during the design process in Section 4.2.3. The column ‘Real-Time Hardware’ states whether the framework supports the integration of fast hardware, as discussed in Section 4.2.1. The ‘Processing Device’ column states which hardware processing device is used for executing the radio with the given framework.

An ideal cognitive radio design method should

- provide a high level of abstraction (i.e., based on dataflow graphs),
- define the dataflow MoC in platform-independent terms,
- be designed for the cognitive radio application domain,
- be able to cope with reconfiguration on a high level during design, and
- be able to target a platform that can integrate real-time hardware.

This would allow fast prototyping of advanced cognitive radio systems which can be executed in real-time, this, it would help enabling cognitive radio experimental research. As can be seen

Platform/ Framework	Abstraction Level	Model of Computation	Application Domain	Reconfiguration during Design	Real-Time Hardware	Processing Device
Iris-FPGA [69]	high	undefined	CR	no	yes	GPP/FPGA
JTRS SCA [14]	high	undefined	SDR	no	partly ¹	GPP/DSP/FPGA
GNU Radio [36]	high	undefined	SDR	no	no	GPP
KUAR [43]	low	N/A	CR	no	yes	FPGA/GPP
WiNC2R [48]	low	N/A	CR	no	yes	custom silicon
Stalion [51]	low	N/A	CR	no	yes	custom silicon
WARP [52]	low	N/A	SDR	no	yes	FPGA/GPP

N/A : not applicable; SDR : software-defined radio; CR : cognitive radio;
¹ “partly”: proprietary extensions allow integration of real-time hardware

Table 4.4: Feature summary of the discussed platforms.

from Table 4.4, none of the platforms/frameworks provides all these features.

The analysis of this chapter led to the development of a novel approach to cognitive radio design, defined on a high level of abstraction, which can be used to design radios for any of the frameworks discussed in this chapter. This is presented and evaluated in the following chapters (in Part II of this thesis). It overcomes the limits of the traditional design approaches discussed in this chapter and provides all features mentioned above.

Part II

Model-based Cognitive Radio Design

5 Model-based Design: Background

Contents

5.1 Introduction and Terminology	114
5.2 Existing Dataflow Models of Computation	118
5.3 Existing Control Flow Models of Computation	136
5.4 Summary	141

The lessons of Part I of this thesis told that developing cognitive radio prototypes for experimentation is a cumbersome task that requires considerable engineering work. However, the motivation behind conducting experimental research is to test theories and ideas for cognitive radios in the real world, the development of prototypes is only a necessary and unavoidable step on the way. Thus, the key enabler for cognitive radio experimental research is a fast, simple, and effective approach to designing radios. Following the experience and insights gained from Part I, this part of the thesis (Part II) proposes a method for simple and efficient cognitive radio design.

The proposed design approach meets the goals outlined in Section 4.2.4, i.e., it provides a high level of abstraction, is largely platform-independent, can cope with the reconfigurability of cognitive radios, and it supports the integration of real-time hardware. It is not bound to a specific target platform or framework, i.e., it can be used to design radios for arbitrary platforms, e.g., GNU Radio [36], Kansas University Agile Radio (KUAR) [45], or Iris-FPGA (Section 2.2). Thus, it enables the transfer of designs between researchers and platforms, so that experiments can be repeated easily. It is believed that this can enable more experimental research in the field of cognitive radio.

A promising approach to achieve platform-independent designs is *model-based design*.

That is, instead of developing software or hardware directly for the target platform, the system is modelled using a domain-specific platform-independent model first. This model can then be implemented with the help of automatic tools.

This chapter gives background information about model-based design, both in general and specific to cognitive radios.

5.1 Introduction and Terminology

Modelling applications and systems using domain-specific models is a widely-adopted and successful approach in many areas. Instead of converting a set of requirements directly to an implementation, it is converted to a model of the application. This abstracts low-level implementation details and focuses on the application's behaviour. Often this model can then be converted to a real implementation using automatic tools, involving little or no user interaction. This enables fast prototyping, easy understanding of the system behaviour, and easier tracking of bugs and errors.

Model-based design is employed by many commercial tools for different domains. For example, Unified Modeling Language (UML) is used extensively in industry to model object-oriented software [92]. It can describe the architecture and interaction between software components, without specifying how they are implemented. Another example is MathWorks Simulink [86]. It can be used to create real-time signal processing systems based on a graphical signal flow model (using the Real-Time Toolbox [88]).

Typically, application or system models define an underlying model of computation (MoC), i.e., their execution semantics. This is essential to be able to assess how the model is executed on the final system without being concerned with implementation details. Effective model-based design requires an understanding of the underlying MoC. Section 5.1.1 defines the term MoC and discusses its implications in more detail.

Using a dataflow graph to model digital signal processing (DSP) systems is a model-based design technique. It is domain-specific, i.e., it is well-suited to the DSP domain of applications. For instance, it cannot be applied to model object-oriented software in general, or to other application domains. Limiting the model to one domain allows abstracting common properties of that domain.

The DSP domain refers to a set of applications and algorithms which focus on manipulating digital signals, such as sound, images, videos, or communication signals. A stream of signal

data is manipulated by these algorithms, transforming them into a different format. Examples are a video codec or an orthogonal frequency division multiplex (OFDM) receiver. Dataflow graphs are ideal for this application area, as they describe how the data is streamed through a system and what algorithms are applied to it. The implementation details of the algorithms are hidden, making the model implementation-independent.

The main part of software defined radios (SDRs) and cognitive radios is essentially a DSP system. Signals received from the radio-front-end are filtered, demodulated, decoded, analysed, etc. Thus, dataflow graphs can be used to model this part of cognitive radios. In Section 5.1.2, a description of dataflow graphs is given.

The second part of SDRs and cognitive radios is concerned with radio control. This manages the execution, monitors external inputs and outputs, initialises the system, and manages its configuration. The two parts (DSP and control) are very different in their nature, with the former characterised by computations on streams of data and the latter by control flow of arbitrary complexity. Therefore, different models have to be applied to the control part. Section 5.1.3 discusses the control part of a radio in more detail.

5.1.1 Models of Computation

Every application model should define an underlying model of computation (MoC), which describes the execution semantics, i.e., how the model is executed in a real system. Generally cognitive radios are modelled as concurrent applications, since parts of the radio can execute in parallel. For instance, while a matched filter is processing an incoming data block, a demodulator can process the previous block concurrently, given available processing resources. A useful framework to analyse the execution of a concurrent application is a concurrent MoC. According to Lee [84], a MoC can be defined as:

A model of computation is the “laws of physics” of concurrent components, including what they are (the ontology) how they communicate and how their flows of control are related (the protocols), and what information they share (the epistemology). It is their concurrent semantics.

A MoC should abstract the hardware execution as much as possible in order to describe the semantics of the application execution without relying on a particular target platform. That is, the MoC does not include implementation-specific details.

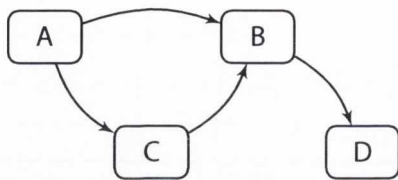


Figure 5.1: A dataflow graph with actors A, B, C, D. Channels are represented by directed arcs.

5.1.2 Dataflow Graphs

Typically the data processing domain of DSP applications is modelled using dataflow models, i.e., a series of operations is applied to flows of data. Such models have proved hugely popular and successful in the past, as they are natural to DSP engineers and expose the inherent parallelism in the algorithm. They can be effectively described graphically, using *dataflow graphs*, or alternatively in a dataflow programming language, e.g., SISAL [93] or SAC [94]. The algorithms described in a dataflow programming language can generally be transformed into a dataflow graph and vice versa. Hence, focusing on dataflow graphs in the following is sufficient.

A dataflow graph is a directed graph, composed of *actors* (nodes) and data *channels* (edges)¹. Each actor performs some operation on data *tokens* which flow along the channels. No direct communication between actors is permitted, the only allowed communication is through the channels. An example dataflow graph is shown in Figure 5.1.

At least conceptually (and possibly actually) the actors in the dataflow graph execute concurrently, each operating on different data tokens. Thus, dataflow graphs expose the inherent parallelism in a particular algorithm.

Dataflow graphs are most efficient for describing algorithms based on a chain of operations applied to streams of data. DSP algorithms fit into that category. Note that general purpose applications do not generally fit; for instance, graphical user interfaces cannot efficiently be described by a dataflow graph.

When dataflow graphs are used to describe systems, a MoC should be defined, describing how the graph is executed on a real systems. Section 5.2 reviews various dataflow MoCs relevant to cognitive radio.

In the following, descriptions of each entity in a dataflow graph are given.

¹In the literature, actors are sometimes called *processors*, *operations*, *components*, or *kernels*. Other names for channels include *buffers*, *links*, or *data paths*.

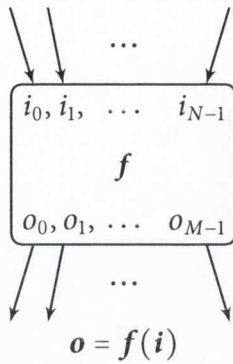


Figure 5.2: Actor with input vector $\mathbf{i}$, output vector $\mathbf{o}$, and actor function $\mathbf{f}$.

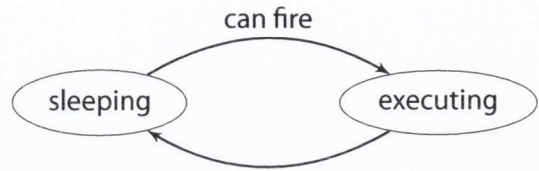


Figure 5.3: States of an actor.

Actor

An actor represents a process that operates on tokens. It is characterised by its inputs $\mathbf{i} = [i_0, i_1, \dots, i_{N-1}]$ and outputs $\mathbf{o} = [o_0, o_1, \dots, o_{M-1}]$, where N is the total number of inputs and M is the total number of outputs. Each input and output is connected to an inbound and output channel, respectively. The actor's functionality is described by a vector function $\mathbf{f}$. A graphical representation is given in Figure 5.2 [95].

An actor can be in one of two states, *sleeping* or *executing*. It defaults to *sleeping*, and only switches to *executing* when a set of firing rules are fulfilled. Once the execution of the actor function $\mathbf{f}$ is finished exactly once, the actor goes back to sleep and waits for the firing rule to become fulfilled again. The firing rule can be defined in different ways, depending on the used MoC. An example is a rule evaluating to true if and only if there is data available in all of the actor's input channels. A state diagram for an actor is shown in Figure 5.3.

The terms *predecessor* and *successor* are defined in the following theorem.

Definition 5.1. Consider two actors A and B . If there is a channel directing from A to B in the flow graph, A is *predecessor* of B and B is *successor* of A .

Channel

A channel is a direct link between an output of one actor and an input of another. It transports data tokens between actors.

Definition 5.2. If a channel c connects an output of actor A with an input of actor B , A is called the *source actor* of c and B is called the *sink actor* of c . Each channel has exactly one source and one sink actor.

A channel implements first in, first out (FIFO) semantics, i.e., it has memory to store multiple data tokens. The sink actor consumes data from the top of the FIFO and the source actor pushes data to the bottom of it.

Token

A token represents one data item. It has a type, for instance a signal sample, a bit, or a packet. Tokens are consumed and produced by actors and stored on channels. Different types of tokens are allowed in a dataflow graph, with one specific type for each channel.

5.1.3 Control Structures

Control structures focus on the representation and specification of the control-oriented aspects of the system. It is the part of the application that is driven by events rather than data and that produces control information rather than output data. Events and control information typically travel through the system at a significantly lower rate than data. It is used to obtain system status information or change the system's behaviour.

In a cognitive radio system, control structures are needed to model the connection to the cognitive engine, i.e., the decision-making and learning process of the radio. The cognitive engine needs system status information as inputs (the meters) and produces reconfiguration actions, i.e., it changes the system's behaviour (the knobs). Control structures can also be used to model the decision-making itself.

Models for expressing control flow are more loosely defined than data flow models. A wide range of models falls into this category. In fact, even dataflow graphs can be used to model the flow of control information through the system. Some relevant models for cognitive radios are reviewed in Section 5.3.

5.2 Existing Dataflow Models of Computation

The execution semantics of a dataflow graph algorithm are described by a dataflow MoC (see Section 5.1.2). Defining the MoC allows understanding the semantics of a dataflow graph, analysing the computational resources required, and understanding the model's limitations.

For DSP systems, such as the processing part of cognitive radios, a number of established MoCs exist. This section reviews a number of these which are considered relevant. Their

properties are given and their suitability for cognitive radios is assessed.

Suitable MoCs for dataflow graphs are based on asynchronous message passing, i.e., the sender actor of the message (token) need not wait for the receiver actor to be ready. This is achieved by the buffering capability of the FIFO channels.

There are several MoCs for asynchronous message passing, some are deterministic and some are not. In deterministic MoCs, the values of the data tokens in the whole graph can be determined at all times, with no unknowns. Executing the same graph twice with the same initial data always yields the same behaviour. Determinism is a basic requirement for all DSP algorithms, thus, we only focus on deterministic MoCs here.

5.2.1 Process Networks

The process networks (PN) is the most general MoC for dataflow graphs. It is based on the original publication by Kahn [96]. In this model, actors are loosely coupled processes, with some conditions to ensure that the behaviour of the process network is deterministic. In fact, Kahn process networks (KPNs) apply to a wider area of applications than dataflow. In the following, a description of KPNs is given, introducing some formalism and properties (based on [96, 97]). Then, dataflow PNs are defined, based on the more general KPNs.

Kahn process networks overview

A KPN is a network of processes, connected through channels holding sequences of data. It can be thought of as a network of Turing machines [98, 99] connected through one-way tapes, where each machine has its own working tape [96]. A KPN is therefore *Turing complete* [100], i.e., all algorithms implementable on a theoretical Turing machine can be realised in a KPN.

All processes run concurrently and asynchronously, and are connected through one-way FIFO channels with unbounded capacity. Each channel carries a possibly infinite sequence of data tokens (a *stream*), where each data token is produced and consumed only once. Writes to a channel are always non-blocking, i.e., a write always succeeds and returns immediately. Reads may be blocking, i.e., if an input channel has no data available, the reading process blocks until sufficient data becomes available. A process in a KPN represents a mapping from one or more input sequences to one or more output sequences through a mapping function f (see Figure 5.2). This mapping function must be *continuous* and *monotonic*, as derived by Kahn [96]. The derivation of the terms *continuous* and *monotonic* function in the KPN sense is given in Appendix D

The key consequence of these properties is that a monotonic process can be computed iteratively. A monotonic process does not need all inputs before it can compute the output. Given a part of the finite input sequences, it can compute a part of the output sequences. If the input sequences are infinite, a process does not need to wait forever before it can compute the output sequences. This property ensures that we can get an execution of the process network, concurrently computing the process mappings on parts of the data streams.

Kahn also shows that a network of monotonic processes is deterministic [96]. That means, all input history on all channels of the network uniquely determines all outputs of the network, with no unknown or uncomputable values. The conditions to ensure that a network is deterministic (and monotonic) are:

1. processes are not permitted to test inputs for emptiness (ensured by blocking reads),
2. processes are deterministic internally,
3. only one process can write to one channel,
4. only one process can read from one channel, and
5. processes do not share variables.

That is, the only allowed communication between processes is through blocking reads and unblocking writes on their associated channels.

Kahn process networks for dataflow

To apply KPNs to dataflow networks, Kahn processes have to be mapped to dataflow actors (see Section 5.1.2). A dataflow actor, when it fires, maps input tokens to output tokens through a function f (Figure 5.2). A set of firing rules determines when an actor can fire. They specify precisely what type and number of tokens must be available on the input channels before the actor function can be evaluated. During one firing, input tokens are consumed and output tokens are produced; a sequence of firings maps input sequences to output sequences. This is a particular type of KPNs, called a *dataflow process network* (PN) [97].

A sufficient condition for dataflow process networks to be continuous is that each actor firing is *functional* and that the set of firing rules are *sequential* [97]. An actor firing is functional if it can be executed without side effects, i.e., the actor does not have state, and if the same input sequences always lead to the same output sequences. The functional condition can be relaxed

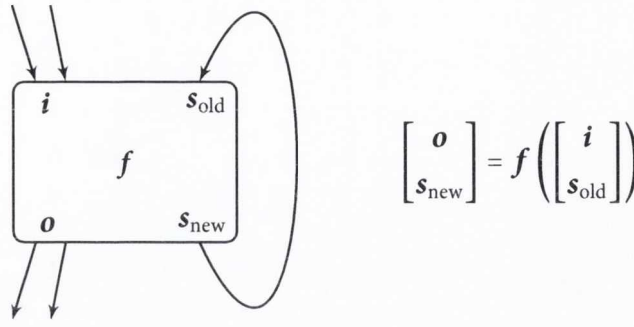


Figure 5.4: An actor with state. This state could be, for instance, a delay-line of a filter, keeping a history of previous inputs.

by treating an internal actor state as a feedback loop, i.e., modifying the actor function as

$$\begin{bmatrix} o \\ s_{\text{new}} \end{bmatrix} = f \left(\begin{bmatrix} i \\ s_{\text{old}} \end{bmatrix} \right) \quad (5.1)$$

where s_{new} represents the updated state of the actor, based on the old state s_{old} . Figure 5.4 illustrates an actor with state.

Firing rules represent conditions on the input sequences on each input port of an actor, specifying when the actor function can be applied. Firing rules are *sequential* when they can be tested in a predefined order using only blocking reads from input channels without causing a deadlock. For example, a rule stating that at least one input token must be available on each of the input channels can be tested by blocking reads only without causing a deadlock. However, a rule of an actor with multiple inputs stating that at least one input must be available on any one of the input channels is not sequential. Attempting to read from one channel without available data tokens will block the process, although the firing rule is satisfied when there is data available on another input. Such rules are not permitted in a PN. If permitted, they could deadlock the graph or cause non-deterministic behaviour. A formal definition of sequential firing rules is given in [97].

The definition of the PN MoC is very general, allowing arbitrary dataflow algorithms to be described in this model. Generally, the execution of such graphs must be scheduled at run-time, either by implementing each actor in a separate asynchronous process with unbounded FIFO buffers for the channels, or by some other scheduler which implements the same semantics. For this reason the PN MoC is sometimes called dynamic dataflow (DDF), with ‘dynamic’ referring to run-time scheduling.

Since they are a network of Turing machines, dataflow PN are affected by the well-known

halting problem of Turing machines. That is, it is generally undecidable whether or not a Turing program will halt. Dataflow PN are allowed to run forever, but the halting problem means that it cannot be determined within finite time whether or not an arbitrary PN is executable. Further, it cannot be decided whether the network is executable within finite memory, as required on any real system (the unblocking writes might result in buffers growing infinitely).

Lee and Parks present a solution for executing dataflow PN that allows to run a large subset of theoretical PN on real machines within bounded memory [97]. They propose to limit the buffer sizes of each channel to a predefined limit and make writes to the channels blocking if that limit is reached. If the graph deadlocks, i.e., all processes are blocked, the size of the smallest buffer with an attached write-blocked process is doubled and the execution is resumed. This is repeated until the buffer sizes reach a maximum, upon which it is decided that the graph cannot be executed further and the system is aborted. Many PN can be executed forever without increasing the buffer sizes, and the methods proposed in [97] achieve an execution on a real machine. Note that, if the graph execution fails at some point using this method, it does not mean that the network is not executable in general. It only means that this execution method cannot proceed any further.

Dataflow PN is the most general MoC for dataflow graphs. More restricted MoCs are synchronous dataflow (SDF) [101], scalable synchronous dataflow (SSDF) [102], cyclo-static dataflow (CSDF) [103], boolean dataflow (BDF) [104], and core functional dataflow (CFDF) [105]. These allow thorough analysis of the graph before execution. Guarantees for executability in bounded memory can be given and static schedules can be computed prior to execution, resulting in efficient graph execution. This comes at the cost of reduced flexibility compared to PN. In the following, a description of the mentioned MoCs is given. The SDF MoC is described in more detail than the others, since this is the most widely used and most relevant MoC for DSP besides PN, and an overview is given for the other MoCs.

5.2.2 Synchronous Dataflow

The synchronous dataflow (SDF) [35, 101] MoC is a special case of PNs where the number of data tokens consumed and produced on each invocation of an actor function is known a-priori and fixed. This additional requirement simplifies the semantics of a dataflow graph significantly. Compared to PNs, SDF has the advantage that in-depth analysis is possible without executing the graph. In particular:

- analysis for potential deadlocks,

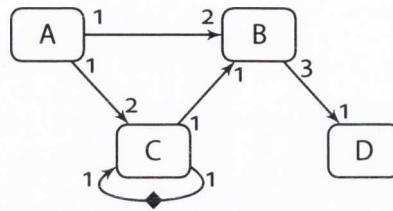


Figure 5.5: An example SDF graph. The numbers indicate the number of data tokens consumed/produced per firing, the diamond indicates a delay on the channel, i.e., one token stored on the channel when the graph is initialised.

- checks for graph consistency,
- computation of static execution schedules, and
- computation of buffer sizes for channels (i.e., execution in bounded memory possible).

These advantages come at the cost of reduced flexibility:

- actors producing or consuming varying amounts of tokens on each firing are not allowed,
- conditional dataflow is not permitted (e.g., a select or switch actor), and
- firing rules cannot depend on data values on input channels.

Despite the reduced flexibility, a large number of DSP algorithms can sufficiently be described using the SDF MoC. Computing static schedules and fixed buffer sizes at compile-time allows for dramatic optimisations and a small run-time overhead, yielding high execution performance. This makes SDF a particularly attractive solution for real-time DSP systems, such as SDR.

Overview

An example SDF graph is shown in Figure 5.5. The numbers on the arcs represent the number of data tokens consumed or produced during one actor firing. For example, actor A in the figure produces 1 data token on each of the outbound channels, while actor C consumes 2 tokens from its first input, 1 token from its second, and produces 1 token on each of its two outputs.

The firing rules in SDF graphs are simple, with exactly one rule per actor. For instance, for actor C the rule states that 2 tokens must be available on input 1 and 1 token on input 2. Note that a rule cannot specify what value these tokens must have. It is always a single rule consisting of the required number of tokens on each input channel. Therefore, the firing rules of each actor in Figure 5.5 are fully specified by the numbers in the figure. Since there is only one firing rule per actor, the rules are sequential, ensuring that the actors implement continuous functions.

Hence, SDF graphs are always deterministic and monotonic. (See the PN MoC explained in Section 5.2.1 for a description of these terms.)

The diamond in the feedback loop in Figure 5.5 symbolises a unit *delay*. This means that one token is stored on that channel before startup, initialised to some value. The notion of delays is necessary to ensure that actor C can execute at startup. Without it, C's firing rule (1 token on the feedback input and 2 tokens on the other input) could never be satisfied, causing the graph to deadlock. Multiple delays are allowed on a channel, often signified with a multiplicity number next to the diamond.

Although the SDF graphs are always monotonic and deterministic, the execution might still deadlock or run in unbounded memory. In [101], Lee and Messerschmitt derive methods to analyse whether an SDF graph is deadlock-free and can execute in bounded memory. In the following, a summary of these findings is given.

Actors cannot fire in arbitrary orders due to the firing conditions. A *schedule* determines when and where (on which processor) actors fire. A scheduling cycle is a sequence of actor firings after which the graph is back in its original state. That means, the channels carry the same amount of data as before the schedule starts. An example single processor schedule for the graph given in Figure 5.5 is

$$\Psi = \{A, A, C, B, D, D, D\}, \quad (5.2)$$

where each letter in the schedule Ψ represents one firing of the corresponding actor. This scheduling cycle can be repeated forever in order to process an infinite stream of data, and the buffer memory required for the channels is bounded. For example, the channel between actors A and B contains two data tokens maximum when the above schedule is applied. Thus, the memory requirements for the graph execution can be computed at compile-time. Blocking reads (and the possibly associated context switch on a processor), a dynamic scheduler, and dynamic memory allocation can be avoided. This results in significantly less overhead than implementing the same graph with the PN MoC.

However, it is possible that SDF graphs are not consistent. Consider, for example, that the actor C would output 2 tokens per firing on the channel between C and B. This would result in a infinitely growing buffer on that channel, an effect generally referred to as *inconsistent sample rate*. Without the delay element on the feedback channel of C the graph would deadlock. This is referred to as *insufficient delays*. If an SDF graph is not consistent, no cyclic schedule exists that brings the number of data tokens on each channel back to their initial state. In the

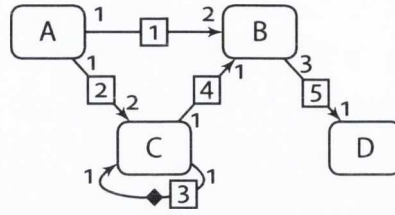


Figure 5.6: Graph from Figure 5.5 with numbered channels (the numbers are framed).

following, an algorithm for finding a cyclic schedule for arbitrary SDF graphs is given. If the algorithm fails, i.e., no such schedule exists, the graph is inconsistent.

Formal description

An SDF graph can be described by a matrix similar to the incidence matrix used for directed graphs in graph theory. It is constructed by first numbering each actor and channel in the graph. Here, the actors A, B, C, D are numbered with 1, 2, 3, 4, respectively. The channels are numbered as shown in Figure 5.6. Then a column is assigned to each actor and a row to each channel. The (i, j) -th entry in the matrix represents the amount of data produced by an actor j on channel i on each firing. Consumed data is denoted by a negative number. If actor j is not connected to channel i , the corresponding matrix element is zero. This yields to the following matrix for the graph shown in Figure 5.6:

$$\mathbf{F} = \begin{bmatrix} 1 & -2 & 0 & 0 \\ 1 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 3 & 0 & -1 \end{bmatrix} \quad (5.3)$$

The matrix $\mathbf{F}$ is called *topology matrix*.

Each channel stores a varying amount of data tokens during execution. Let the vector $\mathbf{b}(n)$ contain the number of tokens at time index n on each channel (represented by the elements of $\mathbf{b}(n)$).

For a sequential schedule only one actor can be executed at a time and the execution time of one firing does not matter. Therefore, the time index n can simply be incremented by one with each firing. Further, the active node at time n is denoted by a vector $\mathbf{v}(n)$, which contains a '1' in the row corresponding to the active actor and zeros otherwise. In our example graph $\mathbf{v}(n)$

contains 4 elements. Each time an actor is fired it consumes data from the associated input channels and produces data on the associated output channels. This change in the channel buffer sizes is given by

$$\mathbf{b}(n+1) = \mathbf{b}(n) + \mathbf{\Gamma}\mathbf{v}(n). \quad (5.4)$$

The topology matrix $\mathbf{\Gamma}$ therefore characterises the effect of one invocation on the buffer sizes of the channels.

To initialise the recursion given in (5.4), $\mathbf{b}(0)$ is set to the number of delays on each channel, i.e., in the given example

$$\mathbf{b}(0) = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}^T. \quad (5.5)$$

Checking for inconsistent sample rates

Using the topology matrix, an SDF graph can be checked for inconsistent sample rates. Inconsistent sample rates preclude the construction of a periodic schedule, executable within bounded memory. A necessary condition for the existence of such a schedule is that $\text{rank}(\mathbf{\Gamma}) = s - 1$, where s is the number of actors [35]. The topology matrix for the example graph in Figure 5.6 has rank 3, hence, this condition is fulfilled.

For every topology matrix $\mathbf{\Gamma}$ with rank $s - 1$ there exists a strictly positive integer vector $\mathbf{q}$ in its right nullspace, as proven in [35]. Thus,

$$\exists \mathbf{q} \in \mathbb{N}_+^s, \text{ such that } \mathbf{\Gamma}\mathbf{q} = \mathbf{0}, \quad \text{if } \text{rank}(\mathbf{\Gamma}) = s - 1. \quad (5.6)$$

For our example graph, the set of such vectors is given by

$$\mathbf{q} = J \begin{bmatrix} 2 & 1 & 1 & 3 \end{bmatrix}^T, \quad (5.7)$$

for any positive integer J . The elements of $\mathbf{q}$ specify the number of times the corresponding actor must be invoked in one cycle of a periodic schedule. That is, invoking (5.4) repeatedly with $\mathbf{v}$ vectors corresponding to the number of times indicated by $\mathbf{q}$ returns $\mathbf{b}$ to its original

state. For our example:

$$\begin{aligned}
 \mathbf{b}(7) &= 2\Gamma \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \Gamma \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \Gamma \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + 3\Gamma \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + \mathbf{b}(0) \\
 &= \Gamma \left(2 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + 3 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right) + \mathbf{b}(0) \\
 &= \Gamma \mathbf{q} + \mathbf{b}(0) \\
 &= \mathbf{0} + \mathbf{b}(0) \quad \text{see (5.6)} \\
 &= \mathbf{b}(0).
 \end{aligned}$$

Hence, a schedule following the repetition vector $\mathbf{q}$, e.g., the one given in (5.2), can be repeated infinitely within finite memory.

Checking for insufficient delays

Insufficient delays can cause the graph execution to deadlock since the firing condition of some actor can never be fulfilled. They are an error in constructing the graph. In [101] it is shown that a large class of scheduling algorithms (termed class S algorithms) will always complete successful if a schedule exists (and if the graph is correctly constructed), and will fail otherwise. Therefore a simple way to check for insufficient delays is trying to find a cycle execution schedule using a class S algorithm.

Scheduling for a single processor

The positive integer vector $\mathbf{q}$ in the right nullspace of Γ determines the number of times each actor has to fire in a valid schedule. A sequential (single-processor) schedule can therefore be constructed by selecting an actor in the graph with all firing rules satisfied, updating the channel buffer vector $\mathbf{b}$, and continuing this process until all actors are scheduled as given by $\mathbf{q}$. Lee and Messerschmitt define class S algorithms as follows:

Definition 5.3. (*Class S Algorithms*): “Given a positive integer vector q such that $\Gamma q = \mathbf{0}$ and an initial state for the buffers $\mathbf{b}(0)$, the i th node is said to be *runnable* at a given time if it has not been run q_i times and running will not cause a buffer size to become negative. A *class S algorithm* (“S” for Sequential) is any algorithm that schedules a node if it is runnable, updates $\mathbf{b}(n)$, and stops (*terminates*) only when no more nodes are runnable. If a class S algorithm terminates before it has scheduled each node the number of times specified in the q vector, then it is said to be *deadlocked*.” [101]

It is proven in [35] that any class S algorithm will complete successfully if a periodic permissible schedule exists. Hence a successful termination of such an algorithm guarantees that there are no loops with insufficient delay in the graph. A simple class S algorithm, as given in [101] is:

1. Find the smallest possible positive integer vector q such that $\Gamma q = \mathbf{0}$.
2. Form a list L of all actors in the graph in arbitrary order.
3. For each $\alpha \in L$, schedule α if it is runnable. Try each α only once.
4. If all $\alpha \in L$ have been scheduled q_α times, stop.
5. If no actor can be scheduled, fail and indicate a deadlock.
6. Else, go to 3.

Discussion

Methods to verify the consistency and deadlock-free construction of an arbitrary SDF graph have been illustrated. These checks are only possible with the SDF MoC, not with the more generic PN model. A simple algorithm to find an efficient static execution schedule of an SDF graph on a single processor has been given. This schedule can be found before executing the graph, avoiding the run-time overhead of a dynamic scheduler or the context switches related to the blocking reads required for a PN. Note that, since the buffer sizes are known at all stages for a specific schedule, the buffers do not have to be implemented using a true FIFO buffer. The memory read and write addresses for each actor firing can be computed at compile-time, avoiding the overhead of FIFO buffer management at run-time. It is also possible to construct near-optimal parallel schedules for multi-processor systems at compile-time, the details of the required algorithm are given in [35].

All these features make SDF graphs very attractive for DSP systems. However, some algorithms cannot be modelled in an SDF graph. Therefore, some extensions to SDF have been

developed, the most prominent of which are scalable synchronous dataflow (SSDF), cyclo-static dataflow (CSDF), boolean dataflow (BDF), and core functional dataflow (CFDF). A brief summary of these MoCs is given in the following.

5.2.3 Scalable Synchronous Dataflow

Scalable synchronous dataflow (SSDF) graphs are a class of SDF graphs with actors supporting multiples of the rates on each input channel per firing. For example, an adder component taking one token on each of its two input channels, adding them, and producing one sum token at the output has all production and consumption rates of one. If the application requires the addition of vectors of data of size n , the actor must be fired n times under the SDF paradigm. With SSDF, it is possible to execute the actor function once, giving it n input samples on each channel at once. This behaviour is functionally equivalent, but has performance advantages on real implementations. The overhead of context switches and function calls can be drastically reduced and the actor function itself is given the opportunity to optimise the code for vector operations (for example using single instruction, multiple data (SIMD) instructions of a vector processor). However, higher vectorisation increases the latency of the application, i.e., the time difference between when a sample enters the graph and when the corresponding output sample is available at the output. This is due to the block-wise processing. Further, more memory is required for the buffers on the channels, which might be a limited resource.

Not all graph structures allow arbitrary vectorisation. When the graph has cycles (feedbacks), the amount of delays in that path presents a limit to the amount of vectorisation possible.

Ritz et al. propose an approach how to create optimum vectorisations for SSDF graphs in [102]. Also methods to construct a schedule for SSDF graphs are proposed in this paper. For brevity, the algorithm is not presented here, the reader is referred to [102] for details.

5.2.4 Cyclo-Static Dataflow

In some common situations, SDF results in inefficient use of memory and computation resources. Consider for example a 4:1 downsampler. Using SDF, the actor consumes four input tokens to produce one output token. This is illustrated in Figure 5.7. That is, four data tokens are passed into the actor but three of which are dropped anyway. It could therefore be avoided to store these three token in the input buffer.

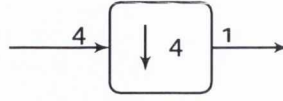


Figure 5.7: A 4:1 downsampler in SDF.

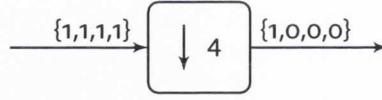


Figure 5.8: A 4:1 downsampler in CSDF.

An alternative is using periodic sampling rates for an actor, as shown in Figure 5.8. In the first firing, the actor consumes one token and produces one token. For the next three firings, a token is consumed but none is produced. Then these rates start from the beginning.

This MoC is called CSDF [103, 106]. It allows actors to specify periodic changes in the token consumption and production rates on the input and output channels. Consistency checks and scheduling is very similar to SDF, adding a little more complexity for the benefit of more efficient schedules.

All CSDF dataflow graphs can be transformed into SDF graphs, by consuming larger blocks of tokens in each firing (as illustrated by the example in figs. 5.7 and 5.8). The SDF version typically requires more memory for the buffers, while CSDF needs a higher complexity scheduling algorithm. A detailed comparison of SDF and CSDF is given in [107].

Obtaining the vector $\mathbf{q}$, describing the number of invocations of each actor in one scheduling cycle, is similar to SDF graphs. Let P_v^u and Q_w^u denote the period of the production and consumption sequences of actors v and w on channel u , respectively. Further, P_v denotes the least common multiple of $\{P_v^u, Q_v^{u'}\}$ among all inputs and outputs of actor v . For the example downsampler actor in Figure 5.8, $P_v = 4$, the least common multiple of the periods of all the channels connected to the actor. The number of tokens produced or consumed by actor v or w on channel u during the first i firings is denoted as $X_v^u(i)$ and $Y_w^u(i)$, respectively. For $i = 0$, this is defined as $X_v^u(0) = Y_w^u(0) = 0$. Now the topology matrix Γ is constructed using the following method:

$$\Gamma_{ij} = \begin{cases} (P_j/P_j^i) \cdot X_j^i(P_j^i) & \text{if actor } j \text{ produces tokens on channel } i, \\ -(P_j/Q_j^i) \cdot Y_j^i(Q_j^i) & \text{if actor } j \text{ consumes tokens on channel } i, \\ 0 & \text{otherwise.} \end{cases} \quad (5.8)$$

To compute the repetition vector $\mathbf{q}$ for a flow graph with topology matrix $\mathbf{\Gamma}$, the positive integer vector $\mathbf{r}$ in the right nullspace of the topology matrix is required, i.e., $\mathbf{r}$ solves the equation

$$\mathbf{\Gamma}\mathbf{r} = \mathbf{0}. \quad (5.9)$$

The repetition vector $\mathbf{q}$, with the elements determining the number of times each actor has to fire in order to return the buffers of all channels in the graph to their original state can then be computed as

$$\mathbf{q} = \mathbf{P}\mathbf{r}, \quad \text{with } P_{ij} = \begin{cases} P_i & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases} \quad (5.10)$$

Checking for consistency of CSDF graphs is similar to SDF graphs. That is, the rank for the topology matrix must be one less than the number of nodes in the graph (i.e., (5.9) must have a one-dimensional solution). A sufficient condition that an CSDF graph is deadlock-free is given in [103]. For brevity, this checking algorithms is not given here, for details the reader is referred to [103].

Bilsen et al. give an algorithm for computing static schedules for CSDF graphs in [103, 106]. It is a tree-based class S algorithm, trying to schedule each node which is runnable in a loop. If the algorithm succeeds to find a periodic schedule for the given graph, it is guaranteed to be deadlock-free and consistent. The CSDF schedule is typically more efficient with memory than the equivalent SDF schedule, at the cost of a higher complexity scheduling algorithm.

5.2.5 Boolean Dataflow

The SDF and CSDF MoCs are adequate to model a large class of dataflow algorithms. However, they are usually not sufficient to model an entire system due to their inability to model data-dependent iteration and conditional execution. The PN MoCs is able to model this but has several disadvantages (see Section 5.2.1).

Buck and Lee introduced a dataflow MoC called *boolean dataflow (BDF)* in [104], which allows conditionals and data-dependent iterations and can still be analysed and scheduled statically. This is achieved by adding two actors, *switch* and *select*. These are illustrated in Figure 5.9.

For example, these actors allow to construct graphs representing an if-then-else construct, as shown in Figure 5.10.

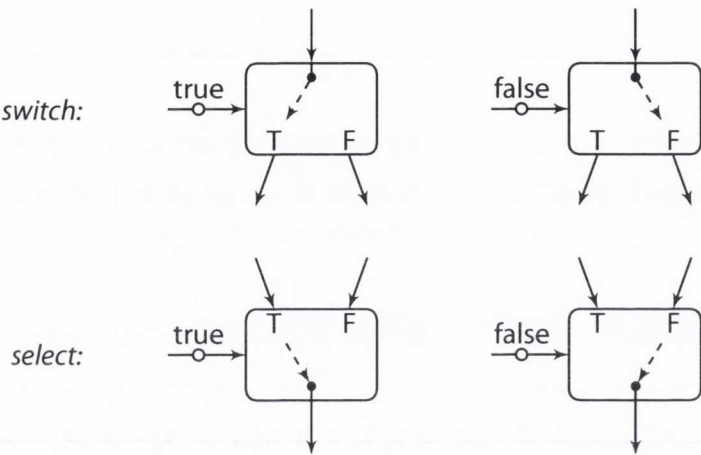


Figure 5.9: The BDF switch and select actors, consuming and producing data depending on the value of boolean inputs.

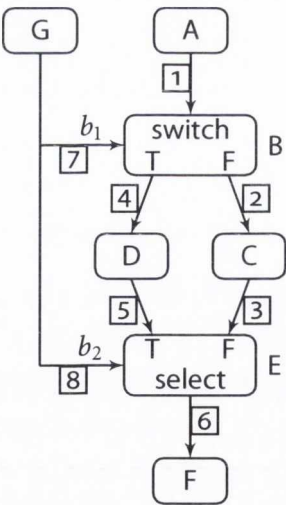


Figure 5.10: If-then-else graph in BDF. Actor G emits boolean-valued tokens (true or false), and selects whether the signal path is through actor C or actor D. The rates on all arcs are one. Arcs are numbered according to the framed numbers in the figure.

Graph consistency can be checked in a similar fashion as for SDF graphs, but the probabilities for true and false tokens on all boolean arcs in the graph are used symbolically in the algorithm. For instance, in the graph in Figure 5.10, the probabilities for the boolean inputs is unknown. It is set to p_1 and p_2 and the consistency checks are computed in the same fashion as for SDF. This yields the following topology matrix Γ ,

$$\Gamma(\mathbf{p}) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & (1-p_1) & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & (p_2-1) & 0 & 0 \\ 0 & p_1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -p_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \quad (5.11)$$

Here the rows correspond to the edges, in order of the framed numbers in Figure 5.10, and the columns correspond to the actors, in alphabetic order.

Similar to SDF, the necessary condition for a consistent graph is that the rank of the topology matrix is one less than full, or equivalently, that its nullspace has rank one. This can be checked by finding a one-dimensional solution to the equation

$$\Gamma(\mathbf{p})\mathbf{q}(\mathbf{p}) = \mathbf{0}. \quad (5.12)$$

In the if-then-else example, $\Gamma(\mathbf{p})$ only has rank 6 if $p_1 = p_2$. From Figure 5.10, it is clear that this condition is always fulfilled (both boolean inputs are fed from the same source).

Not all BDF graphs can be scheduled statically since the probabilities of the boolean values are generally unknown. However, for a large subset of BDF graphs, compile-time schedules can be constructed without knowing the probabilities. Algorithms for finding such schedules are outlined in [104].

The addition of switch and select actors in BDF allows modelling many common signal processing algorithms. Conditional flow can be expressed, including if-then-else, case, and loops. Many of these dataflow graphs can still be scheduled and analysed at compile-time, despite the added flexibility. If this is not the case, a run-time scheduler has to be used, with similar drawbacks as in the PN MoC (see Section 5.2.1).

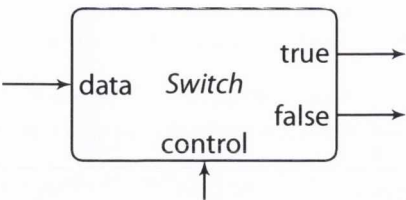


Figure 5.11: The switch actor in CFDF.

Mode	Consumes		Produces		Next Mode
	Control	Data	True	False	
control	1	0	0	0	true/false
true	0	1	1	0	control
false	0	1	0	1	control

Table 5.1: The switch actor in CFDF. When in control mode, the next mode is dependant on the value of the consumed control token.

5.2.6 Core Functional Dataflow

Core functional dataflow (CFDF) is a deterministic dynamic dataflow MoC [105, 108], generalising BDF and SDF. Each actor A has a set of modes M_A in which it can execute. Each mode produces and consumes a fixed number of data tokens on the actor’s ports. The *invoking function* of the actor maps input tokens and the currently enabled mode to output tokens and the next active mode. The *enabling function* indicates which mode may be executed in the current state. Only an enabled mode maybe output by the invoking function.

Figure 5.11 shows a switch actor in CFDF. It has three modes, *control*, *true*, and *false*. When in control mode, the actor only consumes one token from its control input. If the value of the control token is true, the next mode is true, and false otherwise. In true mode, the next input is routed to the first output of the actor, and in false mode, it is routed to the second output. The next mode is then the control mode and the procedure starts again. Table 5.1 shows the modes and the corresponding production/consumption rates. Standard SDF actors can also be modelled with the CFDF MoC, they only contain a single mode.

Graphs applying the CFDF MoC are schedulable at compile-time, using similar methods to BDF graphs. That is, they can be checked for consistency, deadlocks can be detected, and static schedules can be computed before execution. A tree-based scheduling algorithm for CFDF is proposed in [108].

The CFDF is a generalisation of the concepts from the other compile-time schedulable

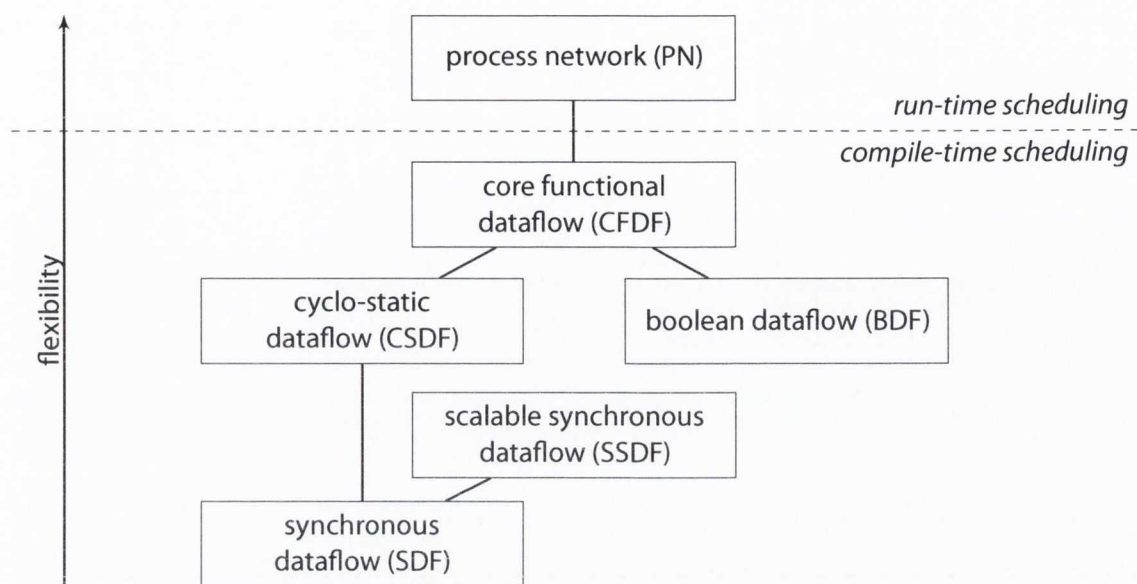


Figure 5.12: Relation between dataflow MoCs. PN is the most generic and flexible model, and SDF is the most specialised and limited model. The MoCs below the dashed line can be scheduled and checked for consistency before execution, while the PN MoC can only be scheduled at run-time. SSDF is a more generic version of SDF (allows multiples of input/output rates), but is not a subclass of CSDF.

MoCs discussed in the previous sections. It provides high flexibility at the cost of more complex actor descriptions and scheduling algorithms.

5.2.7 Relation between Dataflow MoCs

As can be seen from the discussions above, a wide variety of dataflow MoCs exist. They are suited for different purposes, with PN being the most general and SDF the most specialised (and therefore most limited). Figure 5.12 shows all the discussed models in relation.

The figure shows the most generic and flexible model on top, becoming more and more specialised when going down the tree. It also shows the compatibility between MoCs. This means, for example, that all MoCs can alternatively be executed using the PN MoCs, without computing compile-time schedules. Thus, actors developed for a more specialised MoC can also be used in a less specialised MoCs, but not vice versa (provided that the implementation platform provides these capabilities).

The CFDF is the highest level MoC in Figure 5.12 that supports compile-time schedules. It also requires the most complicated scheduling algorithm. When going lower in the figure, the complexity of computing compile-time schedules reduces, with SDF allowing the lowest

complexity algorithms. The SSDF MoC is a slightly generalised version of SDF, but not a specialisation of CSDF (hence its position in Figure 5.12).

The models presented here are the most commonly used models for dataflow applications. There are a number of other models proposed in the literature which typically fit somewhere between the discussed ones in Figure 5.12. They have been omitted here since the discussed ones cover the needs of the dataflow processing for cognitive radios sufficiently.

5.3 Existing Control Flow Models of Computation

For control, typically a different set of MoCs is applied. In contrast to data flow, control is not characterised by processing data streams. Rather, it captures changes in the execution of the data path, switches to different operating modes, and modifications of the system's behaviour.

Not all radio applications require explicit control flow modelling. For instance, in a static linear chain of processing components, such as a simple quadrature phase shift keying (QPSK) receiver, no controller is necessary during execution of the data path. The radio is simply executing its data path forever. Cognitive radio systems, in general, need control. Obtaining information about the environment and the state of the radio (the 'meters'), applying decision-making and learning, and reconfiguring the data path accordingly (the 'knobs') requires control flow.

Several MoCs exist for the control parts of a system. In this section, some models relevant to cognitive radio execution are outlined.

5.3.1 Publish and Subscribe

The publish & subscribe MoC is a well-known model that uses events as the primary mode of communication. It is discussed for example in [84]. One component of a system *publishes* a set of events it might trigger. Other components can *subscribe* to these events. Whenever an event is triggered, it gets transferred to the subscribing components.

This MoC is ideally suited for irregular, untimed communications. That is, the communications occur sporadically in time. The publisher does not need to know who the subscribers are, and subscribers can change. Thus, it is well-suited for an interface to a controller component, which is active only occasionally but does otherwise not interfere with the remainder of the system.

A slightly modified version allows reverse communication along established subscriptions. That is, events are published and subscribed to as before, but subscribers can send messages to the publisher along the established channel. This allows subscribers to send messages to components that do not need to know where the messages are coming from. For instance, reconfigurable parameters of a dataflow actor can be published and subscribers are allowed to adjust the parameter values.

This MoC is suited for the connection of a controller to the data portion of the cognitive radio. Some events can be used for obtaining information about the radio and environment (the ‘meters’) and other events can control the behaviour of the radio (the ‘knobs’). The model allows establishing channels for the exchange of control information, but does not cover the control decision-making itself. It is therefore not suitable for the inner workings of a controller.

5.3.2 Finite State Machines

A finite state machine (FSM) is a deterministic abstract machine that has a set of *states* and moves from state to state in response to external *inputs* [100, chapter 2]. It produces outputs based on a function of the currently active state and the inputs. This makes it ideal to model the control flow of systems which can be in one of many states.

Mathematically, an FSM consists of [100]

- a set of *states* S ,
- a set of *input symbols* Σ ,
- a *state transition function* $\delta : S \times \Sigma \rightarrow S$,
- an initial state $s_0 \in S$,
- and a (possibly empty) set of *final states* $F \in S$.

The state transition function maps input symbols and the current state to a new state, i.e., it defines how the FSM moves between states. Thus, an FSM is described mathematically by a five-tuple

$$A = (S, \Sigma, \delta, s_0, F) . \quad (5.13)$$

This type of FSM is also called *acceptor* since it accepts input symbols but does not produce outputs.

A type of FSM which also produces output is called a *transducer*. It does not have final states F , but defines a set of *output symbols* Γ and an *output function* ω . That is, a transducer

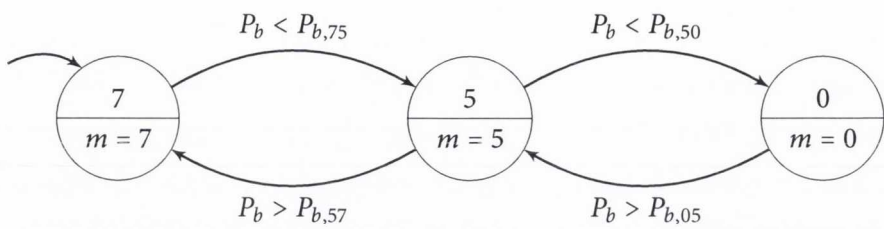


Figure 5.13: The state transition diagram of an example FSM. It models a controller of a receiver where the coding scheme is switched based on the current bit error rate (BER) (P_b), where $P_{b,xy}$ is a predefined BER threshold for switching from state x to state y . The upper part in the circles denotes the state number, the lower part is the output, i.e., the mode of the receiver.

FSM is a six-tuple

$$T = (S, \Sigma, \Gamma, \delta, \omega, s_0) . \tag{5.14}$$

There are two types of transducer FSMs: *Mealy* and *Moore*. In *Mealy FSMs*, the output symbol depends on the current state and input symbol, while in *Moore FSMs*, the output depends on the current state only. Thus, the output function is defined as the following mapping:

$$\omega : S \times \Sigma \rightarrow \Gamma \tag{Mealy FSM} \tag{5.15}$$

$$\omega : S \rightarrow \Gamma \tag{Moore FSM} \tag{5.16}$$

FSMs can be illustrated graphically by a state transition diagram, an example of which is illustrated in Figure 5.13. It represents a receiver supporting three different coding schemes: no coding or convolutional codes with constraint length 5 or 7. The circles represent the states, the arrows and the conditions on them represent the transition function. The input alphabet consists of the variables appearing in the conditions, i.e., the BER P_b , and the initial state is shown by a disconnected arrow into it. Final states are signified in state transition diagrams by a double circle; this example does not have a final state. In Moore machines, where the outputs depend on the state only, the output of the FSM is shown in the lower part of the state circles. In this example, the output represents the operating mode of the receiver, i.e., the coding scheme used. For Mealy machines, where the output is dependent on state and inputs, the output is signified on the transition arrows behind the conditions, usually separated by a slash ‘/’.

Often specific actions are required when an FSM operates, for instance, in the example from Figure 5.13, the receiver needs to be configured to the corresponding coding scheme when a state is entered. Two types of actions are defined: *entry actions* are affected when a state is entered and *exit actions* are affected when a state is left.

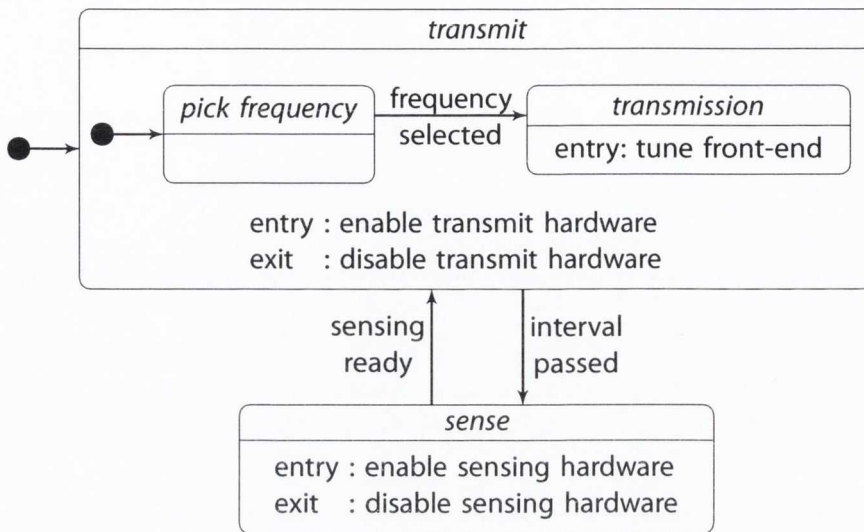


Figure 5.14: UML statechart example. It represents a transmitter that regularly performs sensing. When transmitting, first a frequency is picked from the sensing information, then normal transmission occurs. Entry and exit actions are signified in the states.

The UML standard [92], used for graphical modelling of software, defines an extension of FSM termed *UML statechart*. This includes hierarchically nested states, orthogonal regions (state machines operating in asynchronously in parallel), and extends the notion of actions (e.g., actions can be associated with specific inputs). Figure 5.14 shows an example demonstrating some of the concepts. For details about UML statecharts, the reader is referred to [92].

UML statecharts overcome some limitations of FSMs. For instance, allowing nested states often reduces the number of required states compared to an FSM equivalent. It also keeps the chart cleaner and is therefore easier to understand. With traditional FSMs, only one state can be active at a time, while the orthogonal regions defined in UML statecharts allow parts of the machine to operate in parallel. There are more differences which make UML statecharts more attractive for some designs, see [92] for details.

FSMs (and UML statecharts) are widely used for controlling the behaviour of systems that possess a finite number of operating modes (or states). This model is effective if the transition functions are relatively simple, i.e., they typically do not involve complicated algorithms to determine the next state. Figure 5.13 is an example to model a radio controller for an adaptive coding receiver. It can be used to model the controller for the experiment discussed in Section 3.3 on page 79.

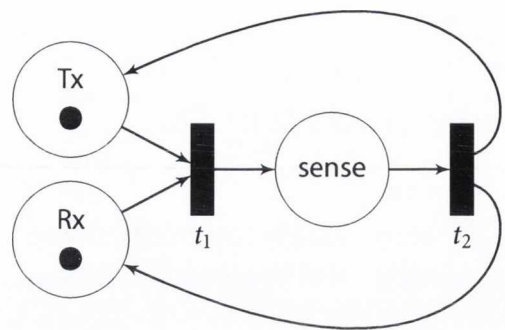


Figure 5.15: An example Petri net for a sensing transceiver. The transceiver is either concurrently transmitting and receiving, or it performs spectrum sensing only.

5.3.3 Petri Nets

Petri nets [109] offer a graphical notation for stepwise processes that include choice, iteration, and concurrent execution. They have an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis. Petri nets are affective to model systems in many applications areas, e.g., protocols and networks, software design, business processes, hardware, or control of systems. They are a useful model for controllers of cognitive radios, especially when concurrency is involved.

A Petri net is a directed bipartite graph, i.e., a directed graph with two alternating types of nodes, one representing transitions (events that might occur, signified by bars) and the other representing places (signified by circles). The arcs describe which places are pre- and/or post-conditions for the transitions. Places on an incident arc represent pre-conditions (called *input places*) and places on an outgoing arc represent post-conditions (called *output places*). Markers represent the current state of the Petri net (signified by dots). Multiple markers are allowed per Petri net and place.

The execution semantics (i.e., the MoC) of a Petri net are as follows. A transition is enabled if all input places contain at least one marker (the pre-conditions are fulfilled). When it fires, it consumes one marker from each input place and produces one marker on each output place. Executions are non-deterministic, i.e., if multiple transitions are enabled at the same time, any one of them may fire. Enabled transitions may fire, but they do not have to. This non-determinism makes Petri nets well-suited for modelling concurrent behaviour.

An example Petri net is shown in Figure 5.15. It represents a controller of a transceiver system that supports spectrum sensing. Normal transmission and reception can occur concurrently, but the transmitter must be silent during sensing. The system starts with both transmission

and reception active. It can then go into sensing (the input places of t_1 each contain a marker), disabling the transmission and reception. When ready, transmission and reception can resume (the input place of t_2 contains a marker) and the system is back in its initial state.

Petri nets can be described formally using the following notation. A Petri net is a 4-tuple (S, T, W, M_0) [110], where S is a finite set of places, T is a finite set of transitions, and $W : (S \times T) \cup (T \times S) \rightarrow \mathbb{N}$ is a multiset of arcs. This defines arcs and assigns an integer multiplicity to each one, i.e., multiple parallel arcs are permitted. A multiset M defines the marking of the graph, i.e., a mapping of $M : S \rightarrow \mathbb{N}$, with M_0 denoting the initial state.

The execution semantics of a Petri net can be defined mathematically based on this formal description. For brevity, these are not given here. The reader is referred to [110, 111].

Petri nets are a superset of FSMs, i.e., every FSM can be modelled as a Petri net. A Petri net with exactly one initial marker and where each transition has only one incoming and one outgoing arc is an FSM. Thus, a Petri net can be viewed as an extension to FSMs which allows concurrency and non-determinism.

Their ability to model concurrent systems with synchronisation between processes makes Petri nets a suitable model for cognitive radio controllers. A simple example is shown in Figure 5.15, although much more complex controllers are possible.

5.4 Summary

This chapter introduced model-based design as a technique to design complex systems. With this technique, requirements are mapped into a model of the system to be designed, rather than directly to a hardware/software implementation. The model captures the behaviour of the system so that most of the information required for implementation is included. At the same time, the model does not include low-level implementation details and is therefore platform-independent in nature. This forms an attractive approach to develop cognitive radios.

Relevant existing models for data processing and control flow have been reviewed in this chapter. Although powerful, none of the discussed models captures the dynamic behaviour required for cognitive radios, i.e., the run-time adaptation of the radio operation to the current environmental and internal radio conditions. The discussed data flow models can model one state of the radio at a given time, but not the changes in its operation. The control flow models can be used to model the connection between a cognitive engine and the signal processing portion of a cognitive radio and for simple decision-making processes, but they cannot model

the changes in the radio's behaviour.

If model-based design is to be used for cognitive radios, a new model needs to be developed that combines some features of the existing models presented in this chapter and adds the capability to model the dynamic behaviour of a radio. This is attempted in the following chapter.

6 A Model-based Approach to Cognitive Radio Design

Contents

6.1 A Model for Cognitive Radios	144
6.2 Model-based Cognitive Radio Design	161
6.3 Implications for Cognitive Radio Frameworks on the Example of Iris-FPGA	172
6.4 Summary	178

This chapter proposes a new model for cognitive radios which serves as a basis for model-based cognitive radio design. The model captures the composition of cognitive radios from functional processing components, their reconfigurable nature, and the interface to the cognitive engine. It allows designs to be understood and verified without first being implemented. This enables radio researchers to focus on applications of interest, designing, iterating, and evaluating real systems, without the need to focus on implementation details. The model reduces design effort, provides higher reliability, and allows easy deployment on different target platforms.

The model proposed in this chapter is based on the dataflow and control flow models reviewed in Chapter 5. It combines features of the discussed models and adds new features, resulting in a model that fully describes a cognitive radio on a high level. It is developed in Section 6.1.

The proposed model can be used as a description of cognitive radios and their behaviour. For instance, researchers can use the model to communicate ideas or describe radio behaviour without implementing it. However, the main aim is to use the proposed model for model-based design of cognitive radio systems. Section 6.2 proposes methods to generate real cognitive

radio implementations from a given model, simplifying the design process significantly.

Although the proposed model can be used to design for a wide range of different target platforms as it is, it has another advantage. It encourages a new way of thinking about the architecture of cognitive radios. This may have an effect on the way cognitive radio implementation frameworks and platforms are implemented, taking ideas from the model into account. As an example, the Iris-FPGA cognitive radio framework, introduced in Chapter 2, was improved in this way. Section 6.3 outlines how the proposed model affected the new generation of this framework.

The model and design approach outlined in this chapter are to be published in [112].

6.1 A Model for Cognitive Radios

It is hugely beneficial for researchers and developers of cognitive radios to be able to design such systems at a high level of abstraction, avoiding detailed platform knowledge where possible. This reduces the design effort, increases reliability, and allows easy deployment on different target platforms. It also prevents the limitations of a single platform from working their way into the radio description, resulting in constraints that are of no relation to the application being described. Developers instead focus on the radio's function and cognitive decision-making and need not concern themselves with platform details.

It follows from the discussions in Section 4.2 and Section 5.4 that employing a model-based design technique for cognitive radios based on dataflow graphs is hugely beneficial. They efficiently model the signal processing part of a cognitive radio. However, dataflow graphs do not capture the dynamic nature of cognitive radios; they can only represent a single configuration. Incorporating other configurations that might arise as a result of the cognitive decision-making cannot be modelled using traditional dataflow graphs.

A model for describing cognitive radio designs is proposed that harnesses the strengths of existing dataflow-based methods while overcoming the current limitations of existing design frameworks. It describes the radio's composition, all reconfiguration options, and defines an interface to the cognitive decision-making process without relying on any platform implementation details. Researchers can use the model not only as a basis for developing real cognitive radio implementations, but also as a description of cognitive radios. This allows other researchers to understand and analyse a radio design, or repeat experiments themselves.

This section proposes a model for cognitive radios with a hierarchical data plane consisting

of dataflow actors termed *components* and *subsystems* and links connecting them. It includes a model for the interface to the control plane, but the details of the control plane are left open for the added flexibility to support several cognitive engine types. The following sections give details of the proposed model and its properties.

6.1.1 Data Plane

Dataflow graphs are an established approach for modelling signal processing systems. Therefore it was decided to base the proposed model on dataflow graphs and to add the capability to capture the dynamic nature of cognitive radios. The proposed solution is to add hierarchy to dataflow models, i.e., actors in the dataflow graphs that can represent other dataflow graphs internally. These actors, termed *subsystems*, contain multiple options for the encapsulated graph, each representing a different configuration. For example, the subcarrier modulation actor in an orthogonal frequency division multiplex (OFDM) system might contain multiple modulators ranging from binary phase shift keying (BPSK) to 64-quadrature amplitude modulation (QAM), only one of which is active at a time. The following paragraphs describe all elements of the data plane in detail.

Each level of hierarchy is a dataflow graph, i.e., a directed graph of actors. In this model, actors can either be processing *components* or *subsystems* and the channels representing the flow of data are termed *links*.

In the proposed model, *components* are the atomic entities for processing data. They consume data from their input *ports*, perform processing, and produce data on their output *ports*. Example components are a quadrature phase shift keying (QPSK) modulator, which maps information bits to complex signal samples, or a pulse-shaping filter, used to limit the transmission bandwidth of a radio. To allow easier reuse of existing components and also as an enabler for parametric reconfiguration, a component may have a (possibly empty) set of *parameters*. For example, a pulse-shaping filter might have parameters for selecting the number of taps and the roll-off factor. For the connection of components to a dataflow graph, they define a set of zero or more input and output ports. For instance, a select component has two input ports and one output port, routing one of the inputs to the output port depending on a select parameter. Components without inputs are termed *sources*, components without outputs are termed *sinks*. Since components are intended as a part of a dataflow graph, they must define at least one port for connecting them. This model does not define the internal representation of components; they are treated as black boxes. Only the system composition is considered.

A dataflow actor in the proposed model may also be a *subsystem*. This represents a new level of hierarchy. It can include multiple mutually-exclusive dataflow graphs, only one of which is active at any given time. Each of these graphs represents a configuration option, termed a *mode* in the subsystem. For example, a subsystem might be chosen to represent different options for the forward error correction (FEC) in a transmitter, each represented by a mode. The FEC scheme can then be selected by activating the corresponding mode. When viewed from the next higher level in the hierarchy, the subsystem is, at any point in time, represented by the dataflow graph for the active mode. The graphs for each mode in a subsystem may have external input and output ports. To ensure that each of the graphs in a subsystem (i.e., each mode) is compatible with the connections of the subsystem in the higher level graph, all modes within one subsystem must have identical numbers of input and output ports. The dataflow graphs contained in a subsystem can themselves contain further subsystems, i.e., arbitrary levels of hierarchy are possible. The lowest level only contains components and no more subsystems, i.e., it cannot be broken down further. The key benefit of this proposed hierarchical extension to traditional dataflow graphs is that it can capture structural reconfiguration. A change of the active mode in a subsystem represents a structural change of the active flow graph, i.e., it is the equivalent of replacing a part of the graph with a different one.

To allow adjusting the behaviour of components, they can expose *parameters*. These define data types, a specification of allowed values (which might be an interval or a list), and a value. To enable parametric reconfiguration, it is proposed to allow parameters to be run-time reconfigurable. For example, a scaling component might expose the run-time reconfigurable parameter ‘gain’ of type ‘real’, with allowed values in the interval $[0, \infty)$, to allow tuning the scaling factor at run-time.

Component or subsystem *ports* can be either input or output ports. A data type is associated with each port, and defines the type of the data items consumed or produced. When active, components consume data from their input ports and produce data on their output ports, for example by mapping information bits to signal samples in a modulator component.

A *link* is a connection of output ports to input ports of components or subsystems. It resembles a first in, first out (FIFO) buffer, i.e., it can store multiple data items. Data items produced on output ports are queued at the end of the buffer and items consumed from input ports are taken from the beginning. For instance, a convolutional coder component passes information bits to a modulator component via a link.

The top section of Figure 6.1 shows an illustration of the data plane of the proposed model for cognitive radios. The figure shows two hierarchy levels and uses a subsystem with three

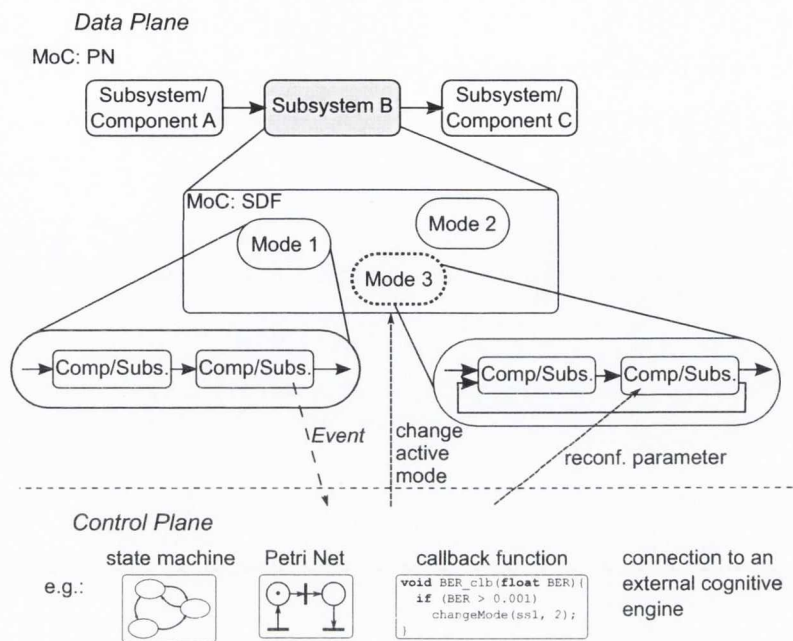


Figure 6.1: A model for cognitive radios. The data plane shows two levels of hierarchy, using a subsystem with three modes. Two different MoCs are applied: process network (PN) and synchronous dataflow (SDF).

modes.

It is important to note that in a practical system, components can continuously be broken down into subsystems containing further dataflow graphs, until components reach the level of logic gates and cannot be broken down further. The desired granularity and interest dictates when this breakdown into subsystems is stopped in a design. For instance, a network node designer might only be interested to model the physical layer as a graph of three components: a radio front-end, a transmitter, and a receiver. Further details are not of interest to him, therefore he decides not to break down the components into further subsystems. Physical layer designers might break down the transmitter component into a subsystem with dataflow graphs containing components down to the granularity level of modulators, filters, or encoders. Implementation engineers might then break down the filter component further, into a graph of multipliers, delays, and adders. The model for the data plane proposed here functions on all levels of granularity. It is up to the radio designer how much detail he wants to include in the model.

Having defined the structure of the data plane, it is essential to understand how this is executed on a real platform, i.e. the MoC has to be defined. It should be independent of implementation details, so that the proposed model can be used to describe radios on a high

level. This information allows radio designers to successfully construct a model of a radio and understand its execution behaviour.

The MoC for the proposed model is defined as follows. Each dataflow graph at any level in the model hierarchy can be treated as a traditional dataflow graph without hierarchy by viewing the subsystems and components it contains as dataflow actors. This allows to apply traditional dataflow MoC, as introduced in Section 5.2 on page 118, at each hierarchy level individually. Thus, each level of hierarchy in the proposed model must define the traditional MoC applied. Further, every mode within a subsystem must apply the same MoC, to allow a consistent execution for all subsystem configurations. It is possible to combine different MoCs at different hierarchy levels, provided that they are compatible according to the discussions in Section 5.2.7 on page 135.

For instance, a flexible receiver radio containing a bandwidth-adaptive OFDM demodulator [89] requires the application of the PN MoC at the top level due to unpredictable input and output data rates of that component. However, decoding and deframing the data are synchronous operations, i.e., fixed data consumption and production rates can be given on the component ports. Therefore, the decoding and deframing can be placed into a subsystem which is internally executed using the SDF MoC. Since SDF is a specialisation of PN, an SDF subsystem can be executed within a PN context. With this configuration, the decoding and deframing can be executed efficiently with static schedules and high memory efficiency, while the remainder of the receiver applies the more flexible but less efficient PN MoC.

Allowing a combination of different MoCs on different levels of hierarchy is a powerful feature of the proposed model. Radio designers can create radio models optimised for performance which still provide high flexibility where needed. They apply the most efficient MoC for every part of the radio flow graph and combine them in a hierarchy. Thus, two major features required for cognitive radio systems are combined: flexibility and performance.

6.1.2 Reconfiguration

Reconfigurability is key in cognitive radios, and, besides the cognitive engine, the key difference to legacy radios. It allows them to adapt to the current conditions, both internal and environmental. Reconfigurability is therefore the key driver for the proposed model of the data plane, the reason why it provides the concept of subsystems with several modes, i.e., options for reconfiguration. Reconfigurations of cognitive radios can be classified into three types: *parametric*, *structural*, and *functional*. The following paragraphs explain these types and detail

how they are supported in the proposed cognitive radio model.

A *parametric* reconfiguration is a change of a component parameter in one or more of the components in the data plane during the radio's execution. For instance, the gain of a scaling component can be adjusted to compensate for lower signal levels at its input. This is enabled by allowing component parameters to be dynamically adjustable in the proposed model. Not all parameters support changes at run-time; those which do are marked as 'dynamic' in the model.

A *structural* reconfiguration is a change of the radio's structure, i.e., of the currently executing dataflow graph. For instance, replacing a QPSK modulator component with a 16-QAM modulator or removing an encoder component from the graph. In the proposed model, structural reconfiguration can be achieved by switching the active mode within a subsystem. The structure of currently executing dataflow graph, characterised by all active modes in the proposed hierarchical model, is modified by this switch, i.e., a structural reconfiguration is performed.

A *functional* reconfiguration involves a complete overhaul of the radio's function, e.g., switching from an IEEE 802.11 [6] system to UMTS [7]. In the proposed model, this can be treated as a special case of structural reconfiguration, where a single subsystem at the top level contains modes for each function. Therefore no special considerations have to be given to this type of reconfiguration.

Thus, the proposed hierarchical structure of the radio data plane along with component parameters enables all types of reconfiguration.

Note that through the use of hierarchy in the data plane, all types of reconfiguration can be converted to parametric reconfigurations. For instance, if a structural reconfiguration is a switch of the active mode in a subsystem, this subsystem could be converted to a component which exposes a parameter for selecting the mode. This can be continued for functional reconfiguration as well, as this is a special case of structural reconfiguration. Thus, at different levels of hierarchy, the same reconfiguration can be of a different type. As discussed in the Section 6.1.1, it is up to the designer how much detail, i.e., how many levels of hierarchy, he wants to include in the model.

6.1.3 Control Plane

The control plane of the proposed radio model monitors the execution of the data plane, decides about reconfiguration actions, and triggers these in the data plane. It executes the *cognition*

cycle of a cognitive radio, which involves three stages: *observe*, *decide*, and *act*, as defined in the introduction. During the *observe* stage, the control plane monitors the state of the radio and can retrieve information from the data plane (the ‘meters’). The *decide* stage performs decision-making using the observed information, and optionally includes machine learning techniques. During the *act* stage, the chosen reconfiguration actions are applied to the data plane (the ‘knobs’). In the literature, the part of a cognitive radio which executes the cognition cycle is often called *cognitive engine*.

To allow reconfigurations at any time during execution, the control plane executes concurrently with the data plane. It may be split into multiple parts, each responsible for a different section of the data plane, so that orthogonal or concurrent parts can be controlled independently.

The control plane typically takes a passive role during normal radio execution, and only comes into action when the radio’s operating conditions or the environment changes. It is therefore beneficial to have a loose link between the control plane and data plane, where data travels untimed and, compared to within the data plane, sparsely. Further, a tight coupling between components in the data plane and the control plane would create unnecessary dependencies that might prevent components to be used in other contexts. Therefore the publish & subscribe MoC was chosen to model this link, as it fulfils these requirements.

That is, components in the data plane publish *events* and elements in the control plane may subscribe to them. When the event is triggered, all subscribers are notified, giving the control plane a chance to react. To obtain information about specific operating conditions, events may carry information data, e.g., the current bit error rate (BER), or a map of available spectrum. This represents the first part of the ‘meters’ of a cognitive radio, used during the *observe* stage of the cognition cycle. The second part is general information about the state of the data plane, doubtless useful for a cognitive engine to base decisions on. In the proposed model it is therefore assumed that the control plane always has an exact picture of the data plane. In a practical framework, this can be achieved, for instance, by maintaining a representation of the current configuration of the data plane and allowing the controller to query for this through an application programming interface (API).

The *decide* stage in the cognitive cycle is less straightforward. A huge variety of approaches and methods for decision-making and machine learning exists in the literature. For example, artificial neural networks [113, 114] can make a decision and perform learning, heuristics can find near-optimum configurations given an objective, genetic algorithms [115–117] can find good solutions for multi-objective functions, hidden Markov models [118] can predict behaviour and

can be adapted for learning, to name just a few. A good survey of various methods can be found in [119]. Apart from the methods applying artificial intelligence and complex optimisations, there are also simpler ways for decision-making. For example, a look-up table can be used with predefined settings in certain scenarios, or an finite state machine (FSM) (see Section 5.3.2) or Petri net (see Section 5.3.3) can be applied in the control plane.

Due to this wide range of possibilities for the realisation of the control plane, it would be too restrictive to define a fixed model for the decision-making part. Thus, the model for this part is unspecified, allowing radio designers a high flexibility. For example, one radio designer might choose an FSM to implement a controller which switches the applied coding scheme based on the current BER. Another radio designer might choose a genetic algorithm optimising the throughput, bandwidth efficiency, and energy efficiency of the system, to find a suitable radio configuration. A generic and flexible solution to allow different implementations of the controller is callback functions in software code, invoked when certain events are triggered from the data plane (through the publish & subscribe MoC used for the interface). These can be used to perform any decision-making algorithm or to interface with an external cognitive engine entity.

The control plane applies the chosen actions to the data plane in the *act* stage. These can be changes in parameter values of components or switches of the active mode in a subsystem, i.e., parametric, structural, or functional reconfigurations. This is achieved through an extended publish & subscribe MoC (see Section 5.3.1). The data plane publishes the reconfigurable component parameters and active modes, and the control plane subscribes to them to establish a message channel. New configuration values are then affected by sending messages through these channels. The publish & subscribe MoC is a good choice due to its suitability to untimed and sporadic message exchange which is characteristic for interfacing the data and control planes.

6.1.4 Model Correctness

When a given cognitive radio is modelled using the proposed model, a high probability for consistency and dataflow errors persists. For example, it is a common error to connect component with inconsistent data types, e.g., a modulator might expect bits at its input port but the file reader component placed before outputs bytes. Another common error is parameter values set out of range, or with the wrong type, e.g., setting the number of coefficients to a negative number in a filter component.

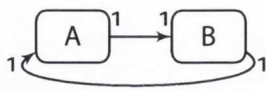


Figure 6.2: An SDF graph with deadlock. Component A can never execute due to insufficient data on its input.

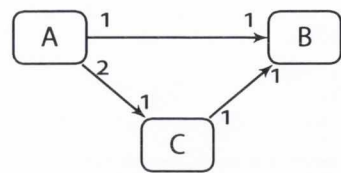


Figure 6.3: An inconsistent SDF graph. The buffer on the link between A and C will grow infinitely.

In case the designed model applies compile-time schedulable MoCs anywhere, more thorough checks can be performed on the model. Deadlocks can be detected or graph consistency problems can be identified (see Figure 6.3). From experience from various experiments within the research group it became apparent that these mistakes are made frequently, and often go undetected until radio execution. Figure 6.2 illustrates an example for an SDF graph that will deadlock, and Figure 6.3 exemplifies a graph consistency problem where the buffer size on one of the links will grow infinitely. These errors in the proposed model can be detected in the design stage, before the radio implementation on a target platform. This saves the developer valuable time and effort, as he can correct the errors early in the design. The correctness checks ensure that the radio is indeed executable, once implemented on a target platform.

The following paragraphs propose model correctness checks that can be applied to the cognitive radio model introduced earlier.

Component information library

Model correctness checks require some additional information about the components in a radio model. The names of components, the parameters, their types and allowed values, published events, supported types on component ports need to be known for many of the checks. Further, when compile-time schedulable MoCs are applied, the corresponding component information is required. For instance, checking for deadlocks and consistent flow graph construction requires the sample rates on each port of an SDF component and whether some parameter affects these (the upsampling factor parameter of an upsampler SDF component has an effect on the sampling rates).

There are multiple ways to provide this information for model checking. First, if the target platform is already decided, this information can be obtained from the component library the target implementation platform provides. Second, the model correctness checks might be performed by automated tools which provide a component information library themselves.

Third, the radio designer provides this information himself. Note that with the second and third option the model correctness checks are still strictly platform-independent, while the first option uses some information from the target platform. Still, only meta-information is used, not the implementation itself. Therefore, model correctness checking can be considered platform-independent.

If the component information is provided by the model-checking tools or the developer, it can be used as a specification for component implementation later in the development process. The information defines the parameters, events, and ports in detail, i.e., it is a full specification of the component's interface to the remainder of the system. This information can help during component implementation if no implementation exists already for the target platform.

Control specification

For more thorough model correctness checks, it is valuable to add a control specification to the radio model. This holds information about the parameters to be reconfigured and the events subscribed to by the control plane. With this information, model correctness checks can ensure that the intended reconfigurations and subscriptions to component events are possible and allowed. Additionally, this information has proven valuable to aid automatic synthesis tools for generating radio implementations automatically.

The control specification lists all parameters that may be reconfigured at run-time along with their possible values. Further, it lists the events the control plane subscribes to, along with the components publishing them. This information can be used during model correctness checking and to aid automatic synthesis tools.

General model correctness checks

The following criteria can be checked for in any cognitive radio model as proposed in this chapter, i.e.,

- component parameter consistency,
- dynamically reconfigurable parameters,
- link data type consistency, and
- compatibility of MoCs in the hierarchy (see Section 5.2.7).

The following paragraphs give details about these checking criteria.

Component parameter consistency — This test determines whether all parameters given and configured in the radio model exist with the components of the component library and whether their values are within range. This can be achieved by comparing the parameters in the model with the component information library.

Dynamically reconfigurable parameters — The control specification lists all parameters that might be reconfigured at run-time by the control plane. This check ensures that these parameters are actually run-time reconfigurable (some parameters are static and do not support changes during execution).

Link data type consistency — The links between components and subsystems in the proposed radio model are associated with a data type. This type can be determined for all links, by checking the port data types for each component from the component information library. For instance, consider a link between a modulator and filter component. When the modulator outputs complex floating point values and the filter supports this data type at its input, the check is successful. Further, within a subsystem, the graphs in all modes must have the same external input and output link data types, so that they can be placed seamlessly into the containing dataflow graph in the next higher level of hierarchy. This correctness check confirms that all link data types are consistent and supported within the full hierarchical data plane of the proposed model.

Compatibility of MoCs — Different dataflow MoCs can be applied in different subsystems throughout the radio model. However, not all combinations are allowed. For instance, a PN subsystem cannot be placed within an SDF graph, while the opposite is possible. The compatibility of MoCs is explained in Section 5.2.7 on page 135. This check confirms that the MoCs applied in the given model are compatible to each other.

As discussed in Section 5.2.1 on page 119, it is generally not possible to determine whether a dataflow graph employing the PN MoC is executable on a real system. Its execution has to be tested on a real implementation and cannot be analysed during design. Therefore, if all hierarchy levels of the given model use the PN MoC, no more correctness checks can be performed at design-time. The radio designer must synthesise the model onto a real platform and execute the radio in order to determine whether it can be executed in bounded memory and without deadlocks. On the other hand, a more thorough analysis of the proposed model is possible if some parts apply compile-time schedulable MoCs, e.g., SDF.

Criteria for models using compile-time schedulable MoCs

All dataflow MoCs discussed in Section 5.2 apart from PN allow consistency checks and the construction of static schedules without executing the radio. That is, guarantees for the existence of a cyclic execution schedule, the absence of deadlocks, and the boundedness of the buffer memory on the links can be given. This feature can be used to develop further correctness checks for the proposed radio model if some portion of it employs one of the core functional dataflow (CFDF), boolean dataflow (BDF), cyclo-static dataflow (CSDF), scalable synchronous dataflow (SSDF), or SDF MoCs. In order to do this, the following notation is introduced.

Let S_n denote the finite set of all modes in subsystem n of a given model. Further, P_m denotes the (possible infinite) set of desired reconfiguration values for parameter m . Then, at any given time instant, the global state of the data plane can be computed by the Cartesian product of all S_n and P_m in the model, i.e.,

$$G = \prod_{n=0}^{N-1} S_n \times \prod_{m=0}^{M-1} P_m, \quad (6.1)$$

where N denotes the total number of subsystems in the model, M is the total number of reconfigured parameters, and $\prod$ is the Cartesian product of the arguments. The resulting set G is a set of tuples, each representing a global state of the data plane. Note that, since some of the sets P_m might be infinite, G might also be an infinite set.

For each of the states in G , the hierarchy can be flattened to a single (non-hierarchical) dataflow graph with fixed parameters. Therefore well-known graph correctness checks can be applied for the portions of the data plane that use a compile-time schedulable MoC (see Figure 5.12). By checking the relevant portions of the flattened graph in each of the states, it can be guaranteed that the model is correctly constructed.

However, the number of states $|G|$ can become very large due to the Cartesian product in (6.1) which results in a potentially very complex model checking procedure. If $|G|$ is infinite, it is impossible to check all possible radio configurations. However, the number of states to be checked can often be reduced drastically and can be made finite for all practical cases. The following state space reduction rules can be applied (examples follow):

1. Only subsystems employing a compile-time schedulable MoC need to be considered (CFDF, BDF, CSDF, SSDF, or SDF), subsystems applying the PN MoC can be eliminated.
2. All reconfigured parameters that do not affect the input/output sampling rates of a

component can be eliminated.

3. On multiple hierarchy levels, some subsystems can never be active at the same time; these combinations can be eliminated.
4. In each compile-time schedulable subsystem, modes with the same external input/output sampling rates can be reduced to one mode.
5. In each compile-time schedulable subsystem, where a common input/output rate can be found for multiple modes by multiplying their individual rates by integers, these modes can be combined by computing the lowest common multiple of the rates and repeating the mode schedules accordingly.
6. If some modes or parameter settings are mutually exclusive, as may be given in constraints by the radio designer, the number of combinations can be reduced.

As an example for rule 1, one could design a radio model applying the PN MoC at the top level, to allow the inclusion of a bandwidth-adaptive OFDM component (with flexible output data rates). However, the decoding and deframing subsystem could be executed with the SDF MoC, since all components and modes have fixed input and output data rates. For this model correction check, only the parameters and subsystems within the decoding and deframing subsystem are relevant, since the PN MoC does not support checking for deadlocks and boundedness of memory. Thus, the parameters and subsystems in the PN part of the model can be eliminated from the state space G .

For rule 2, consider a scaling component within a radio model. It multiplies its input signal with a scaling factor, that can be reconfigured by the control plane. This factor does not affect the input and output sampling rates of the component, they are both 1 (for one input item, one output item is produced). Thus, the corresponding parameter set P_s can be eliminated from G (s is the index of the scaling factor parameter). This is also an example for a parameter with infinite configuration possibilities if arbitrary scaling factors are allowed. After applying rule 2, the set of possible radio model states becomes finite.

In radios targeted for a practical implementation, it is useless to have a parameter with infinite configuration possibilities that affects the input and output sampling rates. This would mean that the radio can execute with an infinite number of data rates, something that is clearly not practical on any real platform. Consider for example an upsampler component. Allowing arbitrary values for the upsampling factor, a parameter which affects the output sample rate of the component, would increase the amount of data to be stored and processed to infinity. This cannot be realised on a real platform. Therefore, for all practical purposes, parameters with infinite reconfiguration options do not affect the input and output sampling rate of the

corresponding component. Thus, the set of states for the radio model can be made finite by applying the state reduction rule 2.

If the radio model has more than one level or hierarchy, some combinations of modes are irrelevant and do not need to be checked for correctness (rule 3). Consider, for instance, a radio that has one subsystem at the top level with two modes, one for sensing and one for reception, and within sensing it has another subsystem with two modes, one for energy detection and one for cyclostationary feature detection. When the reception mode is active, it is not relevant which is the active mode in the subsystem within sensing. Thus, the original state space

$$\begin{aligned}
 G &= S_0 \times S_1 \\
 &= \{(\text{reception}, \text{energy_detection}), \\
 &\quad (\text{reception}, \text{cyclostationary}), \\
 &\quad (\text{sensing}, \text{energy_detection}), \\
 &\quad (\text{sensing}, \text{cyclostationary})\}
 \end{aligned} \tag{6.2}$$

can be reduced to

$$\begin{aligned}
 \check{G} &= \{(\text{reception}, \times), \\
 &\quad (\text{sensing}, \text{energy_detection}), \\
 &\quad (\text{sensing}, \text{cyclostationary})\},
 \end{aligned} \tag{6.3}$$

where $\times$ signifies an irrelevant tuple member.

An example for rule 4 is an encoder subsystem, containing multiple modes for convolutional codes with code rate $1/2$, applying code polynomials with different constraint lengths. A code rate of $1/2$ means that the encoders produce 2 bits for every 1 bit they consume, no matter which constraint length codes have been applied. Since the correctness checks for compile-time schedulable MoCs (deadlocks and graph consistency checks) are only concerned the data rates on all ports of a component and in this example those are identical, the modes can be combined into one.

Rule 5 requires a more complicated example for illustration. Consider an SDF subsystem with two modes, each holding a different subgraph, for example to implement two different synchronisation methods in a receiver. Scheduling each of these subgraphs individually results in an input and output rate for the first mode of 6 and 2, respectively, for the first mode, and

rates of 9 and 3, respectively, for the second mode. Now, a lowest common multiple for both the input and output rate can be found for both modes: an input rate of 18 and an output rate of 6. Thus, when repeating the schedule for the first mode three times and the schedule for the second mode two times, the data rates for both modes are identical. For model checking, they can therefore be combined into one mode with the input and output rates of 18 and 6, respectively, reducing the global state space for the model.

An example for the last rule (rule 6) is a radio where the transmission bandwidth of the radio front-end and the number of subcarriers in an OFDM component can both be reconfigured. The radio designer can give constraints about these reconfigurations, stating that some combinations of the two parameters are invalid and will never be applied. In the example, when the bandwidth is doubled, the number of subcarriers is doubled as well, to keep the subcarrier spacing constant. This eliminates some parameter combinations in the global state space of the model.

Applying all these state reduction methods, the reduced state space for model checking can be expressed as

$$\check{G} = \prod_{i \in I} \text{reduce}(S_i) \times \prod_{j \in J} P_j \tag{6.4}$$

where the index set I is the set of compile-time schedulable subsystem indices, the index set J is the set of indices of reconfigured parameters that affect the component's input/output sampling rates, and the function 'reduce' applies the mode reduction techniques as explained above in rules 3 to 6.

Model-checking by applying the well-known correctness checks of traditional dataflow graphs (see Section 5.2 on page 118) can now be performed for each state in the reduced model state space $\check{G}$. If all checks succeed, the model is correctly constructed. Note that since these checks are performed before execution, they are therefore not performance-critical.

6.1.5 Model Transformation

Due to the hierarchical nature of the proposed model, it is often possible to express the same functionality in different ways. In fact, one representation of a radio can be transformed into another by applying simple rules, without affecting the functionality. This is an important property since the model can be transformed into representations that are easier to synthesise, align more naturally with the resources, and execute more efficiently on any given platform. Thus, the transformations mentioned here are useful during synthesis of the model to a real

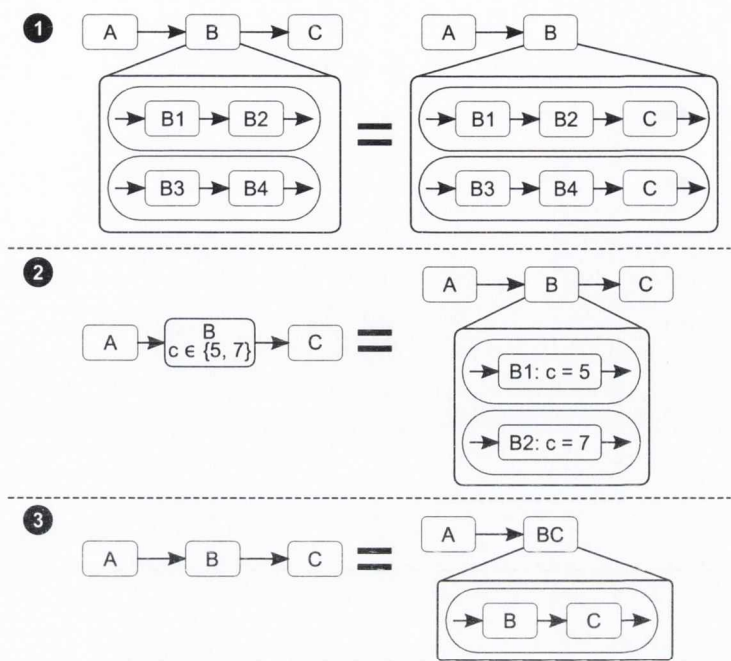


Figure 6.4: Model transformation relationships.

implementation platform.

The following behaviour-preserving transformation rules can be given for the proposed radio model (see Figure 6.4):

- 1. A component at the input or output of a subsystem may be moved into the subsystem by replicating it in all of the subsystem’s modes.
- 2. A component with a parameter with discrete reconfiguration values may be transformed into a subsystem with separate modes for each of the possible parameter values, making them fixed parameters in each of the modes.
- 3. Subgraphs of a dataflow graph might be placed into a subsystem containing only one mode.

An example radio where the transformation rule 1 results in a more efficient implementation is when the components B1-B4 and C in Figure 6.4 are executed on dedicated high-speed hardware, for example field-programmable gate array (FPGA) logic, and component A is executed in software. To avoid costly data transfers between software and hardware and re-routing of data links when a reconfiguration occurs, it is more efficient to connect B2 and B4 directly to C. The first transformation rule allows this direct connection, resulting in a functionally equivalent model which can be executed more efficiently.

For rule 2, assume that component B is a Viterbi decoder and the parameter c represents

the constraint length of the code polynomial. If during model synthesis to a real platform it turns out that the constraint length parameter cannot be reconfigured during execution, two replicas of the Viterbi decoder, each with fixed constraint length parameters, are substituted in a subsystem. This transformation is functionally equivalent, yet, it allows a direct execution on the chosen target platform.

The third transformation rule is useful if during synthesis a subsystem is chosen to represent a dedicated hardware accelerator, running a chain of components more efficiently than the main processor. In that case, the components B and C can be put into this subsystem with only a single mode, resulting in a model with a more direct mapping to the architecture of the target platform.

The transformation rules 1 and 2 only hold strictly when the duplicated component is functional, i.e., when it does not have state. This means in Figure 6.4, that the component C for rule 1 and component B for rule 2 must represent a functional mapping from their input to their output, always producing the same output for the same input (see Section 5.1.2 and Figure 5.2 on page 117). If the components have state, the transformations are not exactly equivalent as the duplicated components might differ in their internal state.

However, in practical scenarios, often this difference in state is irrelevant. Consider, for example, that component C in rule 1 in Figure 6.4 is an finite impulse response (FIR) filter. This is not a functional component, as the delay line is the filter's internal state. However, if a reconfiguration of the active mode in subsystem B is applied, the output is considered invalid until after the filter delay has passed anyway, making the transformed system equivalent for all practical purposes. If component B in rule 2 is a Viterbi decoder, it is meaningless to share the state of the internal Trellis structure between the constraint length 5 and 7 versions. Thus, the transformation is valid in practical systems, even though the Viterbi decoder has state. Therefore, when a model synthesis tool automatically applies these transformations to components with internal state, it can alert the radio designer to allow him to assess whether this is allowed in the given radio.

6.1.6 Summary

The model presented here completely and accurately describes the behaviour of a cognitive radio, including its reconfiguration options. Its execution semantics are clearly defined. A large class of model consistency errors can be detected at design-time, before choosing an implementation platform. The model is implementation-independent, yet, the transformations

presented allow it to be modified to fit a given target platform. Radio designers can design cognitive radios, understand their behaviour and check them for model composition errors without considering a target platform. This greatly reduces the effort needed for testing cognitive radio algorithms in the real world and allows researchers to communicate their designs to each other.

6.2 Model-based Cognitive Radio Design

Having defined the modelling approach, this section discusses how a model can be implemented on physical platforms, creating real cognitive radios for experimentation. First, the general design flow for creating real radio configurations is presented. Then, aspects to consider when synthesising a model are discussed in more detail, in general for arbitrary cognitive radio platforms, and specifically by using Iris-FPGA (Section 2.2) as an illustrative example. Since FPGA-based systems are more complicated to handle than pure software implementations, this example platform is sufficient to demonstrate that the proposed design approach can also be applied to pure software platforms.

6.2.1 General Design Flow

Radio design using the proposed model-based cognitive radio design technique facilitates system-level design and composition of radios using components existent in a component library. Radio designers follow the flow given in Figure 6.5.

In step ❶, the radio designer sketches the model of the cognitive radio to be designed, without considering the target platform or available components. Then (step ❷) the designer maps the desired components to those available in the component information library, possibly substituting components or modifying the model. This library may either be provided by a model checking tool (if this exists), specified by the user, or provided by the target platform (if this is decided already). A more detailed discussion of what information this library contains and where it comes from is given in Section 6.2.2.

The *control specification*, as introduced in Section 6.1.4, specifies which component parameters are required to be run-time reconfigurable and the range or set of values they can take. Further it specifies which events the control plane needs to subscribe to and which components publish these. The model correctness checks (see Section 6.1.4) and model synthesis tools

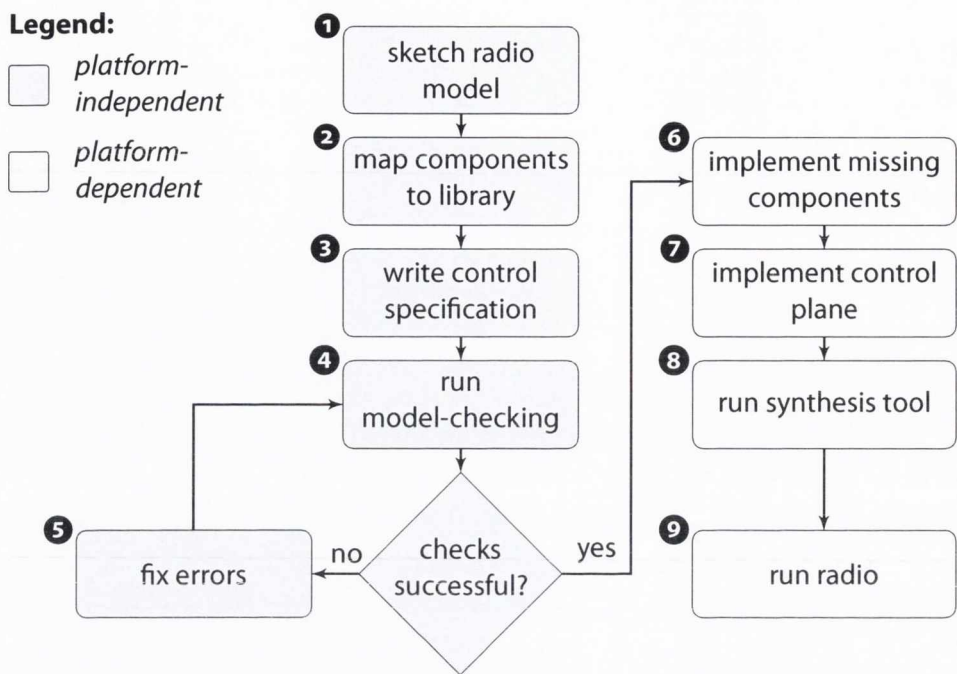


Figure 6.5: Proposed design flow for model-based cognitive radio design.

require this information. The radio designer therefore writes the control specification in step 3 of the design flow.

Then the model can be checked for errors by applying the correctness checks introduced in Section 6.1.4. This can be performed by an automatic tool or manually, if no such tool is available (step 4). The radio designer fixes detected errors and re-runs the checks until the model is constructed error-free (step 5).

The proposed model-based design flow is platform-independent up to this point. Now the radio developer may choose a target platform, provided that model synthesis tools are available for the given platform. If this is not the case, the radio developer has to manually implement the given radio model on the target platform. In such case, the model-based design technique is still beneficial since the model can be used as a design specification for manual implementation.

Once a target platform is selected, missing components can be implemented (step 6). That is, if components are not available for use on the target platform, the radio designer has to implement them manually. The interface and control information obtained from the radio model can be used as a design specification for this step.

The radio designer can now implement the control plane (step 7). Required events need

to be subscribed to, decisions made, and reconfigurations affected. This implementation is inherently platform-specific, since different platforms provide different mechanisms for implementing controllers.

Now the automatic synthesis tools for the target platform can automatically generate a radio configuration that matches the specifications of the model (step ⑧). Critical points and strategies to be considered by such synthesis tools are discussed in the following section. When the synthesis is finished, the radio designer is presented with a radio configuration ready for execution on the target platform (step ⑨). Most low-level tasks have been performed automatically, allowing the designer to concentrate on the behavioural aspects of the radio without concerning himself with implementation details.

6.2.2 Synthesis Considerations

Given a specific radio implementation platform, typically there is no simple one-to-one mapping of all aspects of the proposed radio model to an equivalent on the target platform. Some features can only be supported by implementing additional software code or by using additional tools, and others cannot be supported at all. This limits the set of models that can be synthesised for a given target platform. If unsupported features are absolutely required, radio designers have to consider using a different target platform. The model itself does not require alteration as it is platform-independent.

This section discusses some aspects to consider when synthesising a proposed model to a real platform, outlining potential challenges and solutions. These aspects are discussed in general, for arbitrary target platforms, and the FPGA-based platform presented in Section 2.2 serves as an example for illustration.

MoCs supported by the target platform

Not all of the MoCs supported by the proposed cognitive radio model (Section 6.1) may be supported by the target platform. Typically, only a subset of those, or modified versions are supported.

Some MoCs are compatible with each other (see Section 5.2.7). For instance, all SDF components can also be executed using the CSDF or PN MoC (but not vice versa). See Figure 5.12 on page 135 for a full hierarchy of MoCs. Given the target platform supports a specific MoC, then all MoCs below in the hierarchy shown in Figure 5.12 can also be executed. This execution maybe less efficient than using the modelled MoC directly, but nevertheless it is possible.

For example, Iris-FPGA uses a MoC similar to SDF, with a few limitations. Only directed acyclic graphs are allowed as dataflow graphs for radios, i.e., feedbacks are not supported. However, some radios require feedback. For example, a data-directed synchronisation method at the receiver requires a feedback from a preamble comparison component (after demodulator) back into phase and frequency synchronisation components (before demodulator). Further, branching and merging in a graph is limited, i.e., multiple input or output ports on a component must have the same data type and data consumption/production rates. This limits the branching and merging in a graph. A detailed discussion of the limitations of Iris-FPGA dataflow graphs is given in Section 4.1 on page 93. Since SDF is the most specialised model in Figure 5.12, no other MoCs can be executed by Iris. Therefore, without modification, the target platform only supports cognitive radio models using the SDF MoC with the additional limitations mentioned.

However, to allow synthesis of models applying MoCs not supported directly by the target platform, two approaches are proposed. First, the target platform code could be modified to support more MoCs. This requires access to the source code and often requires complicated changes to the inner workings of the platform. Another approach is to generate a single component for parts of a graph defined in the radio model and implement the MoC within this component's code. That is, a single component running on the platform implements a dataflow graph and the corresponding MoC. This is often easier to achieve than the first approach, but limits the re-usability of components in the library.

If the target platform does not support high-level dataflow-based design at all (see Table 4.4 on page 109), the source code for a full radio has to be generated from the model, applying the appropriate MoC. This includes the scheduling (static or dynamic), memory management, access to hardware parts, etc. Automatic code generation from dataflow graphs has been proposed in many publications, e.g., [120–124]. These methods need to be extended to include the cognitive radio model hierarchy, active modes, reconfigurable parameters, and the interface to the control plane. Although this requires significant work, it is an attractive solution to enable high-level model-based design methods for platforms that otherwise do not support this. The result is an automatic synthesis tool that can be used to automatically synthesise a wide variety of cognitive radios. This enables fast prototyping of cognitive radios for the target platform, based on the generic cognitive radio model proposed in Section 6.1.

Component meta information

The model consistency and correctness checks presented in Section 6.1.4 on page 151 require meta information about the components existing in a component library. Namely, information

about the input/output data rates, supported types, and supported MoCs is required. All parameter names, types, range or allowed values, and whether they are run-time reconfigurable must be known to the model-checking tool.

One solution is to provide this data in a text description together with each component in the library. The Extensible Markup Language (XML) format has proved useful for this data, as it is both human-readable and machine-readable. This approach was taken for the Iris-FPGA platform. That is, each component has an associated XML file, containing the required meta information. These files are used during model-checking, to confirm that the radio model is constructed correctly (see Section 6.1.4).

Another option is to automatically generate this information from the actual components available for the target platform. Typically, these components define an interface through which general information can be queried. If so, the synthesis tools could use this to obtain the component meta information needed.

Interface between control and data plane

The cognitive radio model applies the publish & subscribe MoC for the interface between the control and data planes. That is, the data plane (i.e., processing components) publish events, which the control plane can subscribe to in order to be notified when the events are triggered. See Section 6.1.3 for details.

On cognitive radio platforms and frameworks this kind of interface is typically supported directly. For instance, in Iris-FPGA the interface between the controller and the data plane follows exactly the publish & subscribe model. In GNU Radio [36], this can be accomplished by the provided message queues, where the controller is configured to be a reader of a queue and processing components send events on that queue.

If the target platform does not support controllers and run-time reconfiguration, it cannot be used for implementing cognitive radios. Often it is possible to modify the platform software code to enable such support, although this requires considerable manual work. Radio designers can choose to implement these modifications or to use a different target platform.

Control plane implementation

The proposed radio model does not strictly define the methods to implement the control plane itself (see Section 6.1.3). This is to avoid restricting the user to one model or another, which

might limit the control/cognitive algorithms implementable. Therefore, this implementation is inherently platform-dependent, as shown in Figure 6.5, and may differ significantly between different target platforms.

In Iris, controllers are implemented by registering arbitrary callback functions with each event. That is, these functions are invoked whenever the corresponding event is triggered. This allows a great level of flexibility as virtually any cognitive/control algorithm can be implemented using this method.

If controllers are specified using an FSM (see Section 5.3.2) or Petri Net (see Section 5.3.3), and Iris-FPGA is the target platform, software code must be generated or manually implemented. The callback functions can then be used to post events to the FSM/Petri Net. Reconfiguration actions defined in the model must be translated into the appropriate software code using the Iris-FPGA APIs. This requires relatively little effort.

Transformations to suit the target platform

Often the target platform does not support hierarchical flow graphs in the same fashion as specified by the model. In this case, the transformation rules given in Section 6.1.5 can be applied to the model to create an equivalent representation of the same radio that can be executed by the target platform.

For instance, the Iris-FPGA platform presented in Section 2.2 does not support hierarchical flow graphs at all. GNU Radio [36] does support hierarchy, but allowing multiple modes within a subsystem is not supported. Two solutions to this problem are proposed in the following two paragraphs.

First, multiple modes for a subsystem can be transformed into a flow graph with the graphs for all modes in branches and a switch and select component to select the active branch. An example of this is illustrated in Figure 6.6. This is a flexible approach to transform hierarchies into flat graphs. However, it has two disadvantages. First, the MoC cannot be represented for the subsystem, i.e., if the subsystem applied a more specialised (and more efficient) MoC before this transformation, it has to be executed with the more general (and less efficient) MoC of the next higher hierarchy level after the transformation. Second, all components in all branches of the switch/select structure must be present in the radio at all times, potentially consuming resources, while before the transformation modes could be completely unloaded when the active mode is switched.

Another solution is to represent one level of hierarchy by wrapping a subsystem within a

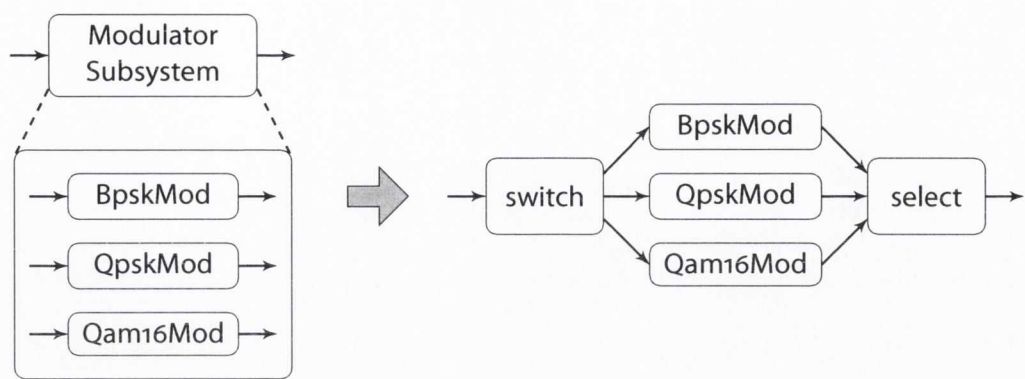


Figure 6.6: Transforming a subsystem into a flat dataflow graph using switch and select components. The active mode is set by selecting the active output and input in the switch and select components, respectively, through a parameter.

single component and exposing a parameter to select the active mode. This is similar to the software wrapper concept used for integrating hardware chains into Iris-FPGA (see Section 2.2.4). The wrapper component, representing a subsystem, can be generated automatically with relatively little effort. Model transformation rules can be applied (Section 6.1.5 on page 158) to transform models with multiple levels of hierarchy into models with only one level of hierarchy, adding modes to the first level subsystems representing all combination of modes in the lower levels.

An example of such transformations is shown in Figure 6.7. It shows the model of a cognitive radio that switches between normal reception and spectrum sensing. Two different sensing techniques are supported, represented by two modes of a subsystem within the sensing mode. The model comprises two levels of hierarchy. Applying the transformation rules, it can be converted to an equivalent model containing only one level of hierarchy, as shown in the lower part of the figure.

Another attractive application of the model transformation rules is to modify the model so that it fits to a hybrid target system. For instance, suppose that implementations of the components in the sensing and reception modes in Figure 6.7 are available for FPGA logic fabric. Then the transformation shown results in a single subsystem containing three different configurations that are implementable in hardware. Recall the customisable processing subsystem (CPSS) of the proposed virtual architecture for the FPGA platform from Section 2.2.2 on page 46. It represents a reconfigurable region of the FPGA logic fabric which can be (re-)configured with chains for radio components in hardware. A software wrapper component is used to represent this CPSS to the remainder of the cognitive radio system, managing all interactions with the hardware. Now, the single subsystem in the lower half of Figure 6.7 can be considered

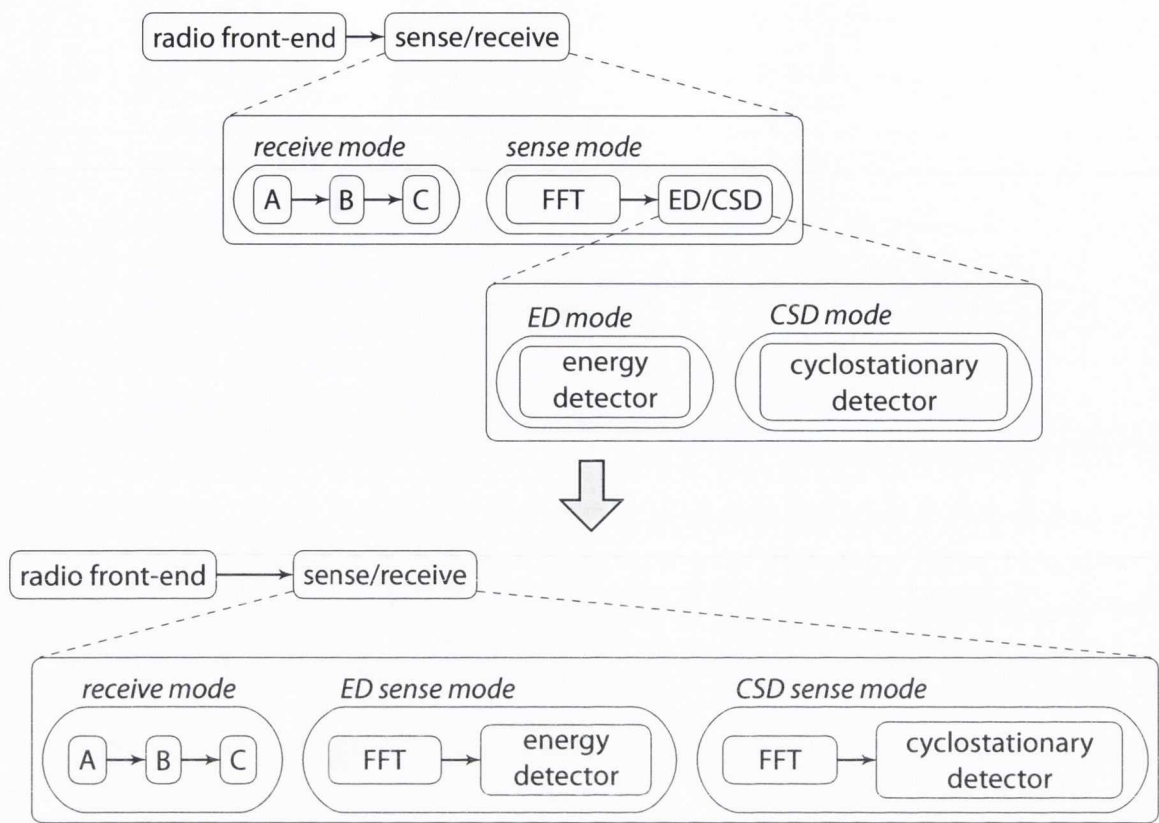


Figure 6.7: Example transformation from multiple levels to one level of hierarchy. A radio switching between sensing and reception is shown, supporting two different methods for sensing. The components in the receive mode are for illustration purposes only.

as a software wrapper component for a CPSS in the FPGA. The modes within can efficiently be executed in hardware, and mode switches map to hardware reconfigurations. Thus, this transformation results in a model that is optimally suited for the hybrid processor and FPGA platform proposed in Section 2.2, illustrating how transformations can be used to create models that fit hybrid target platforms ideally.

Providing a high level interface to the radio controller

The applied model transformations, wrapping subgraphs in wrapper components, and other conversions of the model in order to create an implementation for the target platform result in a system that looks very different from the original model. This creates difficulties to radio designers when implementing the radio controller. Modes and subsystems have been modified, wrapper components have been created, and the graph has been transformed. Ideally, the radio controller should see the original model, not the transformed implementation.

As solution for this problem, it is proposed to generate an abstraction layer between the running radio and the designer-implemented radio controller. The controller accesses this layer as if the original model was executed, seeing all subsystems and modes as specified during the design. The abstraction layer translates all events and reconfiguration actions to the currently running transformed implementation, hiding these transformations from the user. During synthesis, the parameters and translation rules of this abstraction layer can be generated automatically since the synthesis tool knows the applied transformations exactly.

This abstraction layer keeps the radio design at a high level from the user's perspective, while still allowing significant transformations and optimisations of the model to suit the target platform.

6.2.3 Synthesis Steps

A synthesis tool should adhere to the following steps in order to create an implementation from a given radio model:

1. Check availability of all used components in the platform-specific component library.
2. Check if the used MoCs are implementable on the target platform.
3. Transform the model to suit the target platform, optimising for efficiency. Use the *control specification* for information about which parts of the radio need to be reconfigurable.
4. If required, generate software wrapper components for subsystems.
5. Generate abstraction layer to provide high-level view of the implementation to the radio controller
6. Create radio configurations for the target platform

The result is a radio system ready for execution on the target platform. The majority of the platform-dependent low-level tasks can be managed automatically, enabling the user to focus on the important aspects of radio design.

6.2.4 Limitations

Although the proposed model-based design technique simplifies the design of cognitive radio systems significantly, it has its limitations. These are discussed in the following.

Only the system composition is platform-independent

The model-based design technique proposed only concerns itself with the composition of the radio from components that already exist. Components that do not exist need to be implemented the traditional way. This is inherently platform-dependent and requires low-level development expertise for the target platform.

However, implementing single components with a well-defined interface (obtained from the model's component information) is far better than implementing a full radio system. The system composition as well as the radio reconfiguration aspect is covered by the model-based design technique and automated tools.

Additionally, depending on the components to be implemented, there might be implementations already available in other resources. For example, for software or digital signal processing (DSP) processor implementations, implementations may be taken from available software libraries or DSP tools. For FPGA implementations, intellectual property cores which implement the desired functionality might be readily available.

Model correctness checks do not include functional correctness

Even when all model correctness checks complete successfully, that does not guarantee that the radio does what it is supposed to do. That is, it cannot be checked whether a receiver radio can in fact receive and decode the transmitted data correctly. The model-checking stage is ignorant of the function of each component, i.e., it views them as black-box components. It is only concerned with the composition of the radio model, the parameters, links, events, and reconfigurations.

For this reason, a good understanding of communication systems and algorithms is an absolute must. The radio designer must understand how the components work and what communications algorithms and tasks they realise. Thus, the target audience for this model-based design technique are communications engineers experienced with communication algorithms and principles, but not necessarily experienced with the hardware and software implementation of these.

It would be possible to check the functionality of the radio if the radio model could be simulated in a design system (without implementing it on a target platform). This requires an algorithmic description of the internals of all components, e.g., MATLAB code that matches the native implementation of the component. Then the radio models could be simulated and their correct function could be confirmed before implementing them on the target platform.

However, this becomes inherently platform-independent, as the algorithmic description must match a specific native implementation for a target platform. Additionally, it requires considerable effort to maintain a matching algorithmic description for all components available. This could be done in a commercial tool but is beyond the scope of this thesis.

System performance cannot be estimated

It would be ideal if the expected performance of the composed radio system could be estimated for several available target platforms during the design. This could aid the choice of the target platform. For example, a complex radio configuration of a transceiver, applying advanced modulation, channel estimation, equalisation, and forward error correction, might be too complex to execute on a general purpose processor (GPP) built into today's PCs. In this case, using GNU Radio [36], a framework designed for GPPs, is not an option. Only platforms including high-performance hardware are suitable target platforms in this case. However, since the components are viewed as black-boxes during the design, the radio designer might not know this and attempt to synthesise the radio for GNU Radio. The result would be a radio that cannot be executed in real-time.

A possible extension to allow performance estimations is to benchmark all components available for a target platform thoroughly and add the results to the component information library. This could then be used to estimate the expected performance of the composed radio, in all possible configurations. However, this requires considerable effort and becomes inherently platform-specific. It is beyond the scope of this thesis to consider performance estimation; this is left for future work.

6.2.5 Summary

It was shown that the proposed platform-independent model can be mapped to a real implementation platform, applying the design flow proposed in Section 6.2.1. Automatic synthesis tools can perform this mapping, reducing the development effort for the user. Important aspects to be considered by these tools have been presented. These represent a guideline for implementing such tools for a given target platform.

The target platform for the proposed technique can be any cognitive radio framework, for instance, the frameworks discussed in Chapter 1 on page 11. This is illustrated in Figure 6.8. Synthesis tools take a platform-independent cognitive radio model and produce a platform-dependent radio configuration for the given target platform largely automatically. If no such

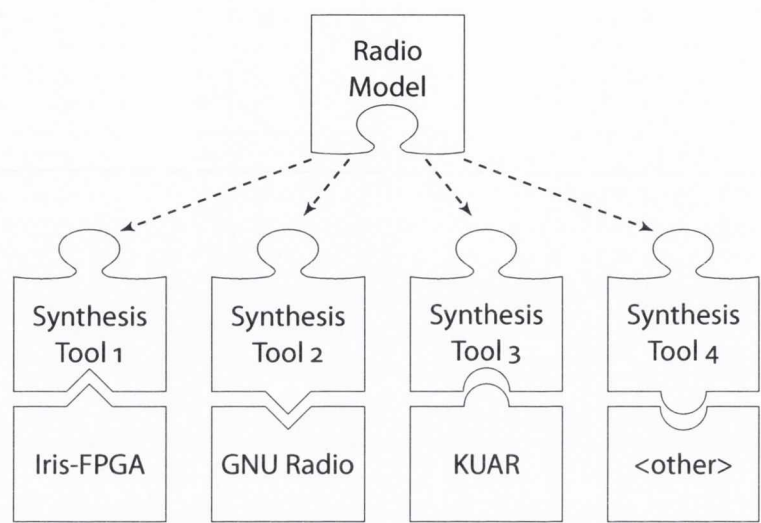


Figure 6.8: Application of model-based design technique for arbitrary target platforms. The synthesis tools transform the model into a configuration suitable for the chosen target platform largely automatic. Note that the implementation of the controller and missing components still needs to be performed manually for each target platform, as shown in Figure 6.5.

tool exists for the target platform of choice, the synthesis steps given in Section 6.2.3 can also be followed manually. Although this involves manual low level implementations, the synthesis steps provide a systematic flow to map a model to an implementation, something that is lacking from the existing frameworks (see Section 4.2 on page 105).

Hence, the proposed platform-independent model can be used for model-based design of cognitive radios for arbitrary target platforms, simplifying the development of such systems significantly.

6.3 Implications for Cognitive Radio Frameworks on the Example of Iris-FPGA

Up to this point, the thesis has shown how practical experimentation can lead to new theoretical ideas. That is, the experiments and examinations from Part I led to the proposed model-based design technique based on a theoretically substantiated model for cognitive radios. This section presents an example of how the theory can lead to new practical ideas. The ideas of the proposed radio model and model-based design technique can influence the way cognitive radio frameworks are implemented, i.e., the target platform itself can be adapted to better fit the model. This shows how theory and experimentation work in concert to continuously

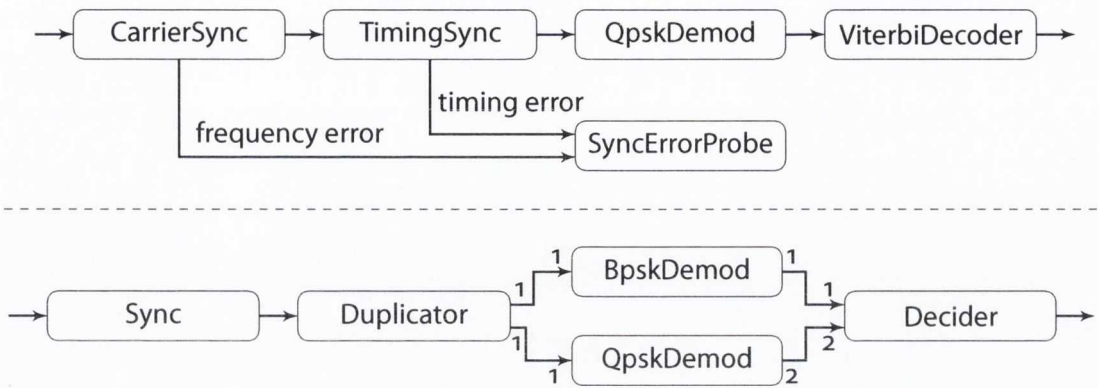


Figure 6.9: Example flow graphs not realisable in Iris-FPGA. Feeding a component from the outputs of two different components is not possible (top). Multiple data rates at a multi-input component are not permitted (bottom).

improve each other.

Before conducting a case study for the proposed model-based design technique (Chapter 7), this section illustrates, on the example of Iris-FPGA, how the ideas of the model can influence cognitive radio frameworks. This not only eases the synthesis steps for model-based design, it also broadens the capabilities and increases the flexibility of the target platform.

In the following, aspects of a new generation of the Iris-FPGA framework, *Iris2-FPGA*, relevant to the proposed model-based design technique are detailed. A complete overview about Iris 2 in general is given in [125].

6.3.1 Flexible Dataflow Graphs

One major limiting factor for applying the proposed model-based design technique to Iris-FPGA as a target platform is the limited set of dataflow graphs it supports. Recall from Section 4.1 on page 93 that Iris-FPGA is mainly made for chains of components without branching and merging. Components typically have only one input and one output port. Multiple ports are possible, but only if all have the same data type and sample rates. The links in a flow graph are implicit, deduced from the component position in the XML file defining the radio (see Section 1.3.1 on page 22 for a description of the XML format). This limits the graph structures that can be described. That is, structures like the examples shown in Figure 6.9 cannot be realised with the existing version of Iris-FPGA.

The proposed radio model, however, allows arbitrary dataflow graphs at any hierarchy level. It defines the dataflow semantics in systematic way, clearly defining allowed dataflow structures

(see Section 5.1.2). When the model is mapped to Iris-FPGA, only a small subset of radios can be synthesised directly. When a radio model includes a dataflow structure not supported by Iris-FPGA, the synthesis tool has to generate software code for a wrapper component which contains the problematic subgraph within a single component. This manages the execution of the contained graph internally, implementing scheduling, memory management, FIFO buffers, etc. This is a complex task, prone to errors and difficult to debug. Effectively, the synthesis tool has to compensate for a missing feature of the target platform.

Therefore it was decided to allow general dataflow structures in Iris2-FPGA. Radio configurations are defined to be dataflow graphs, with all their properties and implications (see Section 5.1.2 on page 116). Components now support multiple input and output ports of arbitrary types and sampling rates. Further, the format of the XML files has been changed, now explicitly listing the links between the components (see Appendix E for details). This combination overcomes the limitations of the Iris-FPGA radio configurations and allows a more direct synthesis of the proposed radio model.

6.3.2 Iris Engines

Another limiting factor of Iris-FPGA is the limited support for advanced dataflow MoCs. Only a limited version of SDF is supported, where components are executed once in a scheduling cycle in a round-robin fashion. This limits the amount of flow graphs that are executable with Iris-FPGA and provides low flexibility.

This limitation is overcome in Iris2-FPGA by using the concept of the *Iris engine*. This encapsulates one or more components of the overall dataflow graph and defines the data-passing, execution, and reconfiguration semantics for those components, i.e., the MoC. A radio implemented in Iris2-FPGA may consist of one or more of these engines, i.e. combinations of different MoCs are possible.

Currently, three types of engines are implemented, each suited to one particular type of dataflow graph. The SDF and PN engines implement the corresponding MoCs and are suited for the physical layer of a radio node. Furthermore, a network stack engine is available, supporting the higher layers of the network stack. This combination of engines allows for efficient and flexible implementations of cognitive network nodes.

All engines within the Iris2-FPGA architecture implement a common interface for data-passing, reconfiguration, events, and life cycle management. This common engine interface ensures that all engine types can be handled in the same manner by the overall Iris2-FPGA

system. One advantage of this approach is the ability to implement controllers which are independent of the engine type. A single controller can for example reconfigure a component in a network stack engine in response to an event triggered by a component in an SDF engine. A further advantage of the common engine interface is the ability to extend the architecture and add new MoCs through the addition of new Iris engines in the future if required.

Since this thesis focuses on the physical layer only, a detailed description of the network stack engine is beyond its scope. In the following, the operation of the SDF and PN engines are detailed.

SDF Engine

This type of engine implements the SDF MoC [101], as explained in detail in Section 5.2.2 on page 122. Components managed by the SDF engine typically lie close to the digital/analog interface, operate at high sample rates and are not reconfigured on a regular basis. For these components, fixed rates of data consumed and produced at the ports can often be established. Examples of components which might operate within the SDF engine are filter, modulation, or interpolation components.

The SDF MoC is characterised by the static sample rates which exist on each port of components within the flow graph. This knowledge allows a thorough analysis of the graph at design-time: required buffer sizes on each link can be determined, guarantees for deadlock-free operation can be given, and static and efficient execution schedules can be determined. It provides the most efficient execution for components implementing fixed-rate signal processing functions.

In Iris2-FPGA, the SDF execution schedules are computed before the radio execution is started. The input and output sample rates of components are queried, the topology matrix and repetition vectors are established, and a schedule is computed. During execution, the SDF engine follows this precomputed schedule for the part of the radio it manages.

If a graph managed by the SDF engine is reconfigured during execution, and this reconfiguration affects the execution schedule, the engine recomputes the schedule at this point. In complicated graphs, this scheduling computation might affect the performance during reconfiguration. To overcome this problem, Iris2-FPGA stores the previous configuration and schedule in a cache, so that it can be loaded again in case the graph is reconfigured to its original state, avoiding the overhead of schedule computation. This caching allows a smooth transition of the graph execution when reconfigurations are applied.

PN Engine

This type of engine implements the PN MoC [96, 97], as explained in detail in Section 5.2.1 on page 119. It is best suited to encapsulate the highly reconfigurable components which provide high flexibility of the physical layer of the node. These components typically exhibit flexible input and output data rates. A reconfigurable OFDM demodulation component, where the number of subcarriers and the length of cyclic prefix changes frequently, might operate within the PN engine for example.

The PN MoC supports the greatest degree of dataflow flexibility. All components are processes that execute asynchronously, producing and consuming data at flexible data rates. It is the most general MoC of those reviewed in Section 5.2 on page 118, all valid dataflow graphs have an execution using the PN MoC.

Implementation of the PN engine using a straightforward PN MoC would involve assigning a single thread of execution to each component. One disadvantage of this approach however, is the overhead incurred as a result of thread context switches in the case where many components execute in parallel on a low number of processor cores. In order to avoid this overhead and achieve tighter control over the number of executing threads in a given radio, a different approach was adopted. Components which execute in an PN engine operate according to a PN MoC. However, each PN engine contains a single thread of execution and within that engine a PN MoC is emulated. Parallel execution of components is achieved through use of multiple PN engines. Thus, given a chain of 3 components, 1, 2 or 3 parallel threads of execution can be employed through use of 1, 2 or 3 PN engines. In this way, the PN engine design leverages the flexibility of the PN MoC while providing the radio designer with tight control over the number of executing threads in the radio. While the PN engine supports significant component flexibility, the associated dynamic allocation of computing and memory resources, the inter-thread synchronisation required for the buffers on the links, and the context switches between threads incurs a higher overhead than with the SDF engine.

6.3.3 Component Hierarchy

According to the hierarchy of MoCs given in Figure 5.12 on page 135, the SDF MoC is a specialised version of the PN MoC. Therefore, Iris2-FPGA implements a class hierarchy reflecting this fact, as illustrated in Figure 6.10. The Iris engines are implemented to support only one type of components, but since the an Iris2-FPGA SDFComponent “is a” PNComponent, it can also be executed within a PN engine.

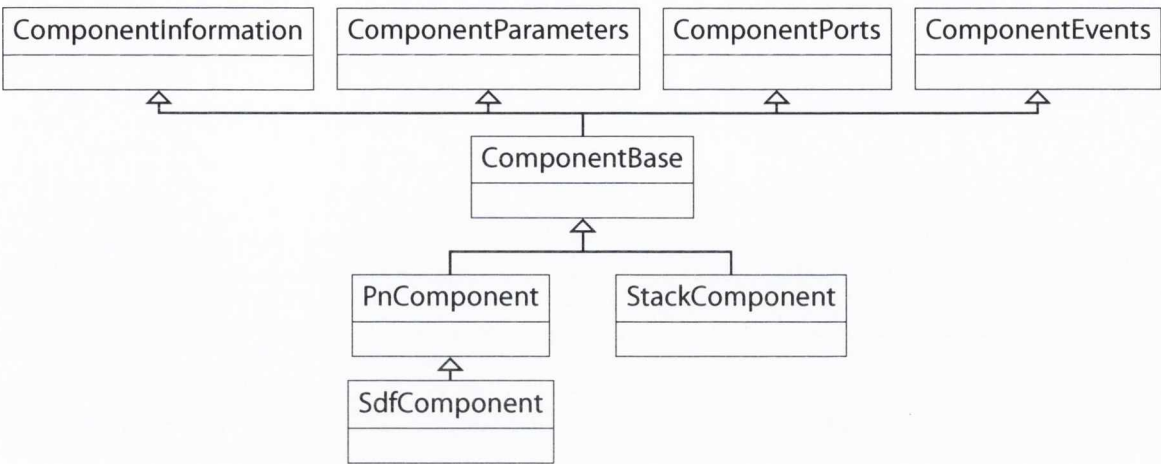


Figure 6.10: The component class hierarchy of Iris2-FPGA.

All components inherit from the `ComponentBase` class, to give Iris-FPGA an generic interface for all components. This interface consists of methods for obtaining general information about components (`ComponentInformation`); setting parameters (`ComponentParameters`); declaring input and output ports (`ComponentPorts`); and registering, subscribing, and activating events (`ComponentEvents`). The `PnComponent`, `SdfComponent`, and `StackComponent` classes implement the specifics required by the corresponding MoC. They are loaded and managed by the corresponding Iris engine type.

All specific component implementations, e.g., an OFDM component, inherit from one of the `PnComponent`, `SdfComponent`, or `StackComponent` classes and implement the relevant methods. They are compiled into shared libraries, loaded from a repository by Iris2-FPGA, and accessed through the interface defined by their base classes.

This allows Iris2-FPGA to handle components for different MoCs through a common interface and it captures the hierarchy of MoCs. If a new dataflow MoC was implemented in Iris2-FPGA, a component class needs to be added at the appropriate point in the class hierarchy. For instance, a CSDF component class would be inserted between the `PnComponent` and `SdfComponent` classes.

6.3.4 Integrating FPGA Hardware

The virtual architecture (Section 2.2.2 on page 46), and the concepts for integrating FPGA hardware component chains within Iris2-FPGA have not been altered from Iris-FPGA. That is, software wrappers are used to handle all hardware interfacing (see Section 2.2.4 on page 57),

and components are chained together and configured into a CPSS on the FPGA. Thus, the hardware aspects of the framework remained the same.

6.3.5 Discussion

The outlined changes in the Iris-FPGA cognitive radio framework were enabled by applying the structured methods from the proposed cognitive radio model (Section 6.1) to the platform itself. The new generation of Iris-FPGA, Iris2-FPGA, has a clearly-defined theoretical background, based on dataflow graphs and their MoCs.

The modifications of Iris-FPGA greatly simplify the automatic synthesis of arbitrary cognitive radio models. The dataflow graphs supported have been improved, matching the theoretic dataflow graphs used in the proposed radio model. The PN MoC, the most general dataflow MoC, is now supported directly by Iris2-FPGA. This means that any other dataflow model is implicitly supported as well, since these are specialisations of the PN MoC. Additionally, the SDF MoC supported in Iris2-FPGA allows an efficient execution of relevant parts of a given radio model, avoiding conversions to PN. The component class hierarchy reflects the hierarchy of MoCs given in Section 5.2.7 on page 135, enabling the execution of SDF components in a PN context.

The improvements of Iris-FPGA illustrated in this section serve as an example to demonstrate how the proposed structured and theoretically substantiated radio model can affect the design of cognitive radio platforms. The result is a versatile platform that allows implementing a wide range of radios based on well-defined and well-understood models.

6.4 Summary

A novel approach to developing cognitive radios has been presented in this chapter, based on a high-level radio model. Existing relevant models have been combined and extended to capture the specific needs of cognitive radios. The proposed model describes the radio composition and dynamic behaviour, including all possible reconfigurations. Correctness checks have been presented, allowing error detection at design-time. The proposed model allows radio designers to develop and reason about cognitive radio applications while being decoupled from specific implementations. It has been shown that the model can be transformed to fit a variety of target platforms.

A method for using this model for automatic synthesis of real implementations has been outlined. A design flow for model-based design has been given, along with steps to be taken by automatic synthesis tools. The Iris-FPGA cognitive radio framework, presented in Section 2.2, has been used as an example target platform in the discussions.

Further, it has been shown how the ideas of the proposed radio model can be used to design or improve cognitive radio platforms themselves. The Iris-FPGA platform has been modified to integrate many features of the model, now supporting a significantly wider range of cognitive radio systems. When this new version of Iris-FPGA is used as a target platform for model-based design, the tasks for automatic synthesis tools are simplified greatly.

Radio developers following the proposed model-based design technique do not require detailed platform knowledge. The development process requires less manual effort, and the design can be easily ported to other target platforms. By applying the proposed model-based cognitive radio design approach, researchers are able to test their algorithms in the real world with significantly reduced effort. It is believed that this can drive cognitive radio research forward significantly.

7 Case Study

Contents

7.1 Case Study Description	182
7.2 Model-based Radio Design	184
7.3 Radio Implementation	210
7.4 Analysis and Discussion	221

This chapter illustrates the benefits of applying the proposed model-based design technique (Chapter 6) by going through the design and implementation process of an example cognitive radio system. All steps of the proposed development process are detailed, from the requirements to the execution of the radio. The benefits of the proposed design technique compared to traditional methods are clearly shown throughout the process.

A frequency-agile cognitive radio system that adapts the applied forward error correction scheme to the current environmental conditions is chosen as the case study example. It implements the same behaviour as the experiment conducted in Section 3.4 on page 84 but the proposed radio design methods are applied. This allows to compare both design approaches. A detailed description of the radio to be designed is given in Section 7.1.

The radio is designed by applying model-based design, as proposed in Chapter 6. It is described using a platform-independent radio model. The model captures the radio’s structure and dynamic behaviour, while platform-specific details are avoided. The radio designer can focus on the application design on a high level; he does not need to concern himself with implementation details. Section 7.2 details the necessary design tasks for the example radio.

Creating a real implementation from the radio design is the next stage. For this, a target platform is chosen by the developer. In this case study, the Iris2-FPGA platform was selected,

executing on an embedded field-programmable gate array (FPGA) system. This is to demonstrate the implementation on a realistic platform capable of high performance processing and dynamic reconfiguration. Section 7.3 describes the implementation process. It can be handled by largely automatic using synthesis tools, so that the user does not need to concern himself with implementation. The result is a working radio setup with the same behaviour as specified in the platform-independent model.

Finally, this whole development process is critically assessed and compared to traditional design methods, outlining the advantages and disadvantages of the proposed approach. Section 7.4 gives the details of this analysis.

7.1 Case Study Description

The radio setup to be designed should demonstrate the capabilities of the proposed design and implementation approaches and the whole development process is to be compared to traditional design approaches. For this reason, the adaptive coding radio link combined with sensing at the receiver from Section 3.4 was selected as an example. Developing this radio using the proposed approach can be compared to the development process undertaken in Section 3.4, pointing out differences, advantages, and disadvantages. The adaptive coding and sensing radio application is described in the following.

A radio system for the transmission of video data over an adaptive link is to be developed. The transmitter sends convolutionally-encoded and differential quadrature phase shift keying (DQPSK)-modulated video data over a wireless link. The receiver demodulates and decodes this data, then displays the transmitted video.

To maintain a given bit error rate (BER), the strength of the coding scheme is adapted to changes in the currently experienced BER at the receiver. Stronger (and more complex) coding methods are chosen when the channel quality worsens, and less complex methods with better quality channels. If the channel quality allows, the coding may be switched off completely. Convolutional codes are increasingly stronger with increasing constraint length of the code polynomials applied. In other words, higher constraint length codes result in lower BER (see Figure 3.11 on page 82 for the BER versus signal to noise ratio (SNR) curves of different codes). The receiver informs the transmitter when the coding strength should be changed. This could be done wirelessly, but for the sake of simplicity of the experiment, this information is sent over an Ethernet connection.

In addition, the receiver radio does not know the exact transmission carrier frequency beforehand. Only maximum and minimum frequency boundaries are known, the transmission signal might be anywhere between these. The receiver estimates the transmission frequency blindly by performing a spectrum sensing technique in search for a transmission signal. When a potential signal is found, the receiver attempts to demodulate and decode the signal. If this is successful, the video stream is displayed and the radio continues to demodulate. If not, the search for the transmission signal continues, until the signal is found.

The transmitter may change its carrier frequency without warning at any time. This is to emulate the appearance of a primary user in a dynamic spectrum access scenario. If the receiver loses the transmission signal, it starts searching for the signal as described above.

The system continues to function in this fashion, adapting the coding strength according to the signal quality and changing the transmission frequency at intervals.

A summary of the cognitive radio system to be designed in this case study is given in the following list:

- Application: Video transmission
- Network: Two radios, single unidirectional link
- Modulation: DQPSK
- forward error correction (FEC): Convolutional coding with varying constraint length
- filtering, framing, and synchronisation as required
- Adaptive coding:
 - current BER is observed at the receiver
 - coding constraint length is adapted, higher for higher BERs, and lower otherwise
 - coding can be disabled when the signal quality allows
- Frequency agility:
 - transmitter sends on arbitrary frequency in a given band
 - receiver scans that band, applying spectrum sensing techniques, in search for the transmission signal
 - once found, the radio front-end is tuned to the frequency and demodulation is attempted
 - if successful, the receiver displays the video
 - if not, the receiver continues sensing
 - if the signal is lost, sensing proceeds to find the signal again, as described above

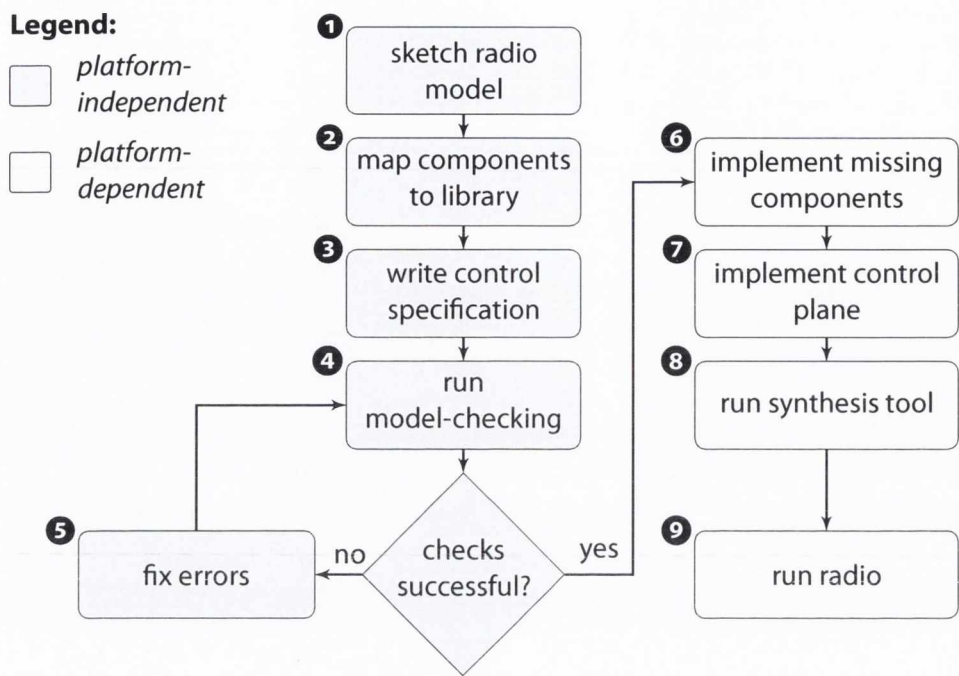


Figure 7.1: Proposed design flow for model-based cognitive radio design.

7.2 Model-based Radio Design

To design the radio described above, the model-based design technique proposed in Chapter 6 is employed. For convenience, the illustration of the design flow proposed in Section 6.2.1 on page 161 is repeated here, in Figure 7.1. The following sections detail the platform-independent in the design flow, i.e., steps 1 to 5.

7.2.1 Sketch Radio Model

The first step in every design is to sketch the radio’s data plane as a hierarchical dataflow graph. This is step 1 in Figure 7.1.

Transmitter

For this case study, the transmitter consists of a simple narrowband chain of radio components. The first is a *video source*, outputting video streaming data for encoding and modulation. A *framer* component is next, adding a frame access code, a checksum and some other information to assist the receiver in recovering the data and checking whether it is correct. The framed data is then encoded by the *encoder* component, applying convolutional codes of varying constraint

length. The data is then modulated in a *DQPSK modulator*, mapping information dibits to phase shifts in the complex baseband. To control the excess bandwidth of the transmitted signal, caused by the side-lobes of the signal in frequency domain, a *pulse shaping filter* is inserted next. Now the signal is ready to be transmitted over the *radio front-end*, the final component in the transmission chain.

All components in the radio graph are synchronous, i.e., they have fixed input and output data rates associated with their ports. This makes the synchronous dataflow (SDF) model of computation (MoC) ideal for their execution, allowing thorough analysis and the computation of efficient execution schedules at design-time, before the radio is executed.

The control plane of the transmitter, implementing the cognitive cycle (*observe*, *decide*, and *act*), is simple. It has to change the carrier frequency of the radio occasionally, to emulate the appearance of a primary user and the resulting sudden change in frequency. This also tests the receiver's sensing capabilities in this case study. Also, the constraint length of the convolutional encoder has to be adapted from the control plane, according to information received from the receiver. It should also be possible to disable the coding altogether, if the channel quality permits. Therefore, the encoding is encapsulated in an encoder subsystem with two different modes: one for coding enabled and one for coding disabled. If enabled, the constraint length of the used coding polynomial is a reconfigurable parameter. No events from the radio need to be subscribed to, since reconfigurations are triggered either by the receiver or by a timer in the controller.

The resulting radio model for transmitter is shown in Figure 7.2.

Receiver

The receiver of the cognitive radio system to be designed is more complex. Two different operating modes need to be supported, one for normal reception and one for detecting the transmission signal within a given band. The intelligence of the system resides in the receiver; decisions about the operating mode and the coding strength are made here.

The two required operating modes only have one element in common: the radio front-end. It obtains digital baseband data from the front-end device. Therefore, the top level of the hierarchical model only contains the radio front-end component followed by a subsystem containing the two operating modes: reception and detection.

In reception mode, the radio is required to receive, demodulate, and decode the transmitted video data. It therefore contains a simple DQPSK receiver chain. To optimise the SNR of the

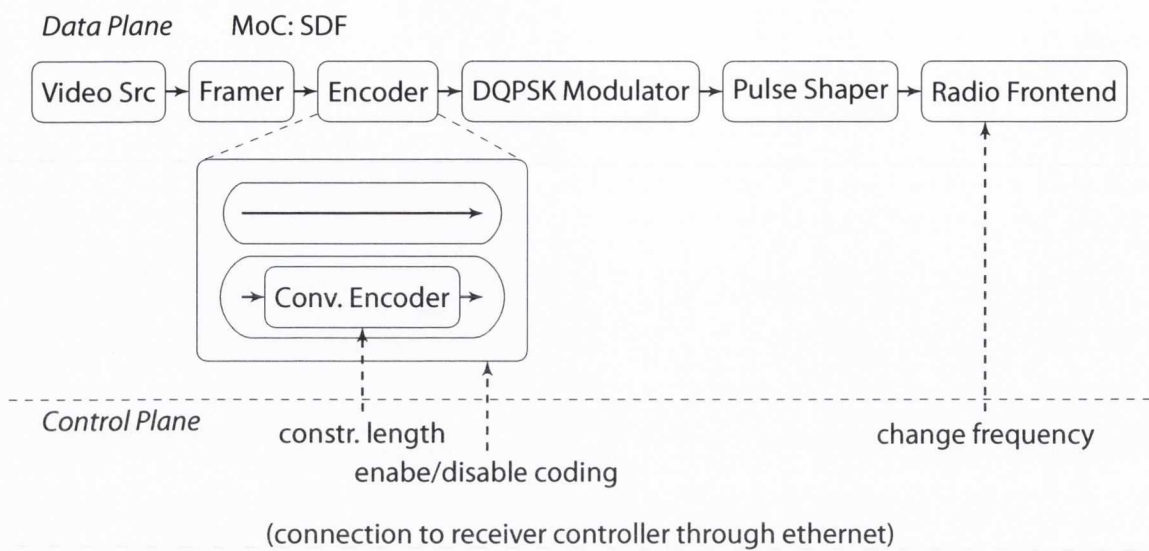


Figure 7.2: The radio model of the transmitter. The coding subsystem has two modes: coding enabled and coding disabled. Further, when coding is enabled, the constraint length is a reconfigurable parameter.

signal, the first component in that chain is a *matched filter*. If this filter matches the pulse shape of the transmitted symbols, noise outside the signal’s frequency band is filtered out, so that the signal quality increases. To correct timing and frequency offsets in the received signal, introduced by offsets in the local oscillator and sampling stage of the radio front-end, *synchronisation* is performed. Then the symbols can be demodulated by a *DQPSK demodulator*, before the data is decoded in the *decoder* stage. Since the receiver has to support coded and non-coded data, the decoder is modelled as a subsystem containing two modes: one for disabled coding and another for enabled coding. An efficient way to decode convolutionally encoded data is to apply Viterbi decoding. Thus, the coding-enabled mode of the decoder subsystem contains a *Viterbi decoder* component. Now the decoded data can be passed to a *deframer* for detection of the frames embedded in the data stream, extraction of their payload, and checking whether the data was correctly received using checksum comparison. This data is then passed to a *video sink*, which displays the video on the screen.

When the receiver is in sensing mode, it applies the algorithm outlined in Section 3.2.1 on page 70. First, the power spectral density (PSD) of the incoming signal data is estimated and then the carrier frequency of the transmission signal is determined by windowing and thresholding the PSD. Thus, two components are required in sensing mode: a *PSD estimator* and a *carrier detector*.

As for the transmitter, it is possible to assign fixed data rates on the inputs and outputs

of all components in the receiver, i.e., all components are synchronous. Therefore, SDF is a suitable MoC, allowing for highly efficient execution and thorough design-time analysis of the radio.

The control plane of the receiver needs to decide when to switch the operating mode (between reception and detection), and it has to adapt the coding strength according to the observed BER. Thus, as meters during the *observe* stage of the cognitive cycle, it needs information about the current BER during reception, the estimated carrier frequency during detection, and when the received signal is lost. The BER can be estimated by the Viterbi decoder, by re-encoding the decoded data and comparing with the input. Thus, the Viterbi decoder can report the BER in an event. When coding is disabled, no Viterbi decoder is present in the system. In that case, the frame error rate (FER) can be used as a measure of the signal quality, reported by the deframer component in an event. In detection mode, the deframer knows when the signal was lost, i.e., when no frames were found, in which case it triggers an event to the control plane. The carrier detection component publishes an event to inform the controller of the carrier frequency it has estimated.

For detection mode, it is desired to scan the whole permitted band at once, so that the detection is fast and the receiver can switch back to reception as quickly as possible. However, due to practical limitations of radio front-end bandwidth as well as the computational power of the baseband processing, this can often not be achieved. In that case, the band has to be scanned in smaller subbands, overlapping to avoid a missed detection at the edges. This detection scheme is illustrated in Figure 3.4 on page 72. Still, the bandwidth of the radio front-end used during detection should be as wide as possible to allow a quick completion of the scan. Thus, the control plane has to implement subband-scanning and needs access to the bandwidth parameter in the radio front-end component.

The radio model sketched for the receiver side is illustrated in Figure 7.3.

The controller itself can be described using a Unified Modeling Language (UML) statechart, an extended version of finite state machines (FSMs) (see Section 5.3.2). This represents the *decide* stage of the cognitive cycle. The events from the data plane are used as inputs (meters) and with each state representing a configuration of the data plane. The same state machine as used in the experiment in Section 3.4.3 on page 88 can be applied here. It is a hierarchical statechart with two outer states, one for reception and one for detection. Within detection, it has four different states representing no coding, and three different code constraint lengths: 5, 7, and 9. The constraint length is increased when the observed BER is above predefined thresholds, and decreased if it goes below the associated thresholds. When the receiver is

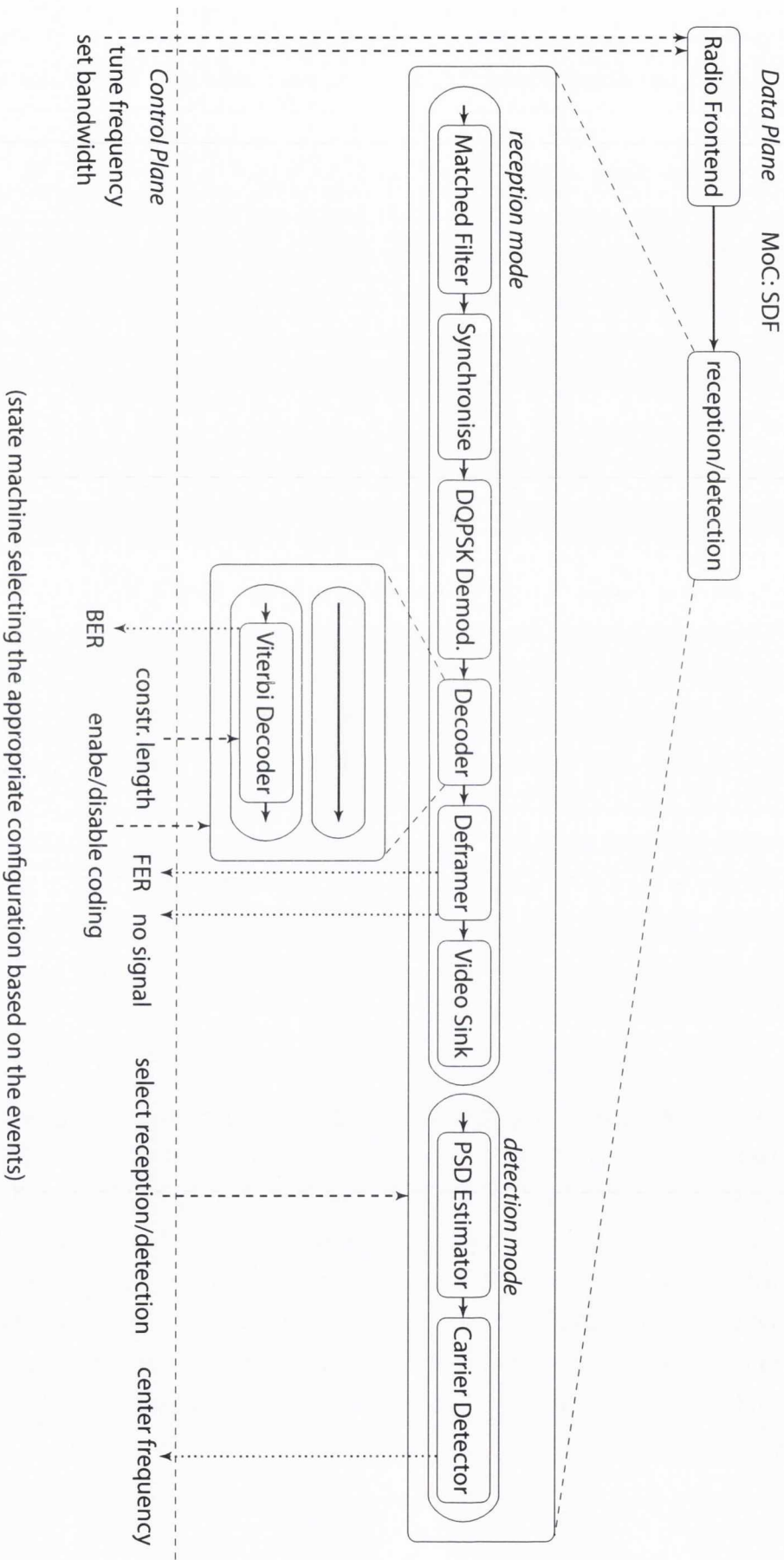


Figure 7.3: Radio model for the receiver.

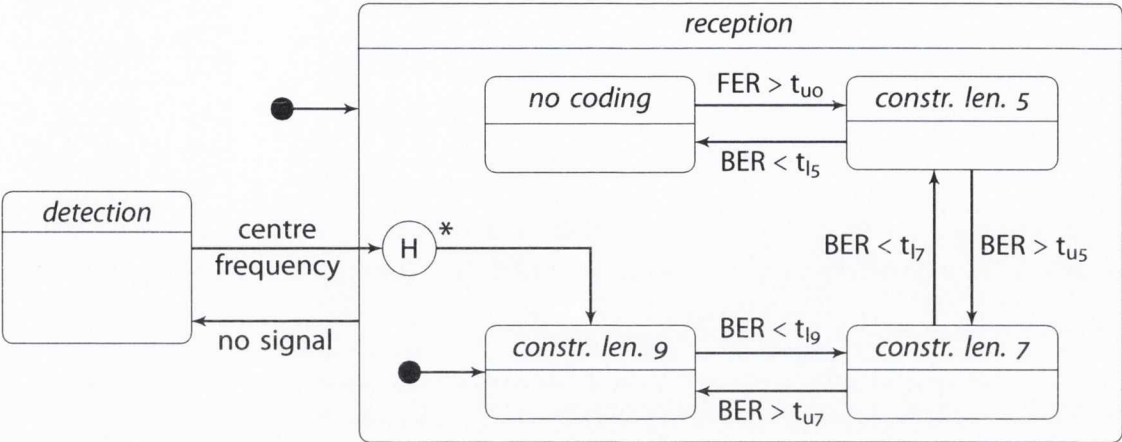


Figure 7.4: Controller statechart for the receiver radio.

in the constraint length 5 state and the BER is low enough that coding can be disabled, the controller switches to the ‘no coding’ state. In this state the FER is monitored, and if it goes beyond a predefined threshold, the coding is enabled again, i.e., the statechart is switched to the constraint length 5 state. The reception state, containing the coding states, has history, i.e., it returns to the previous internal state when the state machine enters the reception state. The controller state machine is depicted in Figure 7.4, repeated from Figure 3.14 on page 90 for convenience.

Having sketched the radio models of both the transmitter and receiver, the radio developer can now proceed to the next step in the design flow.

7.2.2 Map Components to Library

In the next step (step 2 in Figure 7.1), the components occurring in the sketched radio model are mapped to those available in the library. This library contains information about all components available, their parameters, ports, events, and general component information.

As discussed in Section 6.1.4 on page 151, This library could be provided by the target platform, containing information about the components available. If the target platform is not decided yet, two possibilities are possible. Either the model-checking tool used during the platform-independent stage provides a library of the required component information (but not their implementation), or the radio designer has to provide this information for the model-checking tool himself. This component information has the additional advantage that it can be used as a specification for component implementation, if none is available already, once the target platform is chosen.

Property	Value
name	ConvEncoder
model of computation	SDF
number of input ports	1
input 1 data type	bit
input 1 consumption rate	1
number of output ports	1
output 1 data type	bit
output 1 production rate	2
number of parameters	1
parameter 1 name	constraintLength
parameter 1 type	integer
parameter 1 allowed values	5, 7, 9

Table 7.1: Component information for the convolutional encoder. The encoder has a code rate of $\frac{1}{2}$ and supports code polynomials of constraint lengths 5, 7, and 9.

In this case study, it is assumed that the model-checking tool has a library of available components along with the required information. Therefore, in this step of the design, the radio model sketched is mapped to this library.

Transmitter

It turns out that, on the transmitter side, four components are not in the library, namely the video source, convolutional encoder, pulse shaper, and radio front-end. Browsing the library further, it is found that a UDP receiver component exists. This can be used as a video source, in connection with a video streaming application that can send its data to a UDP destination (almost any streaming application can). Although there is no direct implementation of a pulse shaper in the library, an upsampler component as well as a generic finite impulse response (FIR) filter component exist. A pulse shaper can be composed from these components, by first upsampling the data and then configuring the FIR filter with coefficients for a root raised cosine impulse response. The radio front-end component is replaced with the UsrcTx component, a library component for controlling the Universal Software Radio Peripheral (USRP) [65] radio front-end. No substitutions for the convolutional coder could be found in the library, hence, the information about this component needs to be provided by the user. This will be used as a specification for implementing this component later in the development process. The component information about the convolutional coder is shown in Table 7.1.

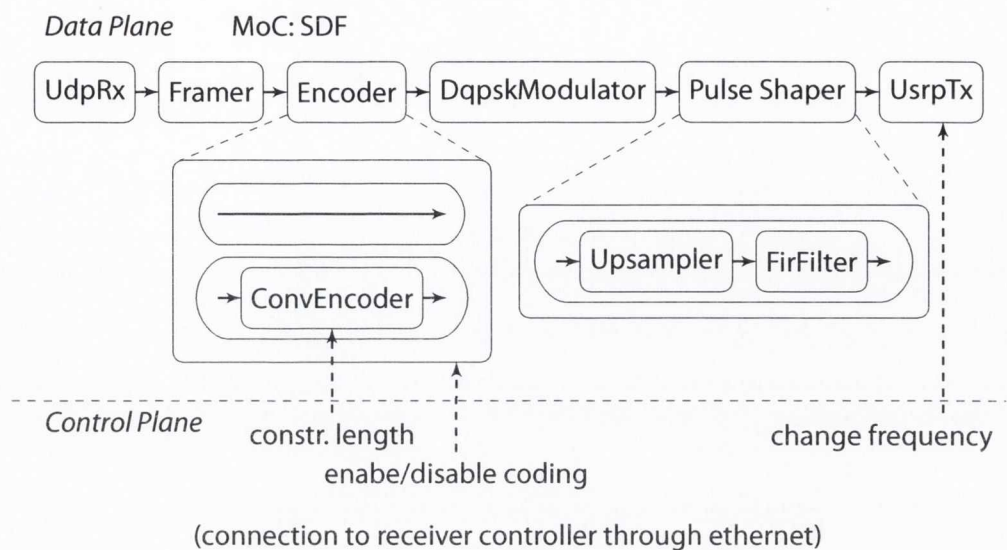


Figure 7.5: The radio model of the transmitter, mapped to the component library. The coding subsystem can have two modes: coding enabled and coding disabled. Further, when coding is enabled, the constraint length is a reconfigurable parameter.

Figure 7.5 shows the resulting radio model after mapping the sketched model to the components available in the library.

Receiver

On the receiver side, the matched filter, synchronisation, Viterbi decoder, and video sink are not available in the component library. Similar to the pulse shaper at the transmitter, the matched filter can be implemented by a generic FIR filter component, initialised with the coefficients of a root raised cosine filter matching the pulse shaper in the transmitter. The library provides two components for synchronisation, a Costas loop phase-locked loop (PLL) for carrier recovery [72, 126], and a modified Mueller and Muller timing recovery component [74] (named CostasLoop and MMTimingRecovery, respectively). Thus, the synchronisation component is replaced with a subsystem containing these two components in a chain. Similar to the video source at the transmitter, the video sink can be substituted with a UDP transmitter component, existent in the library, that can send the video streaming data to a video player via UDP. The player can then display the video. The Viterbi decoder does not exist and suitable substitutions are found in the library. Therefore, the component information has to be provided by the developer. It will be used as a specification for an implementation of that component, once the target platform is decided. Table 7.2 shows the component information for the Viterbi decoder.

Property	Value
name	ViterbiDecoder
model of computation	SDF
number of input ports	1
input 1 data type	bit
input 1 consumption rate	2
number of output ports	1
output 1 data type	bit
output 1 production rate	1
number of parameters	1
parameter 1 name	constraintLength
parameter 1 type	integer
parameter 1 allowed values	5, 7, 9
number of events	1
event 1 name	BER
event 1 type	float

Table 7.2: Component information for the Viterbi decoder. The decoder has a code rate of $\frac{1}{2}$ and supports code polynomials of constraint lengths 5, 7, and 9, matching those of convolutional encoder at the transmitter. Further, the Viterbi decoder needs to estimate the current BER and report this in an event.

The revised radio model, containing only components available in the library, is illustrated in Figure 7.6.

Having mapped the radio model to the component library, the developer is now ready to proceed to the next step in the design flow.

7.2.3 Write Control Specification

In step 3 of the design flow (see Figure 7.1), the developer writes the control specification. This contains information about the parameters to be reconfigured along with the values they can take (the knobs of the radio, used during the *act* stage of the cognitive cycle). This information is essential for model-checking, the next step in the design, to allow checking all configuration possibilities for errors. Additionally, it is valuable for automatic synthesis tools for generating a real implementation from the model.

Besides changing parameters, switching the active mode in a subsystem is another way to reconfigure the radio (another set of knobs). This does not need to be stated explicitly in the

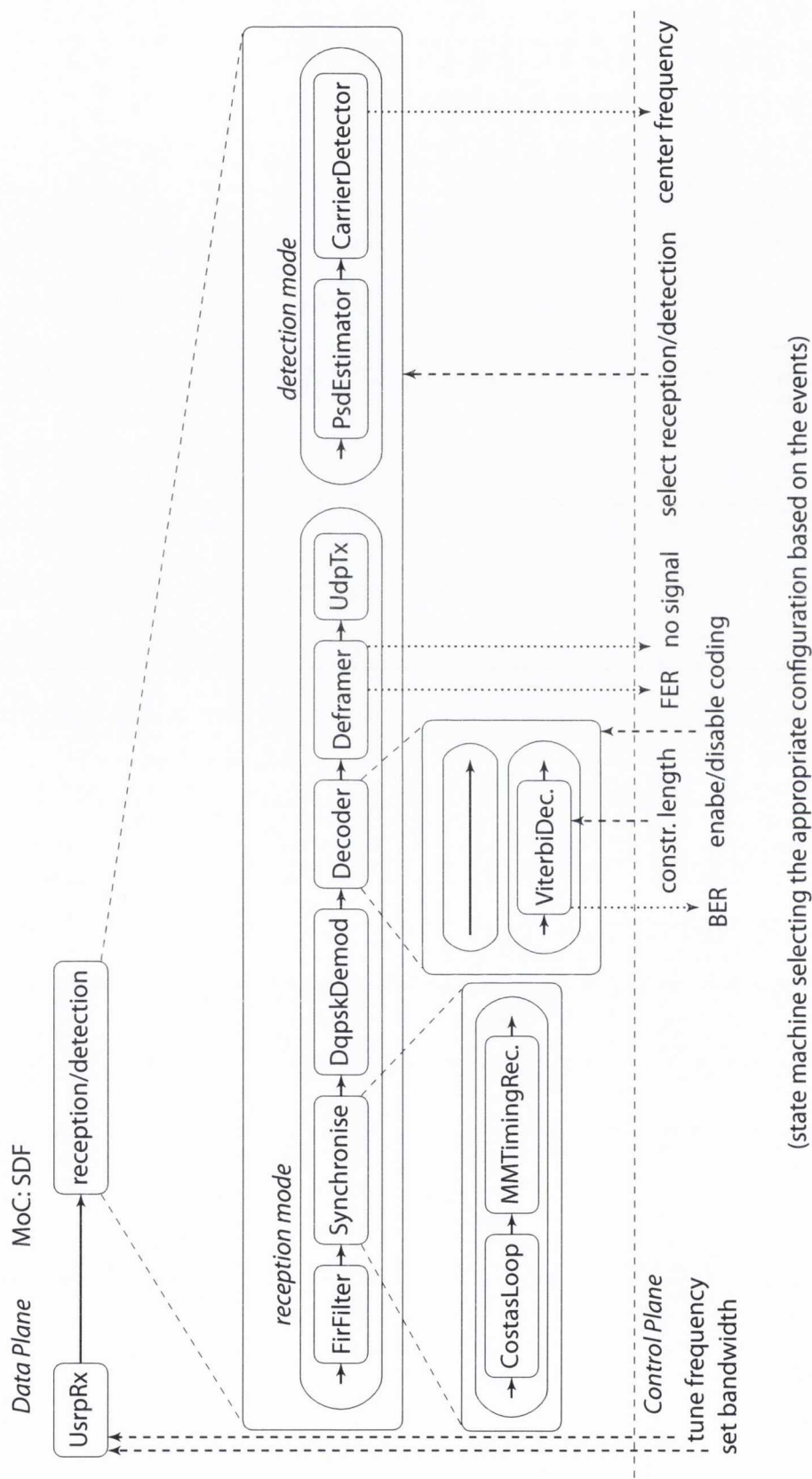


Figure 7.6: The radio model of the receiver, mapped to the component library. The radio can be either in reception or in detection mode. When in reception mode, the coding subsystem can be in one of two modes: coding enabled and coding disabled. Further, when coding is enabled, the constraint length is a reconfigurable parameter.

Component	Parameter	Value Set
ConvEncoder	constraintLength	{5, 7, 9}
UsrcTx	centreFrequency	[2, 493.7, 2, 496.3] MHz

Table 7.3: Control specification for the transmitter radio.

control specification, as the radio model data plane already covers this and model-checking assumes that the active modes can be changed at any time.

For model synthesis, knowing the events required by the control plane is essential (the meters of the radio, used during the *observe* stage). Therefore a list of required events is also part of the control specification.

Transmitter

On the transmitter side, the parameters that may be reconfigured are the constraint length of the convolutional encoder component and the transmission frequency of the radio front-end. In this example, the allowed constraint lengths of the code polynomials are 5, 7, and 9. The allowed centre frequencies in the radio front-end are chosen to be between 2,493.7 MHz and 2,496.3 MHz, a band of 2.6 MHz. This range was selected to be in the upper end of 2.4 GHz industrial, scientific, and medical (ISM) band, so it does not interfere with IEEE 802.11 [6] signals and other services. Since the signal bandwidth is about 500 KHz, there is enough space to change the carrier frequency in the selected band and prove the concept of the case study. Note that it is possible to change this band at a later stage without problems since most of the remaining design development steps are automatic.

The control plane at the transmitter side does not need to subscribe to any events. Thus, this part of the control specification can be omitted.

The control specification for the transmitter is given in Table 7.3.

Receiver

The receiver radio has more options for reconfiguration. However, most of these are covered by the modes in the subsystems, i.e., switching between reception and detection and enabling or disabling coding.

The parameters that need to be reconfigurable are similar as for the transmitter, i.e., the constraint length of the codes used by the Viterbi decoder and the centre frequency of the

Component	Parameter	Value Set
ViterbiDecoder	constraintLength	{5, 7, 9}
UsrcRx	centreFrequency	[2, 493.7, 2, 496.3] MHz
UsrcRx	bandwidth	{2, 8} MHz

Component	Event	Event type
ViterbiDecoder	BER	float
Deframer	FER	float
Deframer	noSignal	boolean
CarrierDetector	centreFrequency	integer

Table 7.4: Control specification for the receiver radio. The bandwidth of 2 MHz is used during reception, and 4 MHz during detection.

radio front-end. Additionally, the bandwidth of the radio front-end needs to be reconfigurable, to allow scanning a wider bandwidth at once during detection.

The events required by the control plane, as annotated in the radio model in Figure 7.6, are included in the control specification, given in Table 7.4.

At this point in the design, the radio model is fully specified and ready to be checked for errors.

7.2.4 Run Model-checking

The aim of this step in the design flow (step 4) is to identify errors in the construction of the radio model, allowing the radio designer to correct these errors before the implementation phase commences. This potentially saves the developer significant time, since detection and correcting errors at a later stage usually takes more effort.

A series of checks are performed on the radio model, designed in a way that allows an automated model-checking tool to carry them out. The radio developer does not need to concern himself with the details. If such tool does not exist, the radio developer can perform the checks manually. In the following, the checks outlined in Section 6.1.4 are carried for the example radio system of this case study and the results are presented.

Component parameter consistency

The first test determines whether all parameters given in the model exist within the components and whether the values given are within range. This test is independent of the applied MoC.

Component	Parameter	Type	Allowed Values	Value	Ok?
UdpRx	port	integer	[1, 65535]	1234	✓
ConvEncoder	constraintLength	integer	{5, 7, 9}	{5, 7, 9}	✓
Upsampler	factor	integer	[2, ∞)	4	✓
FirFilter	coefficients	float[]	(−∞, ∞) per element	rrc coeffs	✓
	numTaps	integer	[1, ∞)	25	✓
UsrpTx	bandwidth	integer	{0.5e6, 1e6, 2e6, 4e6, 8e6}	2e6	✓
	centreFrequency	integer	[2.4e9, 2.5e9]	[2.4937e9, 2.4963e9]	✓
	gain	float	(0, ∞)	0.3	✓

Table 7.5: Component parameters and checks for the transmitter chain. Components not shown in the table do not have parameters. The *e*-notation is a shorthand for $\cdot 10^x$, e.g., $2e6 = 2 \cdot 10^6$.

For the transmitter, all parameters, their values, allowed range, and the result of the parameter consistency checks are shown in Table 7.5. The equivalent table for the receiver side is Table 7.6. It can be seen that for both radio models, the parameters given are existent and their values are within range.

It is easy to imagine that wrong parameter names or values out of range is a common error. The radio designer might have typo in the parameters he specified, or some values might be out of range. For example, the bandwidth of the `UsrpRx` component might be set to 5 MHz during detection, although this is not supported by the front-end. This has an immediate effect on the parameters of the `PsdEstimator` and `CarrierDetector`. This check detects these errors and the radio designer can correct them before proceeding with the implementation.

Dynamically reconfigurable parameters

The control specification lists the parameters that may be reconfigured dynamically (see Section 7.2.3). The second test ensures that dynamic reconfiguration of these is supported by the associated components (some parameters are static and cannot be changed during execution).

For the transmitter, the parameters to be configured dynamically are the `centreFrequency` parameter of the `UsrpTx` component and the `constraintLength` parameter of the `ConvEncoder`. For the former, checking the component information library reveals that run-time reconfiguration is allowed. The `ConvEncoder` component is not yet implemented, thus it cannot be checked at this point whether this parameter is run-time reconfigurable. If it is, no problems arise. If not, it is still possible to build the requested radio, by separating the component into

Component	Parameter	Type	Allowed Values	Value	Ok?
UsrcRx	bandwidth	integer	$\{0.5e6, 1e6, 2e6, 4e6, 8e6\}$	$\{2e6, 4e6\}$	✓
	centreFrequency	integer	$[2.4e9, 2.5e9]$	$[2.4937e9, 2.4963e9]$	✓
FirFilter	coefficients	float[]	$(-\infty, \infty)$ per element	rrc coeffs	✓
	numTaps	integer	$[1, \infty)$	25	✓
CostasLoop	gain1	float	$(0, 1]$	2^{-4}	✓
	gain2	float	$(0, 1]$	2^{-10}	✓
MMTimingRecov.	gain1	float	$(0, 1]$	0.01	✓
	gain2	float	$(0, 1]$	0.000025	✓
ViterbiDecoder	constraintLength	integer	$\{5, 7, 9\}$	$\{5, 7, 9\}$	✓
UdpTx	address	string	N/A	videorxpc	✓
	port	integer	$[1, 65535]$	2222	✓
PsdEstimator	fftSize	integer	$\{2^n n \in \mathbb{N}\}$	2048	✓
	avgWindows	integer	$[1, 65536]$	512	✓
CarrierDetector	psdSize	integer	$\{2^n n \in \mathbb{N}\}$	2048	✓
	signalWidth	integer	$[1, \text{psdSize}]$	256	✓
	thresholdFactor	float	$(0, 10]$	1.2	✓

Table 7.6: Component parameters and checks for the receiver chain. Components not shown in the table do not have parameters. The e -notation is a shorthand for $\cdot 10^x$, e.g., $2e6 = 2 \cdot 10^6$. The gain parameters of the synchronisation components are the loop filter gains of the PLLs.

three different modes in the encoding subsystem and fixing the constraint length parameter in each of them. That is, the second model transformation rule from Section 6.1.5 on page 158 is applied. This has to be decided during model synthesis (Section 7.3.3), once the implementation of the ConvEncoder is available. For now, the check for the transmitter completes successfully.

In the receiver radio, the parameters to be reconfigured at run-time are the centreFrequency and bandwidth parameters of the UsrcRx component and the constraintLength parameter of the ViterbiDecoder, as seen from the control specification. Similar to the transmitter, the frequency and bandwidth parameters support dynamic reconfiguration and the constraint length parameter cannot be tested at this point since the ViterbiDecoder is not implemented yet. Thus, this check completes successfully for the receiver side as well.

Link data type consistency

The next test ensures that the data types on all links in the flow graph can be determined, and that they are supported by the components linked. For this, the component information library

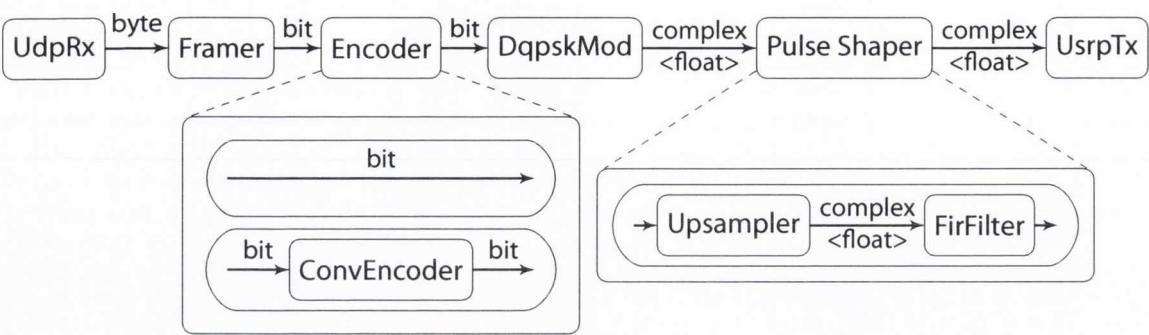


Figure 7.7: Transmitter data plane with annotated link data types.

is consulted, checking the input and output data types for all components. Figure 7.7 shows the data plane of the transmitter model with the determined link data types annotated. Figure 7.8 does likewise for the receiver. It can be seen that the data types can be determined and are consistent. Further checking the component library, it is found that the shown link data types are supported by all components. Thus, this test completes successfully.

Suppose the Framer component on the transmitter side would output bytes, instead of bits. However, the ConvEncoder expects bits on its input. This error would be detected by this check, identifying the critical link to the developer. It could then be corrected by placing a Bytes2Bits conversion component in between, converting the types as required.

Compatibility of MoCs

Next, it has to be confirmed that the MoCs applied in all levels in the model hierarchy are compatible, as defined in Section 5.2.7 on page 135. In other words, specialised MoCs cannot contain more general ones but the opposite is possible. For instance, when the highest hierarchy level applies the cyclo-static dataflow (CSDF) MoC, it is permitted to contain subsystems of applying the SDF MoC, but not the process network (PN) MoC. Since all hierarchy levels in the radio models for the transmitter and receiver are applying the SDF MoC, this test is passed.

Special checks for models using compile-time schedulable MoCs

Up to this point in the model checking, the checks are generic and can be performed on any cognitive radio model. If the model employs a compile-time schedulable MoC anywhere in the hierarchy, e.g., SDF, more thorough checks can be performed without depending on the component implementation. A general description of these checks is given in Section 6.1.4 on page 151 and a description of compile-time schedulable MoCs in Section 5.2.7 on page 135.

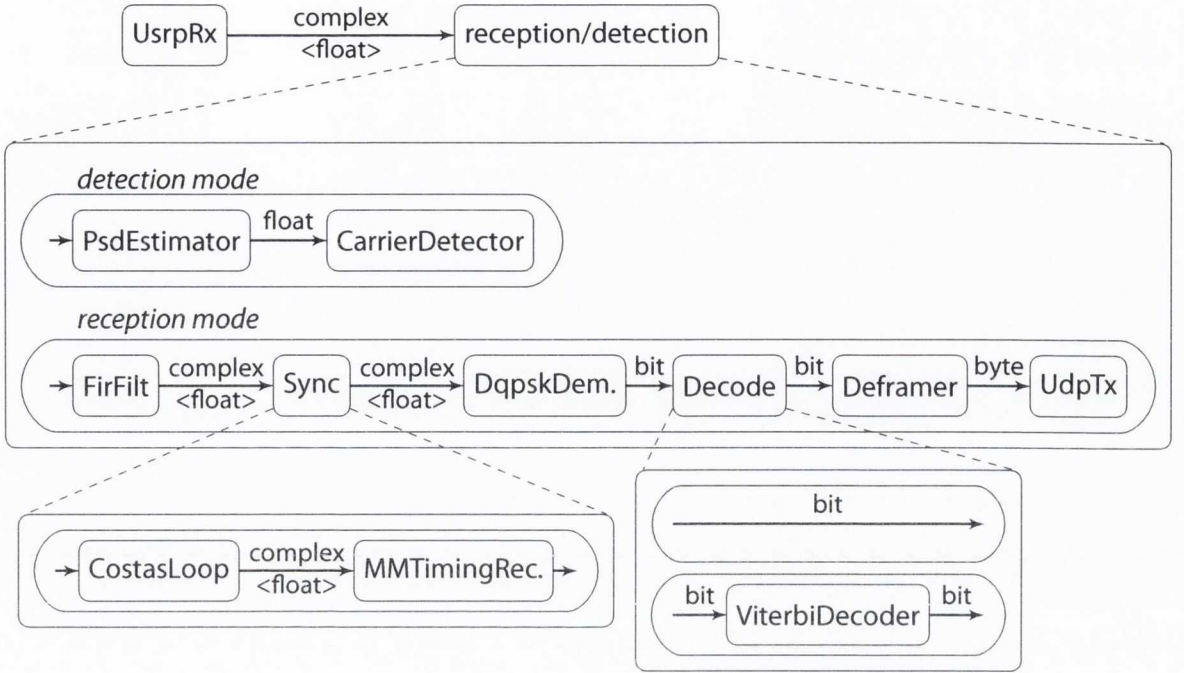


Figure 7.8: Receiver data plane with annotated link data types.

Since the radio models of the transmitter and receiver for this case study apply the SDF MoC, it is possible to perform these checks. They are detailed in the following paragraphs.

Following the outline from Section 6.1.4, the sets S_n , containing all modes in a subsystem n of a given model, and the sets P_m , containing the reconfiguration values for parameter m have to be defined.

Two subsystems exist on the transmitter side, indexed as $n = 0$ for the encoder subsystem and $n = 1$ for the pulse shaper subsystem (see Figure 7.5). The corresponding sets S_n are

$$S_0 = \{s_{0,0}, s_{0,1}\} \text{ and} \quad (7.1)$$

$$S_1 = \{s_{1,0}\}, \quad (7.2)$$

where $s_{0,0}$ and $s_{0,1}$ are the no-coding and coding modes of the encoding subsystem, respectively, and $s_{1,0}$ represents the single mode in the pulse shaper subsystem. According to the control specification, two sets of desired reconfiguration values P_m exist for the transmitter: the frequency parameter of the UsrpTx component, denoted with $m = 0$, and the constraint length

parameter of the ConvEncoder component ($m = 1$). Thus,

$$P_0 = \{f \in \mathbb{Z} \mid 2.4937 \cdot 10^9 \leq f \leq 2.4963 \cdot 10^9\} \text{ and} \quad (7.3)$$

$$P_1 = \{5, 7, 9\}. \quad (7.4)$$

Note that P_0 is a finite set.

Now the global state space of the transmitter radio can be computed as the Cartesian product of all S_n and P_m in the model, i.e.,

$$\begin{aligned} G &= \prod_{n=0}^1 S_n \times \prod_{m=0}^1 P_m \\ &= S_0 \times S_1 \times P_0 \times P_1 \\ &= \{(s_{0,0}, s_{1,0}, 2.4937 \cdot 10^9, 5), \dots, (s_{0,1}, s_{1,0}, 2.4963 \cdot 10^9, 9)\}. \end{aligned} \quad (7.5)$$

The size of this state space is

$$\begin{aligned} |G| &= |S_0| \cdot |S_1| \cdot |P_0| \cdot |P_1| \\ &= 2 \cdot 1 \cdot 2600001 \cdot 3 \\ &= 15600006 \\ &\approx 15.6 \cdot 10^6 \end{aligned} \quad (7.6)$$

For each of the states in G , the radio model can be flattened into a single dataflow graph without hierarchy and with fixed parameters in all components. This allows applying the well-known analysis methods for the compile-time schedulable MoCs to each of these flattened flow graphs, to find graph construction errors.

However, due to the large state space, this requires checking 15.6 million states for this simple transmitter. Fortunately, some state space reduction techniques can be applied, as listed in Section 6.1.4 on page 155.

First, only subsystems using a compile-time schedulable MoC need to be considered. In this example, the whole graph uses SDF, which is a compile-time schedulable MoC, so the state space cannot be reduced by applying the rule.

Second, all parameters that do not affect the input and output data rates of a component can be eliminated. In the example, neither the frequency parameter nor the constraint length parameter affect the sample consumption/production rates on the corresponding components,

thus, those can be eliminated from the state space. This reduces the size of G considerably.

According to the third reduction rule, if some modes combinations in subsystems in the hierarchy are never active at the same time, they can be eliminated. In the transmitter, all mode combinations are valid, so a reduction based on this rule is not possible.

The fourth reduction rule states that modes with the same external input/output sampling rates within one subsection can be reduced to one mode for the model checking. In the radio model for the transmitter, the modes in the encoder subsystem have different data rates. When the ConvEncoder is active, the output data rate doubles compared to no coding (due to the rate $\frac{1}{2}$ coder). The other subsystem only has one mode. Thus, no reductions can be performed based on this rule.

The fifth reduction rule is for subsystems where the external input/output data rates are multiples of each other, which is clearly not fulfilled for this example.

And finally, some parameters might be mutual exclusive and therefore impossible combinations can be removed. But the reconfigurable parameters for this examples have already been removed, thus, this rule cannot be applied.

After applying these reduction rules, the new state space is:

$$\begin{aligned}\check{G} &= S_0 \times S_1 \\ &= \{(s_{0,0}, s_{1,0}), (s_{0,1}, s_{1,0})\},\end{aligned}\tag{7.7}$$

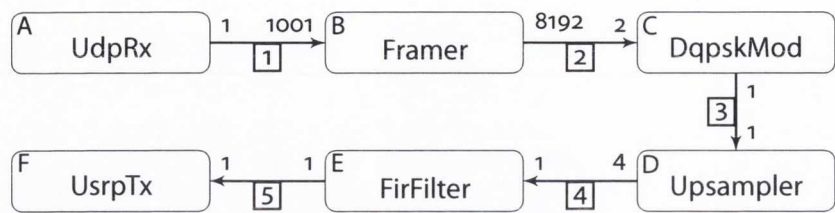
where

$$|\check{G}| = 2.\tag{7.8}$$

Now the SDF consistency checks only need to be applied to two different flattened flow graphs. These are shown in Figure 7.9, annotated with the SDF sampling rates at all ports.

The algorithm for SDF consistency checks have been detailed in Section 5.2.2 on page 122. First, the topology matrix T is defined, with its columns assigned to components and rows assigned to links. The (i, j) -th entry in the matrix represents the amount of data produced by component j on link i per component execution. The topology matrix for the graphs in the

coding disabled, state: $(s_{0,0}, s_{1,0})$



coding enabled, state: $(s_{0,1}, s_{1,0})$

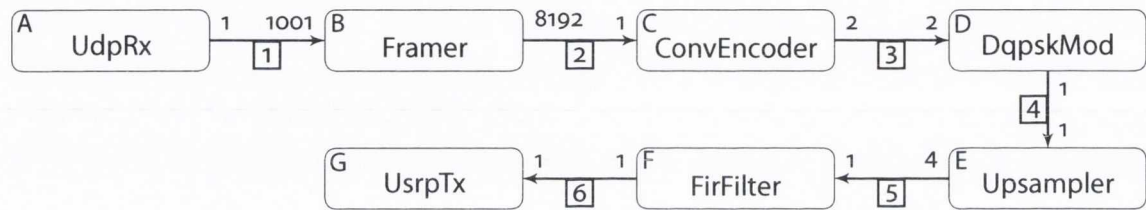


Figure 7.9: Flattened SDF flow graphs for the transmitter radio. The SDF sample rates at each component port, as well as component identifiers and link numbers are annotated in the graphs.

upper half of Figure 7.9 (coding disabled) is

$$\mathbf{I}_0 = \begin{bmatrix} 1 & -1001 & 0 & 0 & 0 & 0 \\ 0 & 8192 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \tag{7.9}$$

and the topology matrix when coding is enabled is

$$\mathbf{I}_1 = \begin{bmatrix} 1 & -1001 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8192 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}. \tag{7.10}$$

The first condition for a valid SDF graph is that its topology matrix is of a rank one less than the number of components. In our example, $\text{rank } \Gamma_0 = 5$, and $\text{rank } \Gamma_1 = 6$, i.e., this condition is fulfilled.

The repetition vector $\mathbf{q}$, indicating how often each component must be executed during one scheduling cycle, is in the right nullspace of Γ . It is defined as a strictly positive integer vector that solves the equation

$$\Gamma \mathbf{q} = \mathbf{0}. \quad (7.11)$$

In the graphs for the example transmitter radio, the resulting repetition vectors are

$$\mathbf{q}_0 = J_0 \begin{bmatrix} 1001 & 1 & 4096 & 4096 & 16384 & 16384 \end{bmatrix}^T \quad (7.12)$$

$$\mathbf{q}_1 = J_1 \begin{bmatrix} 1001 & 1 & 8192 & 8192 & 8192 & 32768 & 32768 \end{bmatrix}^T \quad (7.13)$$

where $J_0, J_1 \in \mathbb{N}$.

A sufficient condition that an SDF graph is correctly constructed is to construct a periodic schedule, i.e., a schedule which returns the number data items stored on the links to their original state after executing each component the number of times indicated by $\mathbf{q}$. In [35] it is shown that class S scheduling algorithms for SDF always succeed if a periodic schedule exists.

To test for the existence of a periodic schedule for the example radio graphs, the simple algorithm outlined in Section 5.2.2 on page 122 has been applied. Example execution schedules for both graphs are

$$\Psi_0 = \{1001A, B, 4096 \{C, D, 4E, 4F\}\} \quad (7.14)$$

$$\Psi_1 = \{1001A, B, 2 \{4096 \{C, D, E\}, 16384 \{F, G\}\}\} \quad (7.15)$$

where multiplicity and bracketing is used to save space. For instance, the expression $\{3 \{A, B\}\}$ is equivalent to $\{A, B, A, B, A, B\}$. These schedules exist and are valid, thus, the graphs for the two states of the radio model in the transmitter are consistent and correctly constructed.

Now, the receiver side is checked for consistent construction in a similar fashion. Three subsystems exist at the receiver, indexed $n = 0$ for the reception/detection subsystem, $n = 1$ for the synchronisation subsystem, and $n = 2$ for the decoder subsystem. The corresponding sets

of modes are

$$S_0 = \{s_{0,0}, s_{0,1}\} \quad (7.16)$$

$$S_1 = \{s_{1,0}\} \quad (7.17)$$

$$S_2 = \{s_{2,0}, s_{2,1}\} \quad (7.18)$$

where $s_{0,0}$ denotes the reception mode and $s_{0,1}$ the detection mode. In the decoding subsystem, $s_{2,0}$ denotes disabled coding, and $s_{2,1}$ enabled coding.

The reconfigurable parameters are the frequency and bandwidth parameters in the `UsrpRx` component and the constraint length parameter in the `ViterbiDecoder`. The corresponding parameter sets are

$$P_0 = \{f \in \mathbb{Z} \mid 2.4937 \cdot 10^9 \leq f \leq 2.4963 \cdot 10^9\} \text{ and} \quad (7.19)$$

$$P_1 = \{2 \cdot 10^6, 8 \cdot 10^6\} \quad (7.20)$$

$$P_2 = \{5, 7, 9\}. \quad (7.21)$$

Now, the global state space of the receiver model can be constructed. It is the Cartesian product of all subsystem and parameter sets, i.e.,

$$\begin{aligned} G &= S_0 \times S_1 \times S_2 \times P_0 \times P_1 \times P_2 \\ &= \{(s_{0,0}, s_{1,0}, s_{2,0}, 2.4937 \cdot 10^9, 2 \cdot 10^6, 5), \dots, (s_{0,1}, s_{1,0}, s_{2,1}, 2.4963 \cdot 10^9, 8 \cdot 10^6, 9)\} \end{aligned} \quad (7.22)$$

The size of this state space is

$$\begin{aligned} |G| &= |S_0| |S_1| |S_2| |P_0| |P_1| |P_2| \\ &= 2 \cdot 1 \cdot 2 \cdot 2600001 \cdot 2 \cdot 3 \\ &= 62400024 \\ &\approx 62.4 \cdot 10^6 \end{aligned} \quad (7.23)$$

The well-known consistency checks for the SDF MoC could now be applied to a flattened dataflow graph for each of these states. However, the number of states can be drastically reduced, using the rules given in Section 6.1.4.

Similar to the transmitter, the constraint length and frequency parameters have no effect on the input and output data sizes of their associated components. The bandwidth does affect

the sample rate (in time), but not the number of output items of the `UsrpRx`. That is, it has no effect on the SDF properties of the `UsrpRx` component and can therefore be removed from the state space as well. Neglecting these parameters during model-checking reduces the state space considerably. Furthermore, since the reception and detection modes are mutually exclusive, some state space tuples are irrelevant and can be removed. For instance, when the detection mode is active, it does not matter what is the active mode of the subsystems contained within the reception mode. This is the application of the state reduction rule three, as stated in Section 6.1.4 on page 156.

After applying all reduction rules, the state space becomes

$$\begin{aligned} \check{G} = \{ & (s_{0,0}, s_{1,0}, s_{2,0}), & & \text{(unencoded reception)} \\ & (s_{0,0}, s_{1,0}, s_{2,1}), & & \text{(encoded reception)} \\ & (s_{0,1}, \times, \times) \} & & \text{(detection),} \end{aligned} \quad (7.24)$$

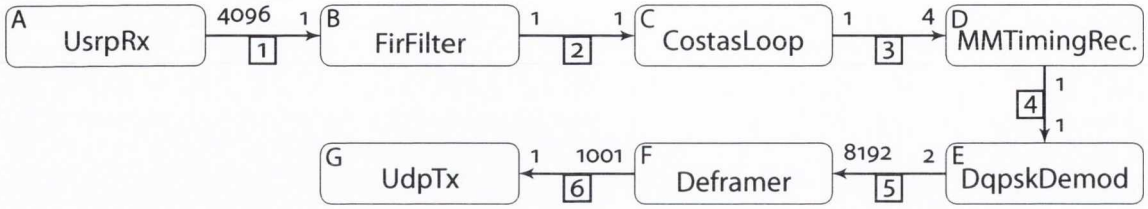
where the symbol $\times$ indicates an irrelevant tuple member. Now $|\check{G}| = 3$, thus, three different global states of the receiver radio model are relevant for SDF model-checking.

For each state in $\check{G}$, the radio model can be flattened into a traditional SDF dataflow graph upon which well-known consistency checks can be applied. These graphs for the receiver in the case study system are shown in Figure 7.10.

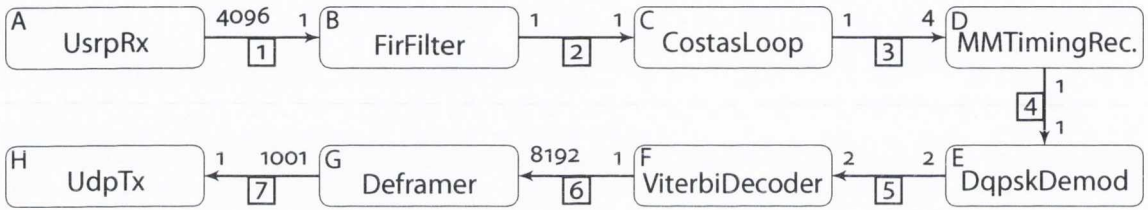
The topology matrices for all three states of the receiver model are

$$\mathbf{F}_0 = \begin{bmatrix} 4096 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -8192 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1001 & -1 \end{bmatrix} \quad (7.25)$$

reception, coding disabled, state: $(s_{0,0}, s_{1,0}, s_{2,0})$



reception, coding enabled, state: $(s_{0,0}, s_{1,0}, s_{2,1})$



detecton, state: $(s_{0,1}, \times, \times)$

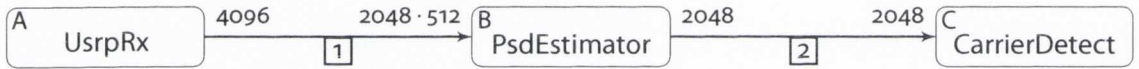


Figure 7.10: Flattened SDF flow graphs for the receiver radio. The SDF sample rates at each component port, as well as component identifiers and link numbers are annotated in the graphs. The PsdEstimator consumes 512 FFT windows of size 2048, to produce one averaged PSD estimate, hence the data rate on its input.

$$\mathbf{I}_1 = \begin{bmatrix} 4096 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -8192 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1001 & -1 \end{bmatrix} \quad (7.26)$$

$$\mathbf{I}_2 = \begin{bmatrix} 4096 & -1048576 & 0 \\ 0 & 2048 & -2048 \end{bmatrix} \quad (7.27)$$

A necessary condition for a consistent SDF flow graph is that the rank of the topology matrix

is one less than the number of components, i.e., the number of columns in the matrix. The topology matrices for all three model states fulfil this condition, i.e., $\text{rank } \mathbf{F}_0 = 6$, $\text{rank } \mathbf{F}_1 = 7$, $\text{rank } \mathbf{F}_2 = 2$.

A sufficient condition for a correctly constructed SDF graph is the existence of a periodic execution schedule which brings the buffers on links back into the initial state. To compute a schedule, the repetition vectors $\mathbf{q}_i$ for each case are required. These are positive integer vectors in the right nullspace of $\mathbf{F}_i$, i.e.,

$$\mathbf{q}_0 = J_0 \begin{bmatrix} 4 & 16384 & 16384 & 4096 & 4096 & 1 & 1001 \end{bmatrix}^T \quad (7.28)$$

$$\mathbf{q}_1 = J_1 \begin{bmatrix} 8 & 32768 & 32768 & 8192 & 8192 & 8192 & 1 & 1001 \end{bmatrix}^T \quad (7.29)$$

$$\mathbf{q}_2 = J_2 \begin{bmatrix} 256 & 1 & 1 \end{bmatrix}^T \quad (7.30)$$

with $J_i \in \mathbb{N}$. Now periodic execution schedules can be constructed. Example schedules for the given graphs are

$$\Psi_0 = \{4 \{A, 1024 \{4 \{B, C\}, D, E\}\}, F, 1001G\} \quad (7.31)$$

$$\Psi_1 = \{2 \{4A, 4096 \{4 \{B, C\}, D, E, F\}\}, G, 1001H\} \quad (7.32)$$

$$\Psi_2 = \{256A, B, C\} \quad (7.33)$$

where multiplicity and bracketing is to shorten the expression. For instance, the expression $\{3 \{2A, B\}\}$ is equivalent to $\{A, A, B, A, A, B, A, A, B\}$. All these schedules return the graph into its original state, i.e., with the same amount of data stored in each link. Therefore, the graphs for the receiver are correctly constructed and consistent.

For illustration purposes, an example SDF graph is constructed where the graph consistency checks fail. Suppose the BER estimation in the Viterbi decoder would not be computed internally, but through separate components. It can be estimated by re-encoding the decoded data and comparing with the input stream, as shown in the partial graph in Figure 7.11. The

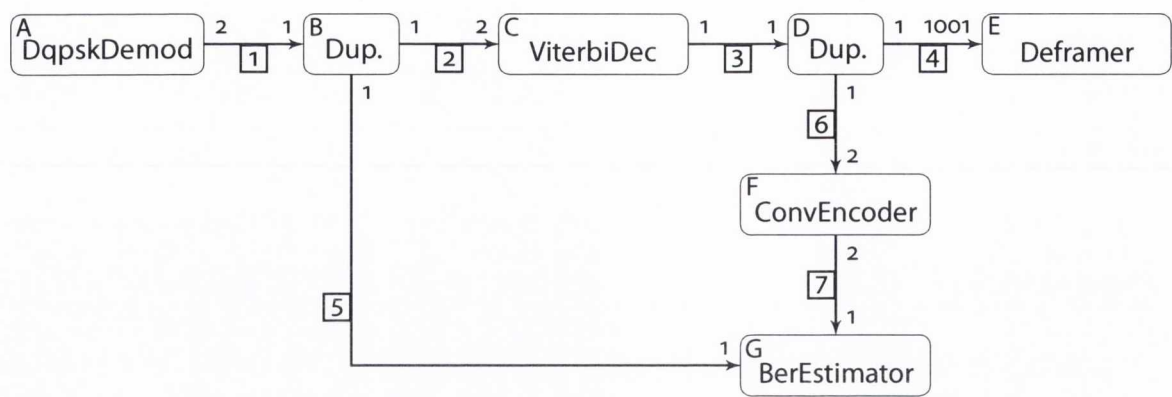


Figure 7.11: Viterbi decoder with BER estimation (erroneous SDF graph). The Dup. component duplicates the input data on both output ports, the BerEstimator computes the BER and reports this to the controller in an event. The graph contains a sampling rate inconsistency.

topology matrix for the SDF graph is

$$\mathbf{I} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1001 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 2 & -1 \end{bmatrix}. \tag{7.34}$$

This matrix has full rank (rank 7), i.e., the necessary condition for a consistent SDF graph is not fulfilled. This means that some sample rate in the graph is not correct or that the graph is connected in a wrong way. Examining the problem closer, the radio designer finds that his specification of the ConvEncoder is wrong. It should consume 1 bit and produce 2. Correcting the input sample rate in the graph, and the corresponding entry in the topology matrix (setting the entry in row 6 and column 6 to) fixes the problem. The corrected topology matrix has rank 6, as required for the check. The radio designer can fix the component specification for the ConvEncoder with the corrected sampling rates at this point in the design. Without this check, the component would be implemented with the wrong data rates, and the error would need to be identified through time consuming debugging methods. This clearly shows the benefit of the graph consistency checks in the implementation-independent design phase.

Model-checking summary

All model-checks were successful for both the transmitter and receiver radios. Thus, the radio models are constructed without errors. If that was not the case, the radio designer would need to fix the error and run the model checks again, until all test complete successfully (as shown in Figure 7.1).

Although the checks outlined can be performed automatically, they can also be carried out manually if no such automatic tool is available. The effort of these checks is relatively small, yet, it would allow to detect model construction errors at an early stage in the development, potentially saving considerable amounts of time.

Note that the checks performed here do not confirm that the radios are functionally correct, i.e., whether the radio does what it is supposed to do. For instance, it cannot be decided whether the output of the receiver radio will be a correct reconstruction of the transmitted video. However, the checks performed here confirm that the radio models are runnable and that the graphs are consistent. That is, data types, links, parameters, and reconfigurations are supported by the components and the radio models are executable in all configurations. Radio designers can fix model construction and consistency errors before choosing a target platform, which eliminates a large source of errors before implementation commences. This contributes to more reliable radios and shorter development times.

7.2.5 Summary

All radio design steps covered in this section are platform-independent. Radio designers can construct and test their designs without deciding on a target platform. The designs contains information about the cognitive radio to be built, including all run-time reconfiguration options, the meters to get information about the radio and the environment, and the knobs to adapt the radio's behaviour. Design consistency can be checked, allowing designers to fix problems before proceeding to the implementation stage.

Using the proposed model-based design technique, the design phase of a cognitive radio system is clearly separated from the implementation phase. Developers do not need concern themselves with implementation details during the design phase, in contrast to traditional radio design methods.

The design of the cognitive radio system used as a case study has passed all correctness tests, thus, the radio designer can be sure that the radio description for both the transmitter

and receiver is error-free in terms of the graph construction, parameters, and links. The radio models for the transmitter and receiver are now ready for implementation on a real platform.

7.3 Radio Implementation

At this point in the model-based design flow (see Figure 7.1), the radio designer has to choose the target implementation platform for the radio he has designed. For this case study, the target platform of choice is Iris2-FPGA (see Section 6.3), the modified version of the Iris-FPGA cognitive radio framework from Section 1.3. It supports the level of reconfigurability required, provides an implementation of the SDF MoC, and allows for integration FPGA hardware in radios, hence, it is ideally suited to the case study. Note that any other cognitive platform would be a valid choice for the target platform; general synthesis considerations for arbitrary platforms are outlined in Section 6.2.2.

The following sections walk through the platform-dependent stages in the design flow given in Figure 7.1 (steps 6–9). Details in every step are given for the chosen FPGA-based target platform running Iris2-FPGA.

7.3.1 Implement Missing Components

Two components of the radio design are not available in Iris2-FPGA, the ConvEncoder and ViterbiDecoder. Thus, the radio developer has to implement them (step 6 in Figure 7.1). The design component information used during model design can be used as specifications for the component implementation. Tables 7.1 and 7.2 (on pages 190 and 192) show this information.

The target platform allows a combination of software and hardware components. Thus, it has to be decided whether to implement the missing components in software or FPGA hardware. Decoding convolutionally encoded data with the Viterbi algorithm requires considerable computation effort. The Viterbi algorithm is heavily parallel, allowing for efficient implementation on FPGAs. The convolutional encoder is simpler, but nevertheless can be implemented efficiently on FPGAs. Furthermore, the library of intellectual property cores provided by Xilinx already contains a cores for convolutional coding and Viterbi decoding [79, 127]. Therefore, it was chosen to implement the ConvEncoder and ViterbiDecoder components in FPGA hardware.

The ViterbiDecoder core from the Xilinx library can be configured to estimate the current BER, through re-encoding the decoded data and comparing to the original input. This infor-

mation is required, since the Viterbi decoder needs to publish an event with the current BER estimate to the controller (as specified in the control specification in Table 7.4).

The cores from the Xilinx library were configured to the needs of the transmitter and receiver radio and wrapped with some simple VHDL code, in order to provide the standard interface for hardware components explained in Section 2.2.3 on page 50.

With implementations for all components available, the radio designer can now proceed to implement the control plane.

7.3.2 Implement Control Plane

The control plane implements the ‘brain’ of the radio, i.e., the decision-making process and, if required, machine learning. In the given example, the controllers on both ends configure the radio according to predefined conditions. Learning, optimisation, or other more advanced decision-making is not required. Implementing the control plane is step 7 of the design flow (Figure 7.1).

Transmitter

The control plane on the transmitter side performs two simple tasks. It configures the radio to use the coding scheme requested by the receiver and it changes the transmission frequency randomly in intervals to test the receiver’s detection capabilities. It does not require any information from the radio in execution, i.e., no events need to be subscribed to.

Using the Iris2-FPGA controller application programming interface (API), the required functionality of the transmitter control plane is implemented in two controller entities, both running in their own thread of execution (asynchronously with the radio).

One controller is responsible for switching the transmission frequency in regular intervals to a random frequency within the given band. This is simply implemented by suspending the thread for a period of time (e.g., 15 seconds), reconfiguring the transmission frequency of the `UsrcTx` component, and repeating the process in a loop.

The second controller is responsible for selecting the required coding scheme as directed from the receiver. Ideally, this information should be transmitted wirelessly from the receiver to the transmitter. However, for simplicity of this case study, Ethernet is used. The controller listens on a UDP socket on a predefined port for instructions from the receiver. When a message is

received, the content is evaluated and the radio is configured accordingly. Then the procedure starts again.

The message format is as follows. It is simply a single byte, interpreted as an unsigned number (i.e., it can take values from 0 to 255). Zero means that coding is to be disabled, while 5, 7, and 9 mean that coding is to be enabled, with a constraint length parameter value according to the message. All other messages are invalid, thus, an error is indicated.

This implements all functionality required in the control plane of the transmitter radio. The waiting for information from the receiver and for timers to pass can be mapped to the *observe* stage of the cognitive cycle. The *decide* stage is trivial, as all decisions are made by the receiver. During the *act* stage, the reconfigurations of the radio are affected as described above.

Receiver

The control plane of the receiver radio needs to implement the state machine shown in Figure 7.4 on page 189, with one state for each configuration of the data plane. State transitions are performed based on the events received from the radio's data plane.

Iris2-FPGA does not support state machine models for the controller directly, they have to be implemented as C++ code instead. Ideally, this code can be generated automatically from a model of the state machine. Although some tools for generating code from an FSM or UML statechart exist, there is not much benefit for simple state machines with only a few states. Therefore it was decided to implement the code for the receiver state machine manually.

A convenient library to simplify the implementation of state machines is the Boost Statechart Library [128]. It uses advanced features of the C++ programming language to simplify the developer's implementation effort. The functionality shown in Figure 7.4 has been implemented using this library, and the Iris2-FPGA controller API subscribes to the appropriate component events and posts them to the state machine when triggered. This is the *observe* stage of the cognitive cycle. The state machine itself performs the decision-making, i.e., it represents the *decide* stage. Reconfigurations are implemented as state entry actions, i.e., they are affected when the state machine enters one of its states. This represents the *act* stage of the cognitive cycle. Further, a UDP packet is sent to the transmitter whenever the coding scheme is changed, to ensure that transmitter and receiver apply the same coding scheme.

With both transmitter and receiver control plane implemented, the radio designer can proceed with the radio implementation.

7.3.3 Run Synthesis Tool

Synthesis is the generation of real implementations from the radio model description. This step can be performed by automatic tools, following the steps outlined in Section 6.2.3 on page 169. In the following, these steps are detailed for the example cognitive radio setup to be implemented in this chapter.

Check component availability

The first step is to check the availability of all components used in the model on the target platform. With the ConvEncoder and ViterbiDecoder components implemented (see Section 7.3.1), this check completes successfully.

Check MoC platform support

Second, the synthesis tool checks whether all used MoCs from the model are implementable on the target platform. The models for transmitter and receiver use the SDF MoC only, which is supported by Iris2-FPGA directly. Otherwise, they could be executed using a compatible MoC, such as PN, at the cost of less efficiency. For a discussion of compatible MoCs, see Section 5.2.7 on page 135. Thus, this check is passed.

Transform model to suit target platform

The third step is to transform the model into a form that suits the target platform, optimising for efficiency. This step takes more effort, and will be examined in more detail in the following.

Since the target platform is an embedded FPGA system running Iris2-FPGA, the processor (PowerPC 440) for executing the software part of the radio is not very powerful (compared to a desktop PC). On the other hand, components in FPGA hardware are extremely fast, typically multiple times faster than software implementations on a PC (see for example the benchmark data given in Section B.3 on page 254). Therefore, the aim of the transformations should be to use FPGA hardware as much as possible, especially for the components requiring high computation power.

Iris2-FPGA does not support hierarchical dataflow graphs. However, when hardware component chains are used, these can be put into a customisable processing subsystem (CPSS) of the hardware architecture and are represented to the rest of the system by a single software

wrapper component (see sections 2.2.2 and 2.2.3 for a description of these concepts). This is essentially a hierarchical component, representing a whole subgraph as a single component (i.e., a subsystem). Software subsystems can be converted into flat dataflow graphs using the switch and select components, as outlined in Section 6.2.2 on page 163 and illustrated in Figure 6.6.

Since the target FPGA platform provides a single CPSS, the aim of the transformations of the example model is to find the largest consecutive chain of hardware components and represent them in a single subsystem. The remainder of the graph is to be transformed into a non-hierarchical form using switch and select components as required.

On the transmitter side, the components available in hardware are the ConvEncoder, DqpskModulator, Upsampler, and FirFilter. Fortunately, these components are connected in a sequence in the radio model, i.e., they can be connected directly in hardware for more efficiency. Thus, the model needs to be transformed so that all these components are in a single subsystem (applying the transformation rules from Section 6.1.5 on page 158). Further, the ConvEncoder code polynomials, i.e., the constraint length parameter, cannot be changed during execution as this requires considerable changes in the hardware. According to the control specification, three different constraint lengths (5, 7, and 9) must be supported. Thus, for each parameter option, a new instance of the ConvEncoder is created and put into a separate mode in the subsystem. Therefore, the hardware subsystem includes four modes: no coding, and constraint lengths 5, 7, and 9.

The remainder of the transmitter model not hierarchical, thus no further transformations need to be applied. The transformed model is shown in Figure 7.12. The FPGA CPSS subsystem represents one CPSS of the FPGA. It is implemented as a software wrapper component with a parameter for selecting the active mode, i.e., the configuration of the FPGA. When this mode is changed, a reconfiguration of the FPGA is performed.

The transformed transmitter model is equivalent to the original model designed by the developer but suited for execution on the target platform.

For the receiver, the available hardware components are the FirFilter, CostasLoop, MMTimingRecovery, DqpskDemodulator, ViterbiDecoder, and the PsdEstimator components. All other components are available in software. As for the ConvEncoder at the transmitter, the different configurations of the ViterbiDecoder require considerable changes in the hardware. Therefore, the constraint length cannot be reconfigured at run-time. Instead, three different instances of the ViterbiDecoder are required, each configured with a code polynomial of a different constraint length. Information about the required configurations are taken from the control specification, i.e., constraint lengths of 5, 7, and 9 need to be supported. The hardware

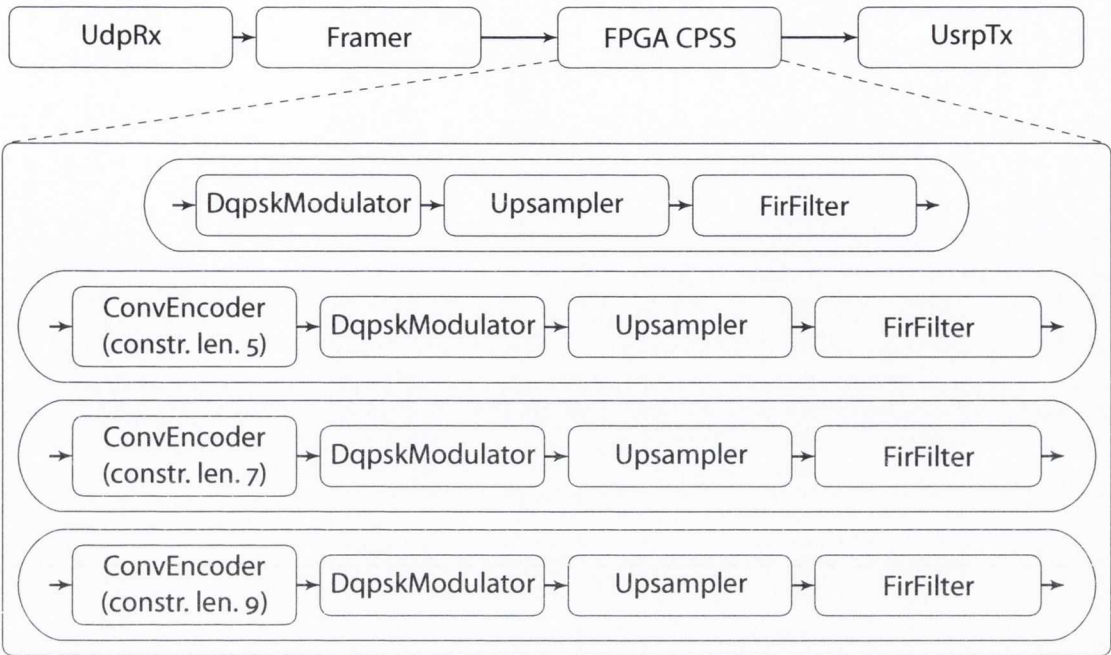


Figure 7.12: Transmitter model transformed to suit the target platform. The FPGA CPSS contains four modes, i.e., four possible configurations of the FPGA. It is represented to Iris2-FPGA as a single software wrapper component with a parameter to select the active mode (the FPGA CPSS is reconfigured when the mode changes).

components are chained together into a single subsystem, representing the CPSS of the target FPGA platform.

The remainder of the receiver model is implemented in software. Since hierarchical graphs are not supported in Iris2-FPGA, this part is transformed to a flat graph using switch and select components as required.

The resulting transformed model for the receiver is shown in Figure 7.13. It contains one subsystem only, representing the CPSS of the FPGA target platform. All hardware components are chained together to achieve higher efficiency in passing the data from one component to the next. The CPSS is represented to Iris2-FPGA by a software wrapper component which exposes a parameter for selecting the active mode, i.e., FPGA configuration. The hardware is reconfigured dynamically when this parameter is changed. The switch is inserted to select the data path between the detection and reception modes in software.

The resulting transformed model is now suited for implementation on the target platform. It has changed considerably compared to the original radio model for the receiver, however, it is functionally equivalent.

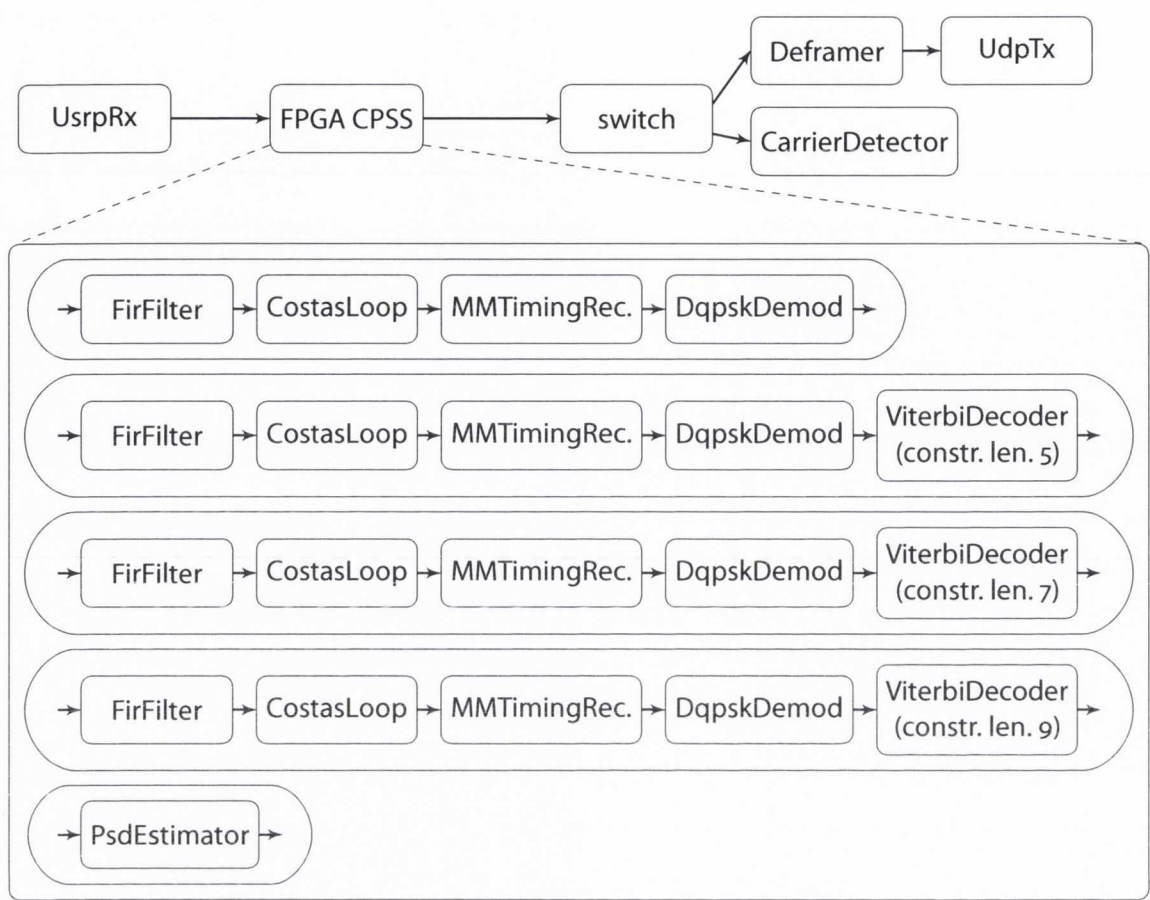


Figure 7.13: Receiver model transformed to suit the target platform. The FPGA CPSS contains five modes i.e., five possible configurations of the FPGA. It is represented to Iris2-FPGA as a single software wrapper component with a parameter to select the active mode (the FPGA CPSS is reconfigured when the mode changes).

Generate software wrapper component(s)

A software wrapper component represents one CPSS of the FPGA to the Iris2-FPGA radio framework and encapsulates all interfacing to the hardware. This concept allows for the integration of hardware without modifying any other part of the Iris2-FPGA framework. It can therefore be used in a similar fashion for other cognitive radio frameworks as well.

In the example radios, the transmitter and receiver platforms contain one CPSS each. The source code for the software wrapper components can be generated automatically, as all required information is available to the synthesis tool. It is a relatively simple component, exposing one parameter for selecting the active configuration for the FPGA (i.e., the mode in the subsystem). All the incoming data is routed through the currently configured hardware chain, and the results are produced on the component’s output port. When a parameter reconfiguration

occurs, the relevant FPGA configuration has to be loaded into the CPSS using the dynamic partial reconfiguration feature of the FPGA. The BER estimate computed by the ViterbiDecoder component is retrieved from the CPSS register interface (see Section 2.2.2 on page 46) and triggered as an event using the Iris2-FPGA API from the software wrapper.

On the hardware side, the FPGA components are connected to the required chains using VHDL code. This is then passed to the Xilinx toolflow for generating FPGA configuration files (bitstream files). The result is four different bitstream files for the transmitter and five different ones for the receiver (one for each mode in the FPGA CPSS subsystems). These files are programmed into the software wrapper component so that it can load the correct FPGA configuration when the active mode is switched.

With the software wrappers generated, the data plane is now ready for execution on the target platform.

Generate abstraction layer for controller

A drawback of the radio model transformations is that the data plane actually running on the target platform has been considerably changed from the original model. However, the radio controller is written in a way that assumes the high level model is executing. This gap can be closed by generating an abstraction layer for the controller which translates all controller actions on the original model to the data plane currently executing. This hides the synthesis transformations from the radio designer, simplifying the controller development.

In Iris2-FPGA, when a controller wants to reconfigure a radio, it creates a `ReconfigSet`, i.e., a set of reconfiguration actions. These contain a description of what is to be reconfigured. The abstraction layer has to translate these descriptions to their equivalents for the currently running data plane. Since all possible reconfigurations are known from the radio model, and the control specification and the transformations performed during synthesis are known as well, the translation layer is straightforward to generate. It is a one-to-one mapping of high-level reconfiguration actions to their low-level equivalents which can be stored in a look-up table. The same is true for the events, where low-level events triggered by the radio actually running are converted into their high-level equivalents.

This translation layer for Iris2-FPGA is generated as a class implementing the standard controller interface. It sits between the radio and the user-implemented controller. The user inherits from this class (instead of the standard controller base class) and is provided with an abstracted view of the radio. All events and reconfiguration actions are translated to the

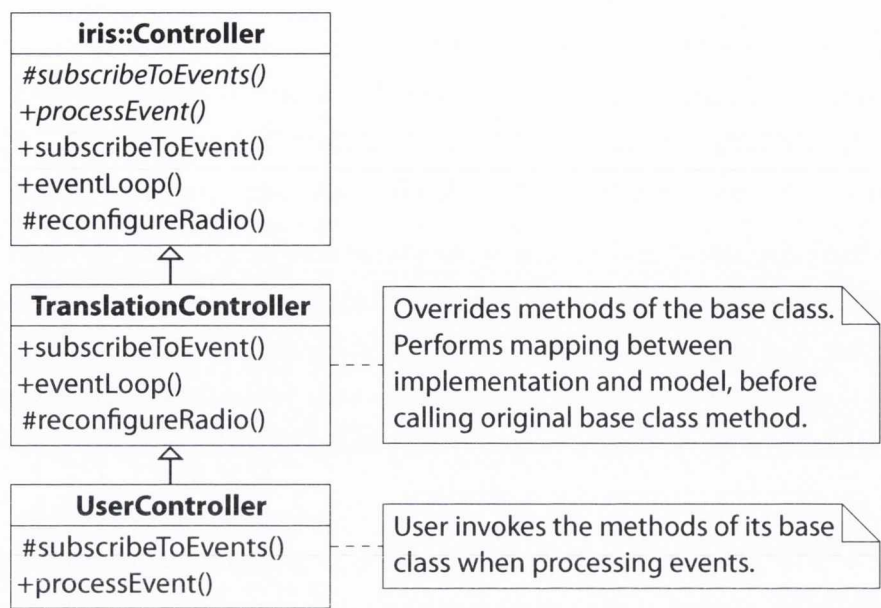


Figure 7.14: Abstraction layer for the radio controller. The user implements the abstract methods of the `iris::Controller` base class, and uses the utility methods of the `AbstractionController` during computation. These perform the translation between the current radio implementation and the original radio model as specified by the user. (UML [92] notation is used here. The parameters of the methods have been omitted for clarity.)

currently running data plane, hiding the details from the radio developer. Figure 7.14 illustrates this translation layer.

With this translation layer generated, the radio controller can be connected and all aspects of the radio are ready for execution.

Generate final radio configurations

To be able to execute the synthesised radio on the target platform, configurations for Iris2-FPGA are to be generated. That is, all components are compiled for the target architecture (including the software wrapper and controller), the FPGA configuration files are generated (bitstream files), and the Extensible Markup Language (XML) files describing the radio are created. This can then be loaded and executed on the target systems.

Compiling all components is a simple job if a cross-compiler or native compiler is available. The GNU Autotools [59] handle cross-compilation¹, configuring the files for building the project accordingly. Iris2-FPGA and the components can then be compiled without difficulties.

¹ Cross-compilation is the process of compiling code for a foreign architecture. It is typically performed when code for an embedded system is compiled on a PC.

```

1 <controller class="TxController"/>
2
3 <engine name="sdfengine1" class="sdfengine">
4   <component name="udprx1" class="UdpRx">
5     <parameter name="port" value="1234"/>
6     <port name="output1" class="output"/>
7   </component>
8   <component name="framer1" class="framer">
9     <port name="input1" class="input"/>
10    <port name="output1" class="output"/>
11  </component>
12  <component name="fpga_cpss1" class="FpgaCpssWrapper1">
13    <parameter name="activeMode" value="4"/>
14    <port name="input1" class="input"/>
15    <port name="output1" class="output"/>
16  </component>
17  <component name="usrptx1" class="UsrpTx">
18    <parameter name="bandwidth" value="2000000"/>
19    <parameter name="centreFrequency" value="2493700000"/>
20    <parameter name="gain" value="0.3"/>
21    <port name="input1" class="input"/>
22  </component>
23 </engine>
24
25 <link source="udprx1.output1" sink="framer1.input1"/>
26 <link source="framer1.output1" sink="fpga_cpss1.input1"/>
27 <link source="fpga_cpss1.output1" sink="usrptx1.input1"/>

```

Figure 7.15: The transmitter XML configuration.

The generation of the FPGA bitstream files is performed using the Xilinx hardware design tools, i.e., the Xilinx ISE Design Suite [44]. The result is a set of bitstream configuration files for the FPGA's CPSS.

The XML files are generated from the transformed model. They are shown in figs. 7.15 and 7.16 for the example radios (see Appendix E for a description of the Iris2-FPGA XML format). Both radios are executed within one SDF engine, implementing the SDF MoC. The `FpgaCpssWrapper1` and `FpgaCpssWrapper2` components are the software wrapper components for the hardware portion of the radios. They are auto-generated and represent the corresponding subsystems in the transformed radio models (see figs. 7.12 and 7.13). The parameter `activeMode` within the wrapper components is used to select the active mode in the subsystem, i.e., which configuration is loaded into the FPGA's CPSS. The controllers use the generated translation layer to abstract the running radio from the developer.

The XML files, bitstream files, and compiled Iris2-FPGA binaries can now be loaded onto

```

1 <controller class="RxController"/>
2
3 <engine name="sdfengine1" class="sdfengine">
4   <component name="usrprx1" class="UsrcRx">
5     <parameter name="bandwidth" value="2000000"/>
6     <parameter name="centreFrequency" value="2493700000"/>
7     <port name="output1" class="output"/>
8   </component>
9   <component name="fpga_cpss1" class="FpgaCpssWrapper2">
10    <parameter name="activeMode" value="5"/>
11    <port name="input1" class="input"/>
12    <port name="output1" class="output"/>
13  </component>
14  <component name="switch1" class="switch">
15    <parameter name="number_ports" value="2"/>
16    <parameter name="selected" value="2"/>
17    <port name="input1" class="input"/>
18    <port name="output1" class="output"/>
19    <port name="output2" class="output"/>
20  </component>
21  <component name="detector1" class="CarrierDetector">
22    <parameter name="psdSize" value="2048"/>
23    <parameter name="signalWidth" value="256"/>
24    <parameter name="thresholdFactor" value="1.2"/>
25    <port name="input1" class="input"/>
26  </component>
27  <component name="deframer1" class="Deframer">
28    <port name="input1" class="input"/>
29    <port name="output1" class="output"/>
30  </component>
31  <component name="udptx1" class="UdpTx">
32    <parameter name="address" value="videorxpc"/>
33    <parameter name="port" value="2222"/>
34    <port name="input1" class="input"/>
35  </component>
36 </engine>
37
38 <link source="udprx1.output1" sink="fpga_cpss1.input1"/>
39 <link source="fpga_cpss1.output1" sink="switch1.input1"/>
40 <link source="switch1.output1" sink="detector1.input1"/>
41 <link source="switch1.output2" sink="deframer1.input1"/>
42 <link source="deframer1.output1" sink="udptx1.input1"/>

```

Figure 7.16: The receiver XML configuration.

the target systems and the radios can be executed.

The radio link works as expected. It streams a video from the transmitter to the receiver over a frequency-agile link which adapts its coding strength according to the channel quality. The system setup is identical to the experiment described in Section 3.4 on page 84 (Figure 3.12).

7.4 Analysis and Discussion

So far, this chapter walked through the development process of an example radio from concept to execution on the target device, using the proposed model-based design techniques. In this section, this development process is analysed and compared to the traditional process. The benefits of the proposed methods are clearly shown.

7.4.1 Comparison with traditional design

Table 7.7 shows a comparison of the proposed design process with tradition methods to design cognitive radios. In the following, each point raised in the table is explained in more detail,

Model-based Design	Traditional Design
• systematic design flow • easy to use • reduced development time • platform-independent system composition • platform-independent design can be synthesised for different target platforms • model captures dynamic radio behaviour • design error detection at design-time (platform-independent) • small design changes fast and easy • automatic integration of hardware accelerators, if components are available	• ad-hoc design • steep learning curve • considerable development time • platform-dependent design and development throughout • one design fits one platform, re-design is necessary when moving to a new platform • only initial radio configuration, reconfigurations not captured • design error detection at run-time • small design changes often require considerable effort • manual integration of hardware accelerators

Table 7.7: Comparison of traditional design and the proposed mode-based design for cognitive radios.

using the radio design in this chapter as an example.

In contrast to traditional cognitive radio design methods, the proposed method is clearly structured and follows a systematic flow, as illustrated in Figure 7.1 on page 184. Developers can follow this flow step by step and have a working system as a result. The tasks for each step are clearly defined, simplifying and accelerating the development process. The design and development process of the case study, detailed in sections 7.2 and 7.3 of this chapter, demonstrates this fact. The traditional approach is more ad-hoc, where developers typically have a concept in mind and start the implementation of the system directly. They continuously add functionality during the development process until the radio is complete and operates as required. Typically, there no complete and structured description of the radio's functionality and behaviour exists in traditional cognitive radio design. The proposed design flow is more structured and can therefore reduce the development time and result in less errors in the resulting system.

The clear and structured design has another advantage: The learning effort of new developers for how to design cognitive radios is drastically reduced. It takes less time to understand the proposed model a cognitive radio than to learn low level details of a specific target platform. Furthermore, these details differ on every target platform, so that the developers face a high learning effort when they need to move to a new target platform. The proposed model-based design technique avoids this. It can be seen from the case study in this chapter that the radio models are simple to design and all complicated tasks can be performed by automatic tools (i.e., model correctness checking and model synthesis).

The majority of the proposed design process is platform-independent. That is, composing the cognitive radio system from components, incorporating reconfigurability, identifying reconfiguration triggers, and checking this design for errors is performed independent of the target platform. For the example radio system chosen for this chapter, all steps covered in Section 7.2 are platform-independent. Developers can decide for the target platform after they are finished with this part of the design, selecting the most suitable platform. In the traditional design flow, the target platform is decided first and the whole design is specific to that platform. Therefore, implementation details inevitably become a part of the design. The developer has to become familiar with the low-level details of the target platform in order to be able to design a radio for it, making the development process unnecessarily complex. Using the proposed model-based design technique, the developer can focus on the important behavioural aspects of a radio and does not need to concern himself with low-level platform details. This reduces the development time and simplifies the design process. In the example radio design illustrated in

this chapter, the only low-level tasks for the developer were the implementation of the missing components (convolutional coder and Viterbi decoder) and the controllers for transmitter and receiver.

Another advantage of the largely platform-independent design process is that designs are portable to different target platforms with minimal effort. Traditionally, porting a design to a new platform means a complete re-design of the radio. However, using the proposed methods, the design can stay unaltered for different target platforms, and only the platform-dependent steps need to be altered. For example, if the video streaming radio of this chapter was to be moved to the GNU Radio platform [36], the designer can start with the tasks in Section 7.3. All steps covered in Section 7.2 are still valid. Since platforms are constantly improved and new ones become available regularly, being able to port a radio design to a new platform with minimal effort is an advantage.

The proposed model captures the dynamic behaviour of a cognitive radio. This is a key advantage of the model-based design technique compared to traditional methods. The model clearly specifies all reconfiguration options. Using traditional design techniques, often the configuration of the radio only represents a snapshot at a specific time. Even if the basis for a design is a radio flow graph, like for example in GNU Radio [36], it only contains the start-up configuration. Modifications of the flow graph later during execution are not captured in the design (only in the implementation code). Therefore it is difficult to assess the behaviour of a cognitive radio during the design process using the traditional methods. In fact, the only way to determine dynamic behaviour is to run the radio and monitor its behaviour in time – something that is clearly not desired for a cognitive radio designer. In the example radios designed here, all configuration possibilities are included in the platform-independent radio models. That is, the different coding schemes, frequencies, bandwidths, and the detection mode are apparent from the model.

This immediately leads to the next advantage of the proposed model-based design technique: the design can be analysed for construction errors before execution. Since the dynamic behaviour of the radio is captured and the MoC is clearly defined, it is possible to analyse the model at design-time and check for errors in its construction. This is before deciding for a target platform. The models for the transmitter and receiver of the radio designed in this chapter were checked for errors before moving to the implementation part. If radio composition errors are found, they can be corrected in the platform-independent stages. Using traditional methods, such errors can often only be detected during execution of the radio, at the end of the development process. It is typically expensive to correct errors this late in the

development process, potentially resulting in extensive re-design. The proposed model-based design approach avoids this.

Many of the platform-dependent tasks in the proposed design flow can be handled by automatic tools, freeing the developer from the implementation burden of low-level tasks. This has the additional advantage that modifications of the radio design can be mapped to a real implementation easily and quickly. Traditionally, developers had to change considerable amounts of low-level code every time the design is changed. For example, changing the modulation technique, adding channel estimation and equalisation, or changing the signal detection technique in the radio designed here only requires small changes in the radio models. If the new components do not exist for the target platform, they need to be implemented but the developer can focus on that component only and does not need to consider the full system. Then, the automatic synthesis tools can generate a new executable radio configuration.

The model synthesis tools allow transforming the high level radio model to a more suitable one for the target platform. They can therefore automatically generate a configuration that suits a platform with hardware accelerators, as done in the example radios for the FPGA platform. This allows developers without hardware design experience to exploit these platforms nevertheless (provided that the target platform has some hardware components already available in its component library). This is demonstrated by the example radio designed in this chapter, where FPGA radio chains are built, yet the developer does not need to concern himself with hardware design and the complications of configuring and reconfiguring the FPGA at the correct times.

7.4.2 Limitations

As outlined in Section 6.2.4, the proposed model-based design technique has its limitations. For the case study system designed in this chapter, these limitations are discussed briefly in the following.

Only the system composition is platform-independent

The radio model only captures the composition of the radio and treats the components themselves as black boxes. Therefore, if components are missing from the library, they have to be implemented manually. This is inherently platform-dependent. The ConvEncoder and ViterbiDecoder in the example radio (see Section 7.3.1) showed this.

Implementing the missing components required low-level FPGA design expertise, but only for the limited scope of these individual components, in contrast to a full radio system. The

component metadata, created during the design phase, was used as an aid for implementation. Further, the components implemented here were available as intellectual property cores from Xilinx. All the developer had to do is configure the core and write a small amount of VHDL code to implement the standard hardware interface required by the target platform. This demonstrates how existing resources, such as software libraries or hardware intellectual property cores, can help with the implementation.

Model-checking does not check functional correctness

Although all model checks completed successfully in Section 7.2.4, that does not guarantee that the receiver can successfully decode the video stream. Since components are viewed as black boxes, their internal function cannot be checked. The only way for the radio designer to verify the radio's function is the final experiment.

A possible method for enabling functional correctness checks is presented in Section 6.2.4, but this is beyond the scope of this thesis.

However, the correctness checks performed here are able to detect a large class of potential design errors and can therefore save considerable amounts of time and effort for the radio developer.

System performance cannot be estimated

As mentioned in Section 6.2.4, it is impossible to estimate the expected performance of the composed radio system on the target platform. That means for this case study, that the synthesised radios might not be executable in real-time, resulting in a failed experiment. The earliest point the radio designer can detect this is at the very end of the development process, when the radio is executed. Then he has to modify his design or change the target platform. Although this can be done with relatively little effort, it is not ideal.

Possible solutions to estimate the performance on the target system at design time are outlined in Section 6.2.4. They are beyond the scope of this thesis and left for future work.

7.4.3 Summary

Designing an example cognitive radio system that can change its operating frequency and adapt its robustness to the current channel conditions demonstrated the benefits of the proposed model-based approach to cognitive radio design. The design is structured, portable, largely

platform-independent, and captures the dynamic aspect of cognitive radios. It can be analysed at design-time, so that errors can be corrected before proceeding to implementation. It is shown that the proposed methods can be applied for a real radio development process and the benefits are highlighted.

Although the radio system picked as a case study is simple, it demonstrates all aspects of cognitive radios. It includes meters (for the *observe* stage) and knobs (for the *act* stage) in both software and hardware, as required by the radio controller. The decision-making (the *decide* stage) is handled by a simple state machine, but the connection through the publish & subscribe MoC allows arbitrary algorithms. That is, a more complex cognitive engine, for instance optimising a multi-objective function through genetic algorithms and applying machine learning, can be implemented using the same principles as for the case study. The proposed cognitive radio design approach can therefore also be applied to more complex radios.

This design method has the potential to drive cognitive radio research forward by allowing researchers to test their ideas in the real world, which often differs significantly from simulations.

Conclusions and Future Work

Contents

Summary of Contributions	227
Future Work	232
Conclusion	237

This thesis proposed the cognitive radio development framework Iris-FPGA, using a combination of field-programmable gate array (FPGA) hardware and general purpose processor (GPP) software for processing. It showed that this is an effective solution for testing cognitive radio ideas in a realistic setting. Further, it presented a model-based design approach for cognitive radios and demonstrated that this is a highly effective method to simplify the design of cognitive radios for tests and experiments with many benefits.

This chapter draws the conclusions from the thesis, highlights its contributions, and outlines areas for further research.

Summary of Contributions

This thesis gave valuable insights into designing for and experimenting with real-time cognitive radio testbeds. It proposes and tests a flexible FPGA-based cognitive radio framework (Iris-FPGA), analyses the radio design approach of this and other frameworks, and proposes a novel method to simplify the design of cognitive radio systems significantly. This reduces design effort, provides higher reliability, and allows easy deployment on different target platforms. The following sections give evidence to these claims by examining each contribution in more

detail. Additionally, implications for the general research field and important insights gained are stated where appropriate.

Experimentation with real-time cognitive radios

1. *Iris-FPGA*

Section 2.2 proposed an FPGA-based cognitive radio framework, Iris-FPGA, which combines the benefits of hardware and software into a single system. It provides mechanisms for the implementation of key cognitive radio concepts, i.e., the cognitive cycle of *observe*, *decide*, and *act*, no matter whether the processing happens in hardware or software. FPGA hardware can be reconfigured at run-time in much the same way as software, the low-level details are handled by the framework.

Concluding, it can be said that integrating FPGA hardware with a software framework poses many challenges. Interfacing the hardware and software parts, hiding low-level implementation details from the user, defining a suitable interface between hardware processing components, choosing an appropriate development board, and executing an operating system on it are just a few examples. These challenges were discussed, possible solutions were given, and the solutions implemented for Iris-FPGA were outlined in Section 2.2.

For cognitive radios, it is important that the FPGA can be reconfigured at run-time, so that the cognitive engine can adjust the radio's operation. The dynamic partial reconfiguration feature of Xilinx FPGAs provides this flexibility and was therefore exploited in the proposed Iris-FPGA framework. This ensures that Iris-FPGA provides the flexibility required by cognitive radios on a platform integrating high performance hardware.

2. *Experiments with Iris-FPGA*

A series of demonstrators of real-world cognitive radios were developed in Chapter 3, demonstrating the capability to implement key features of cognitive radios using Iris-FPGA, such as the cognitive cycle and radio reconfigurability on various levels. Additionally, the experiments helped refining the framework architecture, and provided insights into the challenges of implementing radios for testbeds.

This demonstrated that Iris-FPGA is capable of executing real cognitive radios. It can be concluded that, even with a high level framework like Iris-FPGA, implementing real demonstrators is not trivial and requires considerable effort. Often, many unforeseen practical issues are faced and need to be solved on the way to a working implementation. However, in many cases

it is worth effort as it provides invaluable insights into the feasibility of cognitive radio ideas and theories. Experiments can not only be used as a method to prove or invalidate theories or ideas, often they can help to refine these or trigger new ideas. This was the case for the experiments conducted in this thesis, as they evoked the idea of a new approach to cognitive radio design, presented in Part II of this thesis.

Analysis of cognitive radio design approaches

3. Review and analysis of existing cognitive radio design approaches

Chapter 1 reviewed a representative selection of cognitive radio and software defined radio platforms and frameworks from the literature. Their key strengths and weaknesses are identified and the design methods are outlined. This review can aid as a help for choosing the right framework for a specific cognitive radio application when researchers want to conduct cognitive radio experiments.

One of the most important criteria when designing cognitive radios for experiments is the complexity and effectiveness of the radio design process. An efficient design method helps to prevent that the development of a demonstrator becomes a complex engineering task and enables fast prototyping and execution of the demonstrator. Radio designers are typically interested in the experiment itself, not in the development process. For this reason the radio design approaches taken by the reviewed frameworks and Iris-FPGA was analysed in Section 4.2.

It was found that design approaches vary greatly, require in-depth understanding of the low-level platform architecture, and lack any systematic way of dealing with reconfiguration. The model of computation (MoC), defining the execution semantics of the radio, i.e., how a radio is executed, is often not clearly defined. This results in a cumbersome, ad-hoc approach to cognitive radio development, severely limited design portability, and experimental results which are often impossible for other researchers to fully understand and reproduce. This ad-hoc approach also creates a situation where validating promising research ideas in a real setting becomes a complex engineering task, that distracts from developing the ideas further on a solid experimental foundation.

This discussion motivated the idea of proposing a high-level cognitive radio design approach in Part II of this thesis which can be used to design cognitive radios for wide variety of platforms.

4. *Iris-FPGA: Reflections*

The proposed framework Iris-FPGA is critically analysed in Section 4.1 and its key strengths and weaknesses are identified. Insights into the design of cognitive radios are given.

The key strengths found are the support to implement the cognitive cycle, with *observe*, *decide*, and *act*, the integration of hardware and software components into a single radio, and the effective radio description using *dataflow graphs*. The key weakness identified is that the dataflow graphs used only represent the radio configuration when the radio is started, but do not capture its dynamic behaviour. Cognitive radios, however, can change the structure and configuration of the running radio many times during execution, i.e., the dataflow graph is constantly changed.

The reconfigurations themselves are possible with Iris-FPGA, but are not captured by the otherwise effective radio description, i.e., the radio dataflow graph. These flow graphs are used during radio design, as a method for the developer to describe the radio. Thus, the framework itself is capable of executing cognitive radios of all kinds, as proved by the experiments in Chapter 3, but the radio design process could be improved by using a radio description which includes the dynamic behaviour.

Model-based cognitive radio design

5. *Model-based design background*

From analysing the reviewed platforms in Section 4.2 and from the experiences from chapters 2 and 3 it was found that a model-based design method can solve the cognitive radio design issues identified.

Chapter 5 reviewed model-based design techniques and existing models for signal processing systems from the literature, and evaluated their suitability to model cognitive radios.

This review demonstrated how existing research in related areas can help to trigger ideas for new approaches in the research area or interest. Here, approaches and models applied since the 1970s in traditional signal processing have been reviewed and it was found that they might be applicable for cognitive radio design. They cannot be applied directly, as the highly dynamic nature of cognitive radios is not captured. However, they served as a basis for developing a new model suitable for cognitive radios in Chapter 6 of this thesis.

6. *Platform-independent model for cognitive radios*

A platform-independent model for describing the system composition of cognitive radios based on functional blocks, was proposed in Chapter 6. It allows designs to be understood and checked for computational functionality without first being implemented. The model is also the first to combine both the radio processing and reconfiguration within the same description, simplifying computational verification and precluding the need for implementation platform knowledge.

This model decouples the cognitive radio implementation on a target platform from its behavioural description. It is the equivalent of the models for signal processing, reviewed in Chapter 5, for cognitive radios. It is based on a solid theoretical foundation and can serve as a description for model-based design of cognitive radios. This enables radio researchers to focus on applications of interest without the need to consider low-level implementation details.

7. *Model-based design of cognitive radios*

Methods for using the model as a basis for model-based design of cognitive radios were proposed in Section 5.1. Real cognitive radio implementations can be generated largely automatic from the platform-independent model description. This reduces design effort, provides higher reliability, and allows easy deployment on different target platforms.

It should be noted that the proposed model-based design technique can be applied to a wide range of physical platforms and software frameworks for cognitive radios. It is not a platform or framework itself, it is layer above that. This ensures design portability and does not invalidate the existing platforms.

8. *Improvements of existing frameworks*

The theoretically substantiated cognitive radio model has another potential use: it can influence the way cognitive radio frameworks themselves are implemented. This was demonstrated in Section 6.3, where the Iris-FPGA framework was modified based on the ideas of the proposed cognitive radio model. This resulted in a more flexible and theoretically substantiated implementation platform. Applying the proposed model-based design technique with this framework as a target is more direct, easing synthesis of the model.

This contribution closes the circle between experimentation and theory. That is, as discussed in the introduction of this thesis, experimentation can not only prove theory, it can also trigger new ideas for new theories which in turn are followed by new experiments. This circle is common in many fields of scientific research. For this thesis, the experimentations in Part I

inspired the idea of a novel model-based design technique with strong theoretical foundations. This in turn inspired changes in the experimentation platform, as covered by this contribution.

9. Case study with proposed model-based design approach

In Chapter 7, a case study of the proposed model-based cognitive radio design approach has been presented. The techniques were applied to design a cognitive radio from the idea to its execution on a real-time target platform, demonstrating the benefits of the proposed methods. The design process is largely platform-independent, thus, most of the low level platform-dependent implementation tasks could be handled automatically.

This case study showed that the proposed model-based design techniques can indeed be applied to real cognitive radio design for experimentation. They simplify the process and allow the radio designer to focus on the radio application and the cognitive engine, without the need to concern himself with implementation details. Researchers new to cognitive radio experimentation can use the proposed method to rapidly build a working demonstrator and evaluate their ideas in the real world. The proposed methods significantly reduce the engineering effort involved in implementing the demonstrator, as shown by the case study in Chapter 7. Nevertheless, a good working knowledge of communications theory and algorithms is par for the course.

Future Work

This thesis showed that platform-independent model-based cognitive radio design is promising to enable and simplify experimental research. However, further issues need to be addressed. The following sections give suggestions for future improvements of the proposed methods and indicate other factors that can simplify and encourage cognitive radio experimental research further.

Improve hardware composition tool

A hardware composition tool was outlined in Section 2.2.4 on page 57 that is able to automatically connect hardware components into chains and generate the corresponding software wrapper component code. As shown in Figure 2.6 on page 60, it uses the source code of the hardware components, some metadata associated with it, and a template for the software wrapper component to map an Extensible Markup Language (XML) description of the hardware chain to the corresponding Verilog and C++ code.

This tool reduces the burden for radio designers by automatically generating low-level code from a high level description. However, they still have to decide which components should execute in hardware and which in software, which requires detailed implementation knowledge. The XML description of the hardware chain has to be prepared, although designers should ideally only need to describe the full radio and should not be concerned with how it is implemented. Thus, the hardware composition tool could be improved as follows.

The tool would be provided with a high-level description of the full radio in XML format, not just the hardware part. It could then decide which components are to be executed in hardware and which in software. This is attractive if the used components are available in both software and hardware. The tool would then generate the hardware chains, the corresponding software wrappers, and a new XML file which reflects the mapping decision.

The decisions could be based on the following inputs:

- performance of both hardware and software component implementations,
- resources available on the target FPGA,
- resource requirements for hardware components,
- data transfer performance between hardware and software,
- target performance of the radio, and
- the level of reconfigurability required for components.

The decision-making algorithm could be an optimisation algorithm with the objective to minimise the power consumption, subject to the constraint of meeting the target performance with some margin. Other objectives are possible, for instance, minimising the used FPGA resources to allow using a smaller device, or keeping the processor utilisation low to enable the implementation of higher network layers. Further research is required to determine a suitable decision-making algorithm.

This tool could be integrated into the synthesis tool for the model-based design technique proposed in Chapter 6. The hardware/software partitioning of the generated radio implementation could then be determined automatically, freeing the radio designer from this task. This further abstracts the implementation details from the radio designer.

Higher layers of the network stack

This thesis focuses on the design and experimentation with the physical layer of cognitive radios only. In future, ways to include higher layers of the network stack into the model-based design technique proposed in Chapter 6 could be investigated.

Research in the area of cognitive radios goes beyond physical layer links, it involves cognitive networks consisting of multiple nodes [129]. Today, most of the research performed on cognitive networks is based on simulations. These often do not reflect the reality very well due to many assumptions being made. Being able to test concepts and ideas for cognitive networks in realistic experiments is essential to prove their validity. Thus, frameworks and design approaches for cognitive network nodes, similar to the ones proposed in this thesis for physical layer radios, have the potential to drive research on cognitive networks forward.

The new generation of the cognitive radio framework Iris, briefly described in Section 6.3 of this thesis, provides support for higher layers of the network stack. These layers can be reconfigured at run-time as directed by a cognitive engine. Iris 2 is therefore one possible target platform for experiments with cognitive networks. The model-based design approach proposed in Chapter 6 of this thesis could be extended to include network layers. This should include the dynamic behaviour of these layers to enable a similar analysis as was done in this thesis for the physical layer only.

This extension of the model-based design concept to cognitive networks would result in faster development times and more portable designs, enabling researchers to test their networking ideas in a real setting, rather than in a simulator.

Functional correctness check of the cognitive radio model

The radio design approach proposed in this thesis focuses on the system composition aspect of radio design only, not the internal workings of each component. It treats the individual components as black boxes and is only concerned with their external interface. This implies that the design-time correctness checks proposed in Section 6.1.4 can only check whether this composition is correctly constructed and whether it is executable on the target system. Although this can eliminate a large source of errors, the functional correctness of the radio cannot be confirmed.

One possible solution to allow functional checks at design time is to provide a functional description of each component in some portable high-level language. This can then be used to simulate radios before synthesising them to the target platform.

A similar approach is followed by the Xilinx System Generator tool [31], a blockset for Simulink. All hardware components added to a Simulink model have a Matlab function associated with them which exactly matches the component's implementation on the FPGA. When the model is simulated, this Matlab description is executed. Designers can use the simulation

output to verify the functionality of the model, i.e., to perform a functional correctness check. Once all errors are corrected, the Simulink model can be synthesised for the FPGA and the implementation is guaranteed to match the Simulink model. This eases the design of FPGA systems significantly.

However, it is difficult to maintain a functional description of the component that exactly matches its implementation on the target platform. Since the component implementation is inherently platform-dependent, a perfectly matching functional description often becomes platform-dependent as well. It has to be researched how the description and implementation can be kept consistent. For example, generating the implementation from the description directly would solve this issue.

If FPGA hardware components are used, another possible solution is to directly use existing hardware simulators from a radio simulation tool. That is, the radio simulation tool would use a hardware simulator in the background, e.g., ModelSim [130], for all hardware components. The software components, if implemented in a portable language, could be compiled directly on the design system and used in the simulator. This approach becomes inherently platform-dependent. Nevertheless it might be attractive if the target platform has already been decided.

Target performance estimation in the design

Section 6.1.4 has provided algorithms to check if the radio model is correctly constructed. These are platform-independent. However, to aid the decision about the target platform for the model, it would be beneficial to estimate the expected performance of the radio design for various possible platforms. Radio designers could use these estimations to decide for a specific target platform.

A possible solution is to benchmark all individual components in the component library of a target platform on various different target platforms. The resulting performance data could be kept with the component metadata used for model-checking. Now the overall radio performance can be estimated at design-time.

This estimation is not trivial. The time taken to execute individual software components differs when they are included in a radio since all components share the same processor. If hardware and software components are combined, the data transfer times at the interface between the processor and the hardware needs to be considered. Thus, the estimation the overall radio performance on a target system based on individual measurements needs to be investigated further.

If FPGA hardware is used in the radio, the performance estimation could be extended to include power consumption and resource utilisation on the device. The overall estimate could be obtained in a similar way as for the performance, i.e., by combining individual component measurements. This could aid the radio designer in the selection of the FPGA model.

Additional enabling factors for cognitive radio experimental research

This thesis contributed to enabling cognitive radio experimental research by providing a platform and design methods for fast prototyping of demonstrators in a research context. However, there are more factors which can help to enable more experimentation with cognitive radios. The factors detailed in the following are less technical and are not improvements of the outcomes of this thesis. They represent the personal views of the author.

First of all, the acceptance of experimental results within the research field should be raised. Many high quality journals in the general area do not value experimental results and prefer theoretical findings and simulation results. Due to the complexity involved, often experiments are relatively simple compared to what is possible in simulations. Unfortunately, this results in a higher acceptance for simulation results than for experimental results. This is partly due to the lack of a high quality scientific journal for cognitive radio in general and for experimentation specifically. Existing high quality journals typically publish mainly theoretic and simulation results and do not value the contribution of conducted experiments. A specialised journal for the area of cognitive radios has the potential to improve this. This would enable more researchers to benefit from the results and insights gained by experiments from others. Further, it would encourage more researchers to conduct experiments and publish the results.

Further, especially for experimental research, it should be encouraged to publish negative results. That is, if an idea was invalidated through an experiment, this is a significant contribution to the area of research. It saves other researchers from committing the same mistake or it may trigger new ideas. In fact, it is believed that sometimes a negative result is a bigger contribution than a positive result, as it might have a higher impact on future ideas.

Another limiting factor for experimental research in this field is that results are difficult to reproduce by other researchers. Experiments performed by one set of researchers are typically difficult to repeat by others to confirm the findings. This way of validating results is applied in other areas of science. For instance, the journal "Annals of Internal Medicine" [131] encourages authors to provide the study protocol, the statistical code used to generate results, and the dataset from which the results were produced [132]. Being able to repeat experiments and

reproduce results would make the results stronger and would allow other researchers to improve algorithms and ideas. It is believed that the high level platform-independent model to describe cognitive radios proposed in this thesis contributes to the reproducibility of experimental results. A model can be transferred to other researchers and they can synthesise it for their target platform of choice and conduct the same experiment again. It would be a great benefit to the area if a potential future cognitive radio journal would encourage researchers for every publication which includes experimental results to attach the information to reproduce it, similar to the “Annals of Internal Medicine” journal.

Conclusion

This thesis contributed to enabling cognitive radio experimental research through providing valuable insights into cognitive radio design and experimentation and proposing new methods to simplify the experiment development process. An experimentation platform was proposed, assessed, and evaluated and a high level cognitive radio design method was developed and tested.

Recall the question raised in the introduction: “How can cognitive radio experimentation be simplified for researchers, so that algorithms and ideas can be tested in the real world?” The answer to this, following the contributions of this thesis, is through providing an easy-to-use cognitive radio experimentation framework that is able to cope with the complexity of the signal processing algorithms involved, and through a clearly structured and generic design methodology that allows researchers to focus on the radio design, rather than platform-specific implementation details. This simplifies the design process of cognitive radios for experimentation significantly.

Radio developers do not require detailed platform knowledge, the development process requires less manual effort, and designs can be easily ported to other target platforms. By applying the proposed cognitive radio design approach and using a flexible and capable implementation framework like Iris-FPGA, researchers are able to test their algorithms in the real world with significantly less effort.

Thus, this thesis enables cognitive radio experimental research, which can bring cognitive radio technology closer to reality.

Appendices

A The Iris XML Format

The Extensible Markup Language (XML) format to describe radio configurations for Iris in the version of 2006 (existing before work on this thesis commenced) is described in the following. It is also used for the Iris-FPGA framework, as described in Section 2.2 of this thesis. This appendix first shows a few example configurations before giving a complete syntax tree for the format.

A simple example XML file is shown in Figure A.1.

The description block is for a general description of the radio. A name for the radio and a longer description is given for later reference. This information can be queried, but is not relevant for the radio's execution.

```

1 <softwareradio>
2   <description>
3     <name>Image TxRx Simple</name>
4     <comment>loads image and displays it</comment>
5   </description>
6
7   <structure name="Image TXRX">
8
9     <rfcomponent type="ImageTx">
10      <parameters>
11        <outputBlockSize>512</outputBlockSize>
12        <filename>Iris.bmp</filename>
13      </parameters>
14    </rfcomponent>
15
16    <rfcomponent type="ImageRx" />
17
18  </structure>
19 </softwareradio>

```

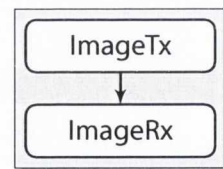


Figure A.1: Example Iris XML description. An image is loaded from a file by the ImageTx component and then displayed on the screen by the ImageRx component.

The main body of the `softwareradio` may contain multiple `structure` blocks, where each describes a separate block diagram. In the example in Figure A.1, only one such structure is present. It contains two `rfcomponent` blocks, each representing one component from the component library. The `type` attribute of the `rfcomponent` tag identifies the component that should be loaded from the library. It is possible to use multiple components of the same type. In this case, the optional attribute `name` (not shown in the example) specifies a unique instance name for identification.

When component parameters need to be configured, a `parameters` block is used. It contains tags for each adjustable parameter of the component. In the example, the `ImageTx` component sets the `filename` parameter to `Iris.bmp`, and the `outputBlockSize` parameter to 512. With these settings, the component reads the given bitmap file in chunks of 512 bytes at a time and passes these chunks to the next component in the chain. The `ImageRx` component does not have any parameters, it simply consumes blocks of bytes from its input and draws the corresponding image on the screen.

In Iris, the connections between components are implicit in the XML file. That is, the order of appearance in the file determines the connections. In the example in Figure A.1, the output of the `ImageTx` component is implicitly connected to the input of the `ImageRx` component. If the data path is linear, with each component having only one input and output, this is relatively straightforward.

If the data path is branched at some point, a `parallel` block must be used. This is illustrated in Figure A.2. All the `rfcomponent` blocks within the `parallel` block receive the same data, i.e., the data is duplicated after the `ImageTx` component. In the example, the image is both displayed on the screen and written into a file as raw data. Each `parallel` has a `name` attribute, used to address it from a controller. The `exclusive` attribute is `false`, meaning that the elements of the `parallel` are both executing with the duplicated signal at the inputs. If the `exclusive` attribute is `true`, then the `parallel` acts as a switch between all its elements, i.e., only one of them is active at a time. The `selected` attribute selects the active branch in this case. In case one branch should contain multiple components in a series, another `structure` block can be embedded inside a `parallel`. Then all the components in that structure are in one branch of the `parallel`. Embedded structure blocks follow the same rules as the top-level structure, i.e., they may contain further `parallel` and `structure` blocks.

In Iris, components may have multiple input or output ports. However, all inputs must be of the same type and size. Such components can be placed after a `parallel`, for merging a branch, or after components with two outputs. This feature is useful for components like a bit error

```

1 <structure>
2   <rfcomponent type="ImageTx">
3     <parameters>
4       <outputBlockSize>512</outputBlockSize>
5       <filename>Iris.bmp</filename>
6     </parameters>
7   </rfcomponent>
8
9   <parallel name="branch" exclusive="false" selected="0">
10    <rfcomponent type="ImageRx" />
11
12    <rfcomponent type="FileWriter">
13      <parameters>
14        <filename>ImageData.bin</filename>
15      </parameters>
16    </rfcomponent>
17  </parallel>
18 </structure>

```

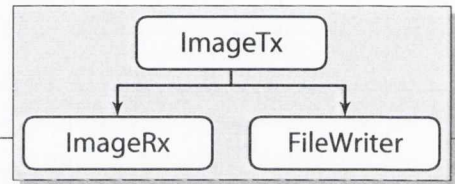
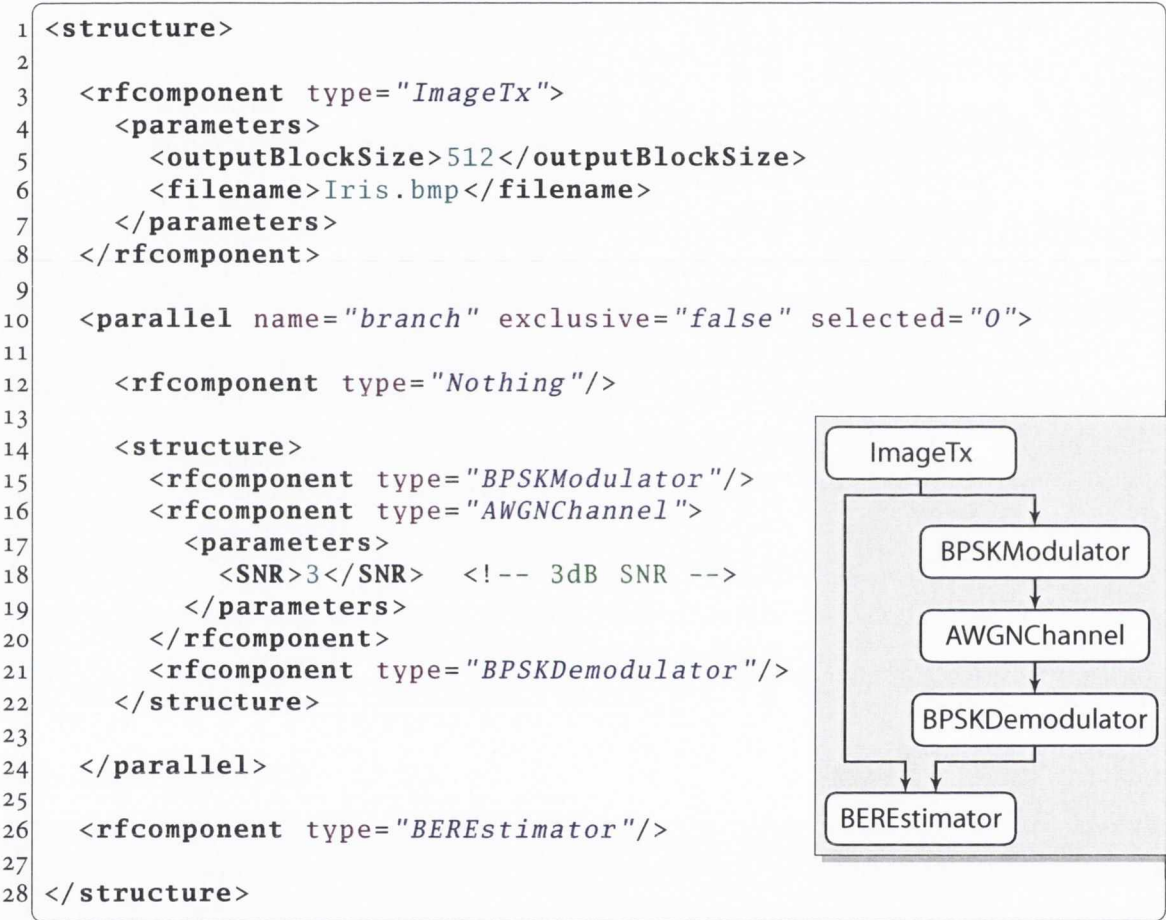


Figure A.2: Iris XML structure with branching, describing a configuration where the data is duplicated after the ImageTx component is passed into both ImageRx and FileWriter.

rate (BER) estimator comparing the values of both inputs. Figure A.3 shows an example. The `parallel` introduces two branches with duplicated signals. One branch contains the `Nothing` component, a special component which just passes its input signal to the output. The second branch contains a structure where the data is modulated, noise is added, and then demodulated again. The `BEREstimator` component after the `parallel` has two inputs. It compares the data bitwise, estimates the BER, and outputs the results on the screen.

The full syntax of Iris XML files is given by the tree in Figure A.4. The `control` block is used to specify the radio controller, responsible for obtaining radio and environment information (the *observe* stage), making reconfiguration decisions (the *decide* stage), and applying these reconfigurations (the *act* stage). The `<control>` block contains the required information, with `<implementation>` selecting the controller name and `<language>` selecting the binary format (can be one of `native.dll` or `java`). The `native` option stands for native dynamic libraries, typically implemented in C++. Controllers can also be implemented in Java using Java's Java Native Interface (JNI), although this feature is rarely used and not fully tested.



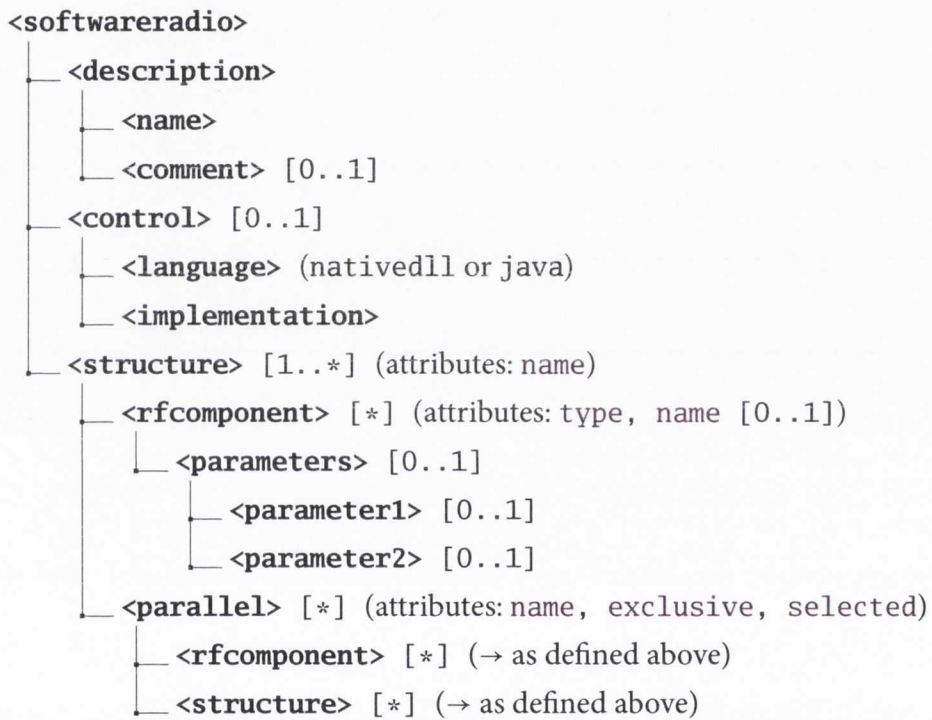


Figure A.4: The Iris XML syntax. A node stands for an XML tag, the numbers in [] denote the multiplicity (none given = exactly once; [0..1] = optionally once; [1..*] = once or more; [*] = any number, incl. zero). Note that the tree is defined recursively (e.g., **structure** blocks can contain **parallel** blocks, which can contain **structure** blocks again, etc.)

B **Hardware Platforms for Cognitive Radio Testbeds**

Contents

B.1 ASIC and ASSP	247
B.2 Microprocessors	248
B.3 FPGA	254
B.4 Summary	255

The following sections discuss several hardware platforms which might be used for cognitive radio implementations and present their advantages and disadvantages. There is a large number of hardware platforms available which combine several of the discussed platforms into one, e.g., tile processors combining multiple of the discussed platforms on one chip. However, only the elementary processing platforms are considered in the following sections.

**B.1 Application Specific Integrated Circuit (ASIC) and
Application Specific Standard Part (ASSP)**

An *application specific integrated circuit (ASIC)* is an integrated circuit (IC) targeted for a particular application. They are custom-made for one customer and only one customer. For example, a particular line of a cellular phone from one company might have an ASIC at its core which has been explicitly developed for that phone. Other companies could not use that chip in their phones. ASICs are highly optimised for the task they are designed for and can perform

only that one task. They are usually the fastest option with the lowest power consumption, compared to other types of hardware platforms.

The non-recurring engineering costs of an ASIC are very high. This includes, among other things, the design of the circuit, development of prototypes for testing, and the production of masks for the silicon. The production costs on the other hand are very low. Thus, ASICs are an attractive solution for mass production of the same system.

Due to the fixed functionality of ASICs which cannot be altered after manufacturing, they are of limited use to cognitive radio systems with their required flexibility. However, it is possible to implement a variety of different functions for a cognitive radio on the same ASIC, possibly parametrised, and to choose an appropriate alternative at run-time. This is an attractive solution for systems where low power consumption and small form factor are the dominating design goals and only a low level of flexibility is required. For research testbeds and more complex cognitive radio systems however, ASICs are not an option due to the lack of flexibility and the high production costs.

An *application specific standard part (ASSP)* is an IC that implements a specific function that, in contrast to ASICs, appeals to a wide area of customers. It typically serves as a building block of a larger system and is available off-the-shelf. Examples are analog-to-digital converters (ADCs) or video decoder chips, integrated in many different products from many industries as building blocks. ASSPs are typically part of every production system. As an analogy, they are the hardware equivalent of a software library, i.e., many different products apply ASSPs in different contexts and application areas. However, they cannot be used to build cognitive radio systems as a whole since their function cannot be altered.

Thus, ASSPs and ASICs cannot be used for the digital baseband processing part of the cognitive radio testbed system to be designed in this work due to the limited flexibility they offer.

B.2 Microprocessors

Microprocessors are single-chip, multi-purpose devices at the core of a computer system. They are programmable using software programs, i.e., a set of microprocessor instructions which are stored in memory. The processor executes the program using four steps: *fetch*, *decode*, *execute*, and *writeback*. During *fetch*, the next instruction is retrieved from memory. In the *decode* step, the parts of the processor that need to be activated for the instruction and the operands

are determined. The microprocessor then executes the instruction and writes the result back to some memory location. After that, the processor continues with the next instruction until the program has come to an end.

This architecture separates the function (software) from the hardware, and therefore the program can be changed and reconfigured completely at any time during execution without difficulty. This high flexibility makes the microprocessor an attractive hardware platform for cognitive radio systems. Additionally, software programs are significantly easier to develop and maintain than hardware designs.

However, traditional microprocessors operate in a sequential manner, i.e., only one instruction is executed at a time. This is a significant drawback and results in very poor performance for algorithms that could be parallelised. It should be noted that instruction pipelining, where the execution of multiple instructions is partially overlapped, has been used in traditional microprocessors for a some time. However, instructions are still executed in order in a serial manner. Many of the digital signal processing (DSP) algorithms used in communications systems can be implemented very efficiently in parallel, with out-of-order concurrent execution of many instructions. Thus, microprocessor implementations of these algorithms are up to several orders of magnitude slower than implementations for a parallel hardware architecture.

For example, a finite impulse response (FIR) filter, a component very common in radio systems, can be highly parallelised. To compute one output sample, the input has to be multiplied with each filter coefficient and the results are accumulated. In a parallel implementation, all these multiplications can be done concurrently, resulting in orders of magnitude faster execution than a sequential implementation where only one operation can be done at a time. Traditional microprocessors only support sequential implementations.

However, there is a trend towards more hardware parallelism in today's microprocessor architectures. Vendors integrate multiple processing cores into one chip to allow efficient execution of parallel programs. However, the level of parallelism with these architectures is typically 2, 4, or 8 processing cores — much lower than for ASICs or other hardware architectures. Additionally, the traditional software development tools and models cannot be applied to parallel software. New models are required for programming multi-core processors, requiring significantly more development and maintenance effort than for traditional software.

The most general type of a microprocessor is the general purpose processor (GPP), found in PCs and laptops. Additionally, there is a large variety of specialised processors available today, optimised for specific application domains. In the following, some commonly used types of microprocessors are outlined and their suitability for cognitive radio baseband processing is

assessed.

B.2.1 General purpose processor (GPP)

Of all microprocessors, GPPs have the most generic instruction set. They are designed to fit any purpose. Therefore all computation tasks can be implemented on a GPP without much effort. Examples of GPPs are Intel's Core i5 or Xeon processors [133], AMD's Phenom or Operton processors [134, 135], IBM's PowerPC processors [136], or SUN's SPARC line of processors [137].

Since GPPs are generic, they can be inefficient for specific tasks. For example the cache design might not suit DSP algorithms, where the same instruction is typically executed on large amounts of data. Additionally, the power consumption of typical GPPs is between 20 W and 150 W, making them unsuitable for small mobile devices (e.g., the Intel Xeon 5000 series of processors consumes between 40 W and 130 W [138]).

GPPs are the number one choice for cognitive radio experimentation in a research environment, simply due to the fact that they are readily available, affordable, and easy to handle. When the power consumption and processing power of GPPs is not enough, other platforms need to be considered.

B.2.2 Digital signal processor (DSP)

This type of microprocessor is especially optimised for the fast operational needs of DSP algorithms. DSP processors often contain specially optimised hardware for instructions used frequently in DSP algorithms, e.g., multiply-accumulate or single instruction, multiple data (SIMD). Their memory architecture is optimised for streaming data, so that large amounts of data can be processed efficiently. Many DSP processors include multiple processing cores, which are able to execute the same instruction on different data items in parallel. Multiple data paths are also commonly found in DSP processor architectures. All these features allow high performance execution of DSP algorithms, but make the execution of general purpose programs inefficient or impossible.

For cognitive radio implementations, DSP processors are an attractive hardware platform since the physical layer of a communication system is mainly performing DSP tasks. Reconfiguring the algorithms during execution means replacing or changing a software program, which is generally simple and fast. However, efficient programming of DSP processors often requires the use of assembly language with instructions specific to the make and type of the processor.

This programming is difficult to optimise, maintain and port to other processors. Furthermore, the level of parallelism in DSP processors is usually lower than the implemented programs would theoretically allow. For example, an FIR filter with 255 taps could theoretically execute all 255 multiplications required for one output sample in parallel. However, if implemented on an TMS320C6455 DSP [139], currently Texas Instrument's fastest processor, only 8 multiplications can be executed in parallel. When 6 of these cores are embedded onto the same device, also available from Texas Instruments, 48 operations can be executed in parallel. This is a considerably high level of parallelism, however, many algorithms could be parallelised more on other hardware architectures. Furthermore, it should be considered that the TMS320C6455 is one of the most advanced DSPs available on the market, i.e., other DSPs allow less parallelism.

B.2.3 Graphics processing unit (GPU)

A graphics processing unit (GPU) is a specialised microprocessor optimised for 3-dimensional (3D) graphics algorithms. These algorithms include texture mapping, rendering polygons, geometric transformations, and programmable shaders; all of which are of high computational complexity, involve matrix and vector operations, and can typically be parallelised easily. Thus, GPUs are parallel architectures consisting of many processing cores (up to hundreds of cores) with specific instruction sets for 3D graphics.

Due to the high performance of GPUs, their application to non-graphics algorithms has increasingly been studied. Nvidia, one of the largest high-performance GPU manufacturers, released a software development kit (SDK) called CUDA in 2007 which allows to use their top-end GPUs for general purpose computing with a C-like programming language [140]. This enabled GPU-based acceleration of many applications. For instance, Adobe utilises GPUs in its Creative Suite since version 4, and the VideoLan VLC Player accelerates video encoding and decoding by using the GPU.

Additionally, the relatively new open standard and programming language Open Computing Language (OpenCL) [141] is expected to be widely adopted in future. It allows programming GPUs and other parallel architectures with a simple C-like language for general purpose computing. The consortium supporting the standard includes the two major vendors for high performance GPUs, Nvidia and AMD. Therefore, it is expected that the number of non-graphics applications using GPUs will continue to rise.

For cognitive radios, GPUs are attractive since the type of algorithms used for digital baseband processing generally fits very well to the capabilities of GPUs. It is therefore expected

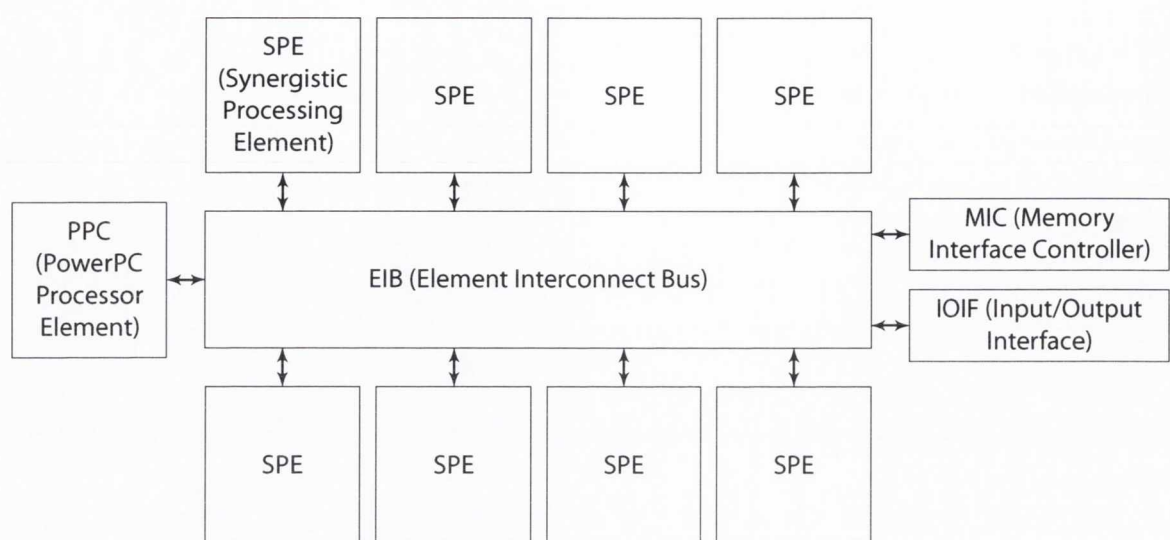


Figure B.1: The Cell Broadband Engine architecture.

that cognitive radios could benefit from the computational power of GPUs.

However, GPUs are always used in conjunction with GPPs which control the application flow and schedule the tasks for the GPU. There is significant overhead in copying data to and from the GPU, so that to see real performance benefits, a significant amount of computationally intensive work has to be performed on the GPU. General purpose GPU computing is still in an early phase, the available SDKs are only just beginning to be adopted. Therefore, to the author’s knowledge, GPUs have not been applied in cognitive radio testbeds to date.

B.2.4 Cell Broadband Engine (CellBE)

The STI Alliance (Sony, Toshiba, and IBM) designed and developed a processor architecture entirely different from other known microprocessors, the Cell Broadband Engine (CellBE). It was originally designed for Sony’s Playstation 3 but is used for a wide range of high performance computing tasks today. It contains one PowerPC Processor Element (PPE), eight Synergistic Processing Elements (SPEs), the Element Interconnect Bus (EIB), the Memory Interface Controller (MIC), and the Input/Output Interface (IOIF) on one chip. Figure B.1 shows a diagram of the CellBE. A brief summary of each element is given in the following. The information given about the CellBE is based on [142, 143].

The PPE is the master of the CellBE, it runs the operating system and distributes and gathers tasks and data across the chip. Its instruction set is based on the PowerPC 970 (as used

in Apple's Power Mac G5) with a 64-bit architecture. It also supports AltiVec SIMD vector operations for higher performance.

The SPEs are the main carriers of the processing power of the CellBE. They are designed and optimised for high-speed SIMD operations only, e.g, they cannot be used for executing an operating system. External memory cannot be accessed directly from the SPEs, instead a relatively small but high performance local store is used. It is not a cache, it can be used in a similar manner to regular memory which is local to one SPE only.

The EIB is the infrastructure to transfer data between the PPE and SPEs, between SPEs, or between external memory and the local store of the SPEs. This can only be done by direct memory access (DMA) transfers, maximising performance and parallelism (processing can continue while data is being transferred).

The MIC is used to provide an efficient connection to the system memory. It can only be accessed by the PPE, which sets up DMA for transfer in and out of the SPE local stores.

The IOIF is the connection between the CellBE and external peripherals. It is used, for example, to interface to a GPU and other peripherals.

The combination of the tremendous processing power of the SPEs, the fast on-chip communication bus EIB, and the PPE make the CellBE a very high performance microprocessor. For example, it can achieve 204.7 Gflop/s for dense matrix multiplication with single precision [143, p. 5].

As a downside, it is difficult to program, since all the hardware operations, movement of data, usage of caches, etc. have to be hand-optimised and are not performed by the compiler. Developers have to know the CellBE's architecture very well to be able to optimise their code and achieve performances close to the theoretical maximum of the device. Straightforward implementations from portable C/C++ often execute considerably slower than on standard x86 architectures since they are running on the PPE only.

The CellBE in its first version consumes approximately 50 to 60 W and is available at a similar cost as regular PC central processing units (CPUs). It is suitable for the implementation of cognitive radio systems as it combines high processing power with high flexibility and reconfigurability. GNU Radio [36], an open-source general purpose software defined radio (SDR) toolkit, for instance, makes use of the CellBE in the Playstation 3. Due to the difficulties involved in using the CellBE's full potential and, compared to hardware solutions, still limited performance figures, it was decided not to use the CellBE for the framework developed for this thesis.

B.3 Field-Programmable Gate Array (FPGA)

Field-programmable gate arrays (FPGAs) are configurable silicon devices that allow application designers to implement custom hardware architectures to accelerate computation. By tailoring the hardware design to the algorithm at hand, specifically exploiting parallelism, it is possible to achieve speed-ups in orders of magnitude over and above GPPs. Signal processing is one domain where they have found significant use, primarily due to the regular, parallel structures involved in most computations.

The most common FPGAs today are static random access memory (SRAM)-based; when powered-on, they are effectively blank – a configuration is applied from an off-chip memory and the chip functions as per the configuration. Most FPGA designs are implemented using hardware description languages (HDLs), such as VHDL and Verilog. However, there are an increasing number of tools that allow mappings from higher-level descriptions. The HDL or higher level descriptions are synthesised by automatic tools to obtain an FPGA configuration for a particular chip.

Today's FPGAs often contain specialised hard silicon for specific applications, e.g., memories, signal processing elements with multipliers, or even full embedded processors. This gives hardware developers the tools to develop faster circuits and allows using FPGAs for a wider range of applications. Embedded processors are of particular interest for more complex systems since they allow to design a full system-on-a-chip (SoC) with a combination of software and hardware for maximum efficiency and performance.

Xilinx FPGAs support dynamic partial reconfiguration, where parts of the circuit configuration can be replaced at run-time, while the remainder of the design continues to function. This offers great opportunities for the design of adaptive systems and is a clear enabler for FPGAs as a platform for cognitive radio implementations as it combines high computational power with run-time reconfigurability.

Several different types and sizes of FPGAs are available from different vendors. They typically consume powers from a few tens of milliwatts to up to around five watts. As an indication for the performance of FPGAs compared to DSP processors, Berkley Design Technology, Inc. performed an industry standard benchmark based on orthogonal frequency division multiplex (OFDM) channels [144]. It specifies the number of standardised OFDM channels the device under test can execute in real-time. The fastest DSP processor tested is the Tilera TILE64 TLR26480 at 866 MHz which can process 15 OFDM channels. In comparison, the Xilinx Virtex-4 FX140 FPGA can process 432 of these channels, i.e., an order of magnitude

difference.

The combination of all the above facts makes FPGAs attractive as a platform for cognitive radios. However, in general FPGAs are more difficult to handle and to design for than micro-processors. Typically, considerable hardware design experience is required to build systems as large and complex as cognitive radios. This disadvantage needs to be overcome by providing a framework and high level design tools to build cognitive radios with ease.

B.4 Summary

A summary of all the discussed platforms along with their advantages and disadvantages is shown in Table B.1. As can be seen, the platforms that are cheap to develop for and offer high re-programmability are comparably slow, while the faster platforms for signal processing are more difficult to develop for.

	DSP speed	Power Consumpt.	Design Flexibility	Repro-grammable	Developm. Cost	DSP Tool Support
ASIC	outstanding	low	outstanding	poor	high	poor
ASSP	outstanding	low	poor	poor	low	poor
low-cost GPP	poor	medium-high	fair	excellent	low	poor
high-perf GPP	good	high	fair	excellent	low	poor
DSP	good	low	fair	excellent	low	good-outstanding
GPU	good	high	fair	excellent	low	medium
CellBE	good	high	fair	excellent	low	poor
FPGA	excellent	low- medium	excellent	good	medium-high	good

Table B.1: Comparison of processing platforms for cognitive radios. The data and grading scale is taken from [145], except for GPU and CellBE, which is an estimate of the author from his own experience. The scale is, from best to worst: outstanding, excellent, good, medium, poor. For power consumption and development cost, the following scale is used: high, medium, low.

C Design Details of the Iris-FPGA Experiments

Contents

C.1 Simple Audio Transmitter Setup	257
C.2 Video Streaming Link with Spectrum Sensing	260
C.3 Video Streaming Link with Adaptive Coding	261
C.4 Combined Adaptive Coding and Spectrum Sensing	264

This appendix describes some design details for the experiments described in Chapter 3. Hardware configurations, implementation details, and other details are presented.

C.1 Simple Audio Transmitter Setup

The first demonstrator serves as an initial case study, used to prove the Iris-FPGA framework is functional and to gather experience in the implementation of radios, so that the framework could be refined. Hence, it uses a simple setup for the hardware interface in the customisable processing subsystem (CPSS). This does not include the embedded shared memories for the data transfer of the virtual architecture proposed in Section 2.2.2. Instead, shared registers are used to transfer the data, with individual samples moved to hardware and processed one by one. In the following the details of this implementation are illustrated and reasons for altering this architecture to the one given in Section 2.2.2 are given.

The CPSS configuration as used for this demonstrator is shown in Figure C.1. As discussed

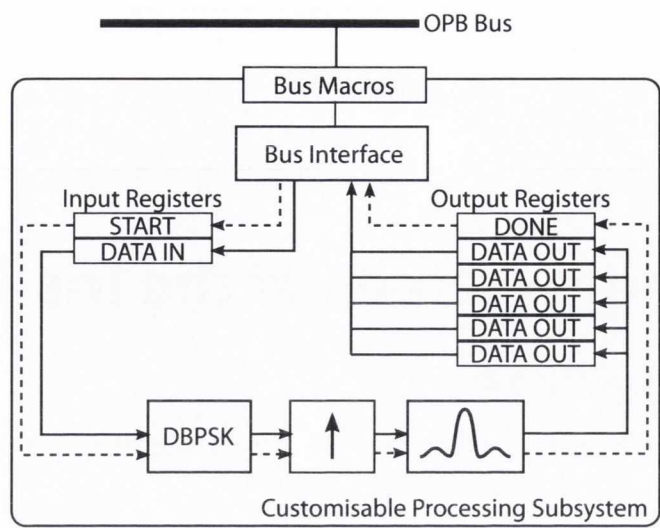


Figure C.1: The CPSS for the audio transmission experiment. The input, output, and control registers are shown. Solid arrows represent data flow; dashed ones represent control flow. Note: the number of registers is not exact.

in Section 3.1.2, the processing components in hardware are the modulator, the upsampler and the pulse shaper, all shown as symbols on the bottom of Figure C.1.

The hardware components were implemented manually. The differential binary phase shift keying (DBPSK) modulator takes 1-bit symbols and produces 1-bit sample values, representing either $1 + 0j$ or $-1 + 0j$ in the complex baseband. These samples are then upsampled by a factor of four and passed to the pulse shaper (i.e., root raised cosine filter). Since the only possible sample values are ± 1 , the filter is implemented without using multipliers. Instead, the value of each filter coefficient is either added or subtracted, depending on the input sample. This saves significant logic area, and allows the circuit to achieve the 100 MHz timing required by the On-Chip Peripheral Bus (OPB) bus with ease.

The utilisation of the field-programmable gate array (FPGA)’s logic fabric is given in Table C.1. It can be seen that the CPSS uses only a small portion of the available logic. This suggests that significantly more complex hardware sub-systems can be used in future applications. This implementation does not make use of the other heterogeneous resources available such as embedded multipliers and memories, but these are of crucial importance in more complex signal processing components.

	PSS	CPSS
Slices	5,927 of 6,848	1,446 of 6,848
(% full)	87 %	21 %
BlockRAMs	14	0
Multipliers	0	0

Table C.1: FPGA utilisation in the PSS and CPSS regions. BlockRAMs are 18 Kb embedded memories, and slices are the basic measurement of logic utilisation (a Virtex-II slice contains two 4-input lookup tables and two flip-flops). Multipliers are 18x18 bit hard multipliers.

Recall the software wrapper is a term used to describe an Iris-FPGA software component that represents and manages one CPSS (see Section 2.2.3). It manages all interactions with the hardware, hiding the details from the remainder of the system. The software wrapper is significantly simpler than for the refined architecture presented in Section 2.2.3, due to the choice of not using shared memories in this first design. It was implemented manually, not employing the hardware composition tool outlined in Section 2.2.4. It performs the algorithm outlined in the following pseudo-code:

```
1 for each inputbyte do
2   FPGAWriteReg(DATA_IN, inputbyte)
3   FPGAWriteReg(START_REG, 1)
4   wait until FPGAReadReg(DONE_REG)=1
5   for each output_register do
6     outputSamp[i]=FPGAReadReg(DATA_OUT[i])
7   done
8 done
```

where the functions FPGAWriteReg and FPGAReadReg write to and read from a memory-mapped register, respectively.

After the software has written a byte of data to the input register (i.e., 8 symbols), it issues a “start” command by writing to the control register (i.e., the START register in Figure C.1). The hardware awaits this signal and on receiving it begins to process data. Once the hardware has finished processing the input data, this is signalled to the software by writing to the DONE register. The software waits for the hardware to finish its execution by polling the DONE register (line 4 in the above pseudo-code). Finally, the software wrapper reads the array of sample values. In this setup, the hardware generates 32 output samples for one input byte.

When testing the performance of the system it was found that about 95 % of the processing time on the FPGA is spent moving data to and from the CPSS. The processor utilisation on the board is about 90 %. Thus, moving the data one by one using a register interface, as used

in this demonstrator, is inefficient and would not allow more complex setups to execute in real-time. Therefore it was decided to alter the design of the CPSS to use embedded memories for the input and output of data from hardware for future demonstrators, as described in the virtual architecture presented in Section 2.2.2.

C.2 Video Streaming Link with Spectrum Sensing

This section gives design and implementation details for the demonstrator presented in Section 3.2.

Due to the higher performance required for video streaming and based on the experience from the previous experiment (see Section 3.1), the architecture of the CPSS has been altered to use embedded memories for the data transfers between hardware and software. This experiment led to the final virtual architecture design as presented in Section 2.2.2.

At the transmitter, the convolutional coder intellectual property core from Xilinx [127] is used for coding, a core that can be instantiated for a variety of different convolutional codes. It was decided to stay with a simple $1/2$ rate code with constraint length 7, as this is a good trade-off between the achieved bit error rate (BER) performance and the hardware resources needed. All other components at the transmitter have been coded manually using VHDL.

The Viterbi decoder in the receiver chain uses the Xilinx Viterbi Decoder intellectual property core [79]. This is the component with the highest complexity in the chain, it could not have been implemented in software on the PowerPC for real-time processing. Even the processors in laptops and PCs struggle to decode a constraint length 7 convolutional code in real-time, although this depends on how the implementation is optimised. However, the Viterbi algorithm lends itself to implementations on parallel architectures and therefore the FPGA implementation can achieve the required data rates without difficulties. This is one demonstration for the benefits of using an FPGA-based platform for cognitive radio experiments.

The two modes of the receiver are configured into one CPSS using dynamic partial reconfiguration of the FPGA, a feature offered by Xilinx FPGAs which allows to reconfigure parts of the FPGA while others continue to function. Switching the modes using this feature results in less resources needed at any one time and allows reconfiguring the FPGA at run-time. The software wrapper component handles the reconfiguration and completely hides this from the radio controller. The controller itself only switches between the two chains by selecting the current mode of operation, a parameter of the software wrapper. This configures the CPSS accordingly,

	PSS	Transmitter	Reception	Detection
Slices	6,691	1,424	5,354	5,828
(% full)	98 %	21 %	78 %	85 %
BlockRAMs	13	2	6	29
Multipliers	0	24	37	22

Table C.2: FPGA utilisation for the sensing rendezvous experiment. BlockRAMs are 18 Kb embedded memories, slices are the basic measurement of logic utilisation (a Virtex-II slice contains two 4-input lookup tables and two flip-flops), and multipliers are 18x18 bit embedded multipliers. The FPGA is divided into two equally-sized parts for the PSS and CPSS, each containing 6,848 slices.

hiding the details from the controller. This allows implementing the decision-making without knowing the low-level details of the platform.

The resource utilisation for this demonstrator is given in Table C.2. As can be seen, the receiver is significantly more complex than the transmitter. The largest fraction of the area is occupied by the Viterbi decoder component, which takes approximately 2,040 slices. This is about 38 % of the full receiver chain.

C.3 Video Streaming Link with Adaptive Coding

This appendix gives some experimental results and details the benefits of applying the adaptive coding scheme for the demonstrator described in Section 3.3. Benefits in hardware utilization, power consumption, and communications performance are given.

FPGA Resource Utilisation

Table C.3 shows the resource utilisation of the CPSS with the various chain configurations. While the resource requirements of the transmission chains are almost constant, the area savings when reducing the constraint length at the receiver are clear. The CPSS area overhead consists of the two memories used to transfer data, along with the register interface and minimal logic used to interface to the bus (see Section 2.2.2 for the CPSS configuration).

It should be noted that implementing all three receiver processing configuration on a single device without partial reconfiguration would require a larger device. It can be seen that in this case, the receive chain with constraint length 8 Viterbi decoder is close to filling the capacity of the CPSS. Without partial reconfiguration, all coding schemes would need to be present at

	PSS	Transmitter			Receiver		
constr. length	N/A	none	6	8	none	6	8
Slices	6,691	1,426	1,433	1,435	2,594	3,693	6,555
(% full)	98 %	21 %	21 %	21 %	38 %	54 %	95 %
BlockRAMs	13	8	8	8	10	12	14
Multipliers	0	24	24	24	41	41	41

Table C.3: FPGA resource utilisation for the adaptive coding experiment. Both the PSS and CPSS occupy half of the FPGA, i.e., 6,848 slices. The percentages indicate how much of this space is utilised. Slices are the basic measurement of logic utilisation (a Virtex-II slice contains two 4-input lookup tables and two flip-flops). BlockRAMs are 18 Kb embedded memories, Multipliers are 18x18 bit embedded multipliers.

	Sampling Rate ^a [MHz]		Bitrate ^b [Mbit/s]	
constr. length	none	6 or 8	none	6 or 8
Tx HW max ^c	100	100	50	25
Rx HW max ^c	25	25	12.5	6.25
Demonstrator ^d	1	2	0.5	0.5

^aRate of 32-bit samples (16-bit complex) sent to the USRP.
^bRate at the framer/deframer (physical layer bitrate).
^cMaximum of FPGA hardware chain (from FPGA simulation).
^dRate chosen for video demonstrator.

Table C.4: Performance of adaptive coding experiment. The system bus is clocked at 100 MHz, which explains the the maximum sampling rate achievable.

the same time, with a switch to select the active scheme. This would not be possible on the FPGA used for this demonstrator due to a lack of space. Using dynamic partial reconfiguration allows to adapt the coding scheme at run-time on a smaller device than what would be needed without.

Communication Performance

Table C.4 shows both the maximum performance of the signal processing hardware and the sampling and bit rates chosen for this experiment setup. In the table, the maximum performance is the rate achievable by the hardware chain on its own, neglecting the interface to the software and to the radio front-end. The maximum hardware sampling rate is the clock speed of the system bus, i.e., 100 MHz, which the transmitter achieves. The receiver is slower due to the feedback loops in the phase-locked loops (PLLs) of the timing recovery and carrier recovery

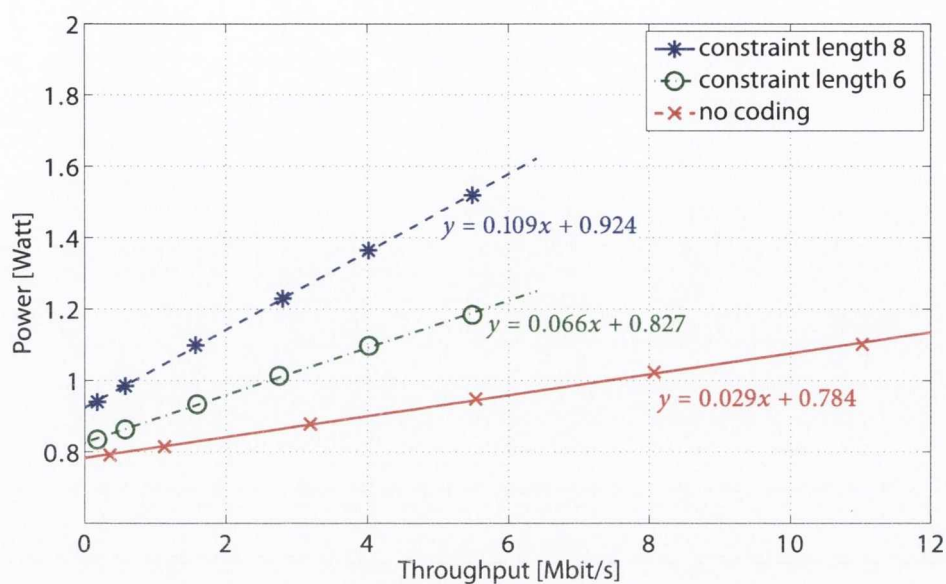


Figure C.2: Measured FPGA power consumption of adaptive coding experiment, with fitted linear regression curves.

components. These components need to use the flow control signals (see Section 2.2.2) to stop new data from coming in while the feedback loop is computing.

In this demonstrator the rates are primarily limited by the bandwidth of the Ethernet connection used to connect to the radio front-end. This limits the maximum achievable bit rate to 500 kbit/s¹. Clearly if the demonstrator is executed on a more advanced development board with built-in radio front-end, the throughput can be drastically improved, up to a maximum of 6.25 Mbit/s with this radio (see Table C.4).

FPGA Power Consumption

In order to quantify the power benefits of adapting the coding scheme, the average power consumption of the three receiver radio chain configurations has been measured (i.e., no coding and coding with constraint lengths 6 and 8). To obtain these measurements, a power supply with integrated ammeter was connected to the FPGA’s internal core voltage (i.e., $V_{ccInt} = 1.5\text{ V}$). Figure C.2 shows the power consumption when increasing the throughput for the complete FPGA (i.e., PSS and specific CPSS).

¹ Since each complex sample is represented by two 16-bit numbers, a 2 MHz sampling rate represents 64 Mbit/s of data over the Ethernet, close to its theoretical capacity of 100 Mbit/s. With 4 samples per symbol, differential quadrature phase shift keying (DQPSK), and 1/2 code rate, the 2 MHz gives a bitrate of 500 kbit/s at the framer/de-framer.

It is apparent that the power consumption increases linearly with throughput. Additionally, the graph shows the large impact the coding constraint length has on power consumption, given the increased amount of FPGA resources required.

Discussion of the results

In this application, coding might be required when there is a noisy channel, yet when channel conditions are more favourable, switching off coding offers a power saving at the receiver. For applications that require a constant bitrate, such as video streaming, this means the sampling rate can be halved at the radio front-end when no coding is used (due to the 1/2 code rate). This results in less radio spectrum occupancy and therefore reduces noise and possible interference in the signal. So it is preferable to use non-coded transmission when possible. Hence adaptation offers both power and system performance advantages in this scenario.

From Figure C.2, the power reduction benefits of run-time adaptation to channel conditions (i.e., signal to noise ratio (SNR)) can be quantified using an illustrative scenario. Assume that in order to provide the required BER in a high noise environment, a radio chain with coding of constraint length 8 is used. As can be seen in the figure, at a throughput of 6 Mbit/s, the constraint length 8 implementation consumes approximately 1.6 W. If the received SNR improves to the point where a non-coding radio chain implementation can provide the required BER, an adaptive reconfiguration can reduce power consumption to under 1 W, a saving of over 60 %.

In this figure, only the resultant *dynamic* power reduction through adaptation at the application level has been considered. It must be also noted that using FPGA partial reconfiguration also contributes to reducing the *static* power consumption, given that functionality can be time-multiplexed on the same physical resources, enabling the use of a smaller FPGA, hence reducing the static power.

C.4 Combined Adaptive Coding and Spectrum Sensing

This section gives implementation details of the combined adaptive coding and spectrum sensing demonstrator presented in Section 3.4.

All radio components could be re-used from the other experiments without modifications, except for the encoder and decoder components, where the range of codes was changed to the codes with constraint lengths 5, 7, and 9. Since both the encoder and decoder were instantiated

	Flip Flops	LUTs	BlockRAMs	DSP48
FPGA capacity	44,800	44,800	148	138
system	28,370	27,045	57	83
—PSS	8,789	10,969	27	13
—Transmitter	1,841	1,993	0	28
—Conv. Enc. 5	9	4	0	0
—Conv. Enc. 7	11	4	0	0
—Conv. Enc. 9	13	4	0	0
—Other comp.	1,808	1,981	0	28
—Receiver	10,779	9,154	16	25
—Viterbi Dec. 5	627	496	1	0
—Viterbi Dec. 7	1,737	1,377	2	0
—Viterbi Dec. 9	6,177	4,960	8	0
—Other comp.	2,238	2,321	5	25
—Detector	6,961	4,929	14	17

Table C.5: FPGA resource utilisation for the combined adaptive coding and sensing experiment. The FPGA is configured to hold the PSS and all possible configurations for the CPSS at once (since no partial reconfiguration is used). The tree on the left illustrates the hierarchy. LUTs are 6-input look-up tables, the basic element to realise a logic function. BlockRAMs are 36 Kb embedded memories, DSP48 are elements for signal processing computations, containing a 25x18 bit multiplier, an adder, and an accumulator.

as Xilinx intellectual property cores, where parameters can be changed in a graphical user interface (GUI), this change was simple and fast.

The hardware resources used are shown in Table C.5. The dynamic partial reconfiguration feature of the FPGA has not been used for this demonstrator due to insufficient tool support at the time of development. Therefore, all possible configurations for the CPSS are placed on the FPGA in parallel, alongside to the PSS. In fact, three CPSS instances have been created, for the transmitter, receiver, and detector chains. Within the transmitter and receiver chains, switching the coding scheme is realised by a switch in hardware, controlled via a register that can be configured from the software warp-per component. This allows selecting any possible configuration for the transmitter and receiver.

If tool support for partial reconfiguration is added, the different chains can be reconfigured at run-time, reducing power consumption and resource utilisation of the device. This frees

FPGA area which can be used for more configurations or more complicated cognitive radios.

D Kahn Process Networks: Continuous and Monotonic Functions

Kahn process network (KPN) [96] require that a mapping function executed by a Kahn process is continuous. This appendix defines the term *continuous* function in the KPN sense. The notation and formalism used here is based on [96, 97].

Let $X = [x_1, x_2, \dots]$ denote a (possibly infinite) sequence of data tokens, where each x_i is a data token. An ordering of sequences is defined as follows. The sequence X is a prefix of (or is equal to) the sequence Y if X is fully contained in Y and Y starts with X . This is denoted as $X \sqsubseteq Y$. For instance, $[x_1, x_2] \sqsubseteq [x_1, x_2, x_3]$. It is said that X approximates Y if $X \sqsubseteq Y$. The empty sequence $\perp$ is therefore a prefix of any other sequence.

Now a possibly infinite *increasing chain of sequences* can be defined.

Definition D.1. An ordered set of sequences $\chi = \{X_0, X_1, X_2, \dots\}$ is an *increasing chain of sequences* if and only if the following condition holds: $X_0 \sqsubseteq X_1 \sqsubseteq X_2 \sqsubseteq \dots$

Such an increasing chain has at least one upper bound Y , where $X_i \sqsubseteq Y$ for all $X_i \in \chi$.

Definition D.2. The *least upper bound* $\sqcap \chi$ of an increasing chain of sequences χ is an upper bound such that for any other upper bound Y , the following condition holds: $\sqcap \chi \sqsubseteq Y$.

The least upper bound may be an infinite sequence.

A *complete partial order* is the set S containing all finite and infinite sequences. The prefix order defines the ordering in the set. This set is “complete” in the sense that every increasing chain has a least upper bound which is also an element of S . Now, let S^p denote the set of

p -tuples of sequences, for instance $\mathbf{X} = (X_1, X_2, \dots, X_p) \in S^p$. Further, the set $\perp \in S^p$ is the set of empty sequences.

An ordering for such sets of sequences can be defined as $\mathbf{X} \sqsubseteq \mathbf{X}'$ if $X_i \sqsubseteq X'_i$ for $1 \leq i \leq p$. A set $\chi = \{\mathbf{X}_0, \mathbf{X}_1, \dots\}$, where each $\mathbf{X}_i$ is a p -tuple of sequences, always has a *greatest lower bound* $\sqcap \chi$, which might be $\perp$. It may or may not have a *least upper bound* $\sqcup \chi$. If it is an increasing chain, i.e., $\chi = \{\mathbf{X}_0, \mathbf{X}_1, \dots\}$ where $\mathbf{X}_0 \sqsubseteq \mathbf{X}_1 \sqsubseteq \dots$, then the least upper bound $\sqcup \chi$ exists. Thus, S^p is a complete partial order for any integer p .

Kahn defines a *functional process* as a mapping from a set of input sequences to a set of output sequences: $F : S^p \rightarrow S^q$ [96]. It will map an increasing chain of sets of sequences χ to another set of sequences Ψ , which may or may not be increasing.

Definition D.3. Let $\sqcup \chi$ denote the least upper bound of the increasing chain χ . A mapping F is said to be *continuous* if for all such chains χ , the least upper bound $\sqcup F$ exists and $F(\sqcup \chi) = \sqcup F(\chi)$.

Kahn shows in [96] that networks of continuous processes are monotonic. A process F is monotonic if $\mathbf{X} \sqsubseteq \mathbf{X}' \Rightarrow F(\mathbf{X}) \sqsubseteq F(\mathbf{X}')$. That means that given an increasing chain χ , a monotonic process maps this set to another increasing chain Ψ . Also note that a network of all monotonic processes is monotonic as a whole.

E The Iris2-FPGA XML Format

To incorporate more advanced dataflow graph structures and to include the Iris engines (see Section 6.3.2 on page 174, the format of the Extensible Markup Language (XML) files for Iris2-FPGA has changed significantly. Now the from the links between components are listed explicitly and separately from the components. One example XML file is shown in Figure E.1

Figure E.1 shows a radio which reads image data from a file, modulates the data, adds white Gaussian noise, demodulates, and compares with the original data to compute the bit error rate (BER). It is functionally the same radio as presented in Figure A.3 on page 244 for the first version of Iris. The XML shows a list of components, encapsulated within two process network (PN) engines, and a list of links between these components. All input and output ports of the components are named, so that they can be referred to in the links.

The general XML syntax for Iris2-FPGA is described by the tree shown in Figure E.2. The usage of most the tags in the figure is demonstrated by the example in Figure E.1. The example radio does not make use of the `<controller>` tag, which is used to specify a radio controller to be used with the radio. Multiple controllers may be used.

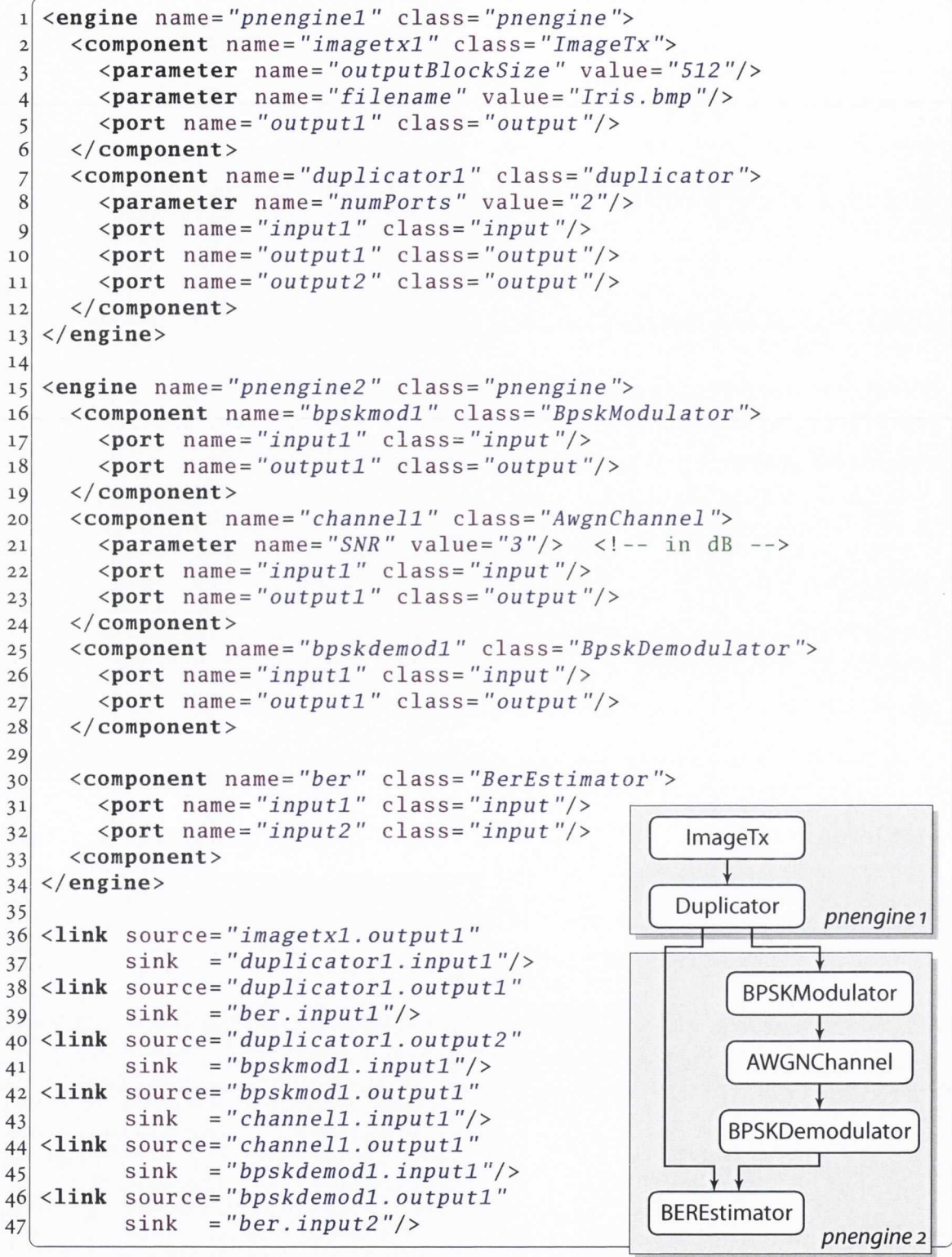


Figure E.1: Example Iris2-FPGA XML file with branching and merging.

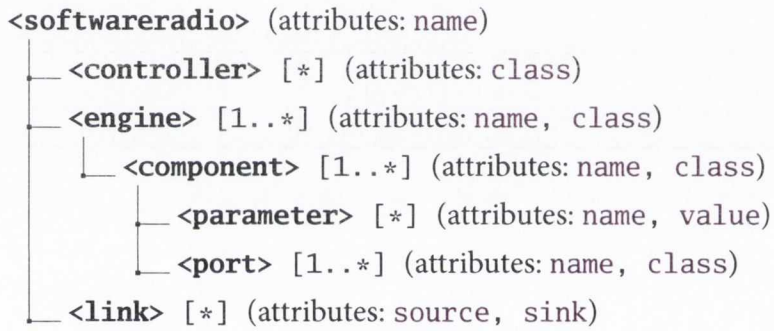


Figure E.2: The Iris 2 XML syntax. A node stands for an XML tag, the numbers in [] denote the multiplicity (none given = exactly once; [0..1] = optionally once; [1..*] = once or more; [*] = any number, incl. zero).

Bibliography

- [1] J. MITOLA III and G. Q. MAGUIRE, JR., "Cognitive radio: Making software radios more personal," *IEEE Personal Communications*, vol. 6, no. 4, pp. 13–18, 1999.
- [2] J. MITOLA III, "Cognitive radio: An integrated agent architecture for software defined radio," Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, May 2000.
- [3] *IEEE Standard Definitions and Concepts for Dynamic Spectrum Access: Terminology Relating to Emerging Wireless Networks, System Functionality, and Spectrum Management*, IEEE Std. 1900.1-2008, Sep. 2008.
- [4] T. W. RONDEAU, "Application of artificial intelligence to wireless communications," Ph.D. dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, Sep. 2007.
- [5] R. W. THOMAS, D. H. FRIEND, L. A. DASILVA, and A. B. MACKENZIE, "Cognitive networks: adaptation and learning to achieve end-to-end performance objectives," *IEEE Communications Magazine*, vol. 44, no. 12, pp. 51–57, 2006.
- [6] *IEEE Standard for Information Technology-Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999), Jun. 2007.
- [7] 3RD GENERATION PARTNERSHIP PROJECT (3GPP), "Technical Specification Group Radio Access Network; Physical layer - General description," Sep. 2009, TS 25.201 V9.0.0.
- [8] J. G. ANDREWS, A. GHOSH, and R. MUHAMED, *Fundamentals of WiMAX – Understanding Broadband Wireless Networking*, ser. Communications Engineering and Emerging Technologies, T. S. RAPPAPORT, Ed. New Jersey, USA: Prentice Hall, 2007.
- [9] S. HAYKIN, "Cognitive radio: brain-empowered wireless communications," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 2, pp. 201–220, Feb. 2005.
- [10] C. CLANCY, J. HECKER, E. STUNTEBECK, and T. O'SHEA, "Applications of machine learning to cognitive radio networks," *IEEE Wireless Communications Magazine*, vol. 14, no. 4, pp. 47–52, 2007.

- [11] N. BALDO, N. BALDO, and M. ZORZI, "Cognitive network access using fuzzy decision making," in *IEEE International Conference on Communications (ICC)*, 2007, pp. 6504–6510.
- [12] O. A. DOBRE, A. ABDI, Y. BAR-NESS, and W. SU, "Blind modulation classification: a concept whose time has come," in *IEEE/Sarnoff Symposium on Advances in Wired and Wireless Communication*, 2005, pp. 223–228.
- [13] K. V. DAVIS, "JTRS-an open, distributed-object computing software radio architecture," in *Digital Avionics Systems Conference (DASC)*, vol. 2, St. Louis, Missouri, 24–29 Oct. 1999, pp. 9.A.6–1 – 9.A.6–8.
- [14] JOINT TACTICAL RADIO SYSTEM (JTRS) JOINT PROGRAM EXECUTIVE OFFICE (JPEO), "Software communications architecture specification," JTRS Standards, final, version 2.2.2, 15 May 2006. [Online]. Available: <http://jtrs.spawar.navy.mil/sca/downloads.asp?ID=2.2.2>
- [15] P. A. EYERMANN and M. A. POWELL, "Maturing the software communications architecture for JTRS," in *IEEE Military Communications Conference (MILCOM)*, vol. 1, Washington, DC, USA, 28–31 Oct. 2001, pp. 158–162.
- [16] *IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7*, IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004), Jan. 2008.
- [17] JOINT TACTICAL RADIO SYSTEM (JTRS) JOINT PROGRAM OFFICE, "Extension for component portability for specialized hardware processors (SHP) to the JTRS software communications architecture (SCA) specification," JTRS Standards, v. 3.1x, 17 Mar. 2005. [Online]. Available: <http://www.mc.com/uploadedFiles/SCA%20Component%20Portability%20Spec%20-%20CP289.pdf>
- [18] L. LAND, "SCA framework puts a CORBA face on FPGAs," *COTS Journal*, Oct. 2004. [Online]. Available: <http://www.cotsjournalonline.com/home/article.php?id=100207>
- [19] *Objective Interface's ORBexpress FPGA for Xilinx Virtex FPGAs*, Objective Interface Systems, Inc., Herndon, VA, USA, May 2007, data sheet. [Online]. Available: http://www.ois.com/images/stories/datasheets/DS_ORBexpressFPGA_F0507.pdf
- [20] *ORBexpress DSP*, Objective Interface Systems, Inc., Herndon, VA, USA, Oct. 2007, data sheet. [Online]. Available: http://www.ois.com/images/stories/datasheets/DS_ORBexpressDSP_F1007.pdf
- [21] C. R. A. GONZALEZ, 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," *IEEE Communications Magazine*, vol. 47, no. 10, pp. 48–55, Oct. 2009.
- [22] M. CARRICK, D. CORMIER, C. COVINGTON, C. B. DIETRICH, J. GAEDDERT, B. HILBURN, C. I. PHELPS, S. SAYED, D. SEERALAN, J. SNYDER, and H. VOLOS, *OSSIE 0.8.0 Installation and User Guide*, Wireless@Virginia Tech, Virginia Polytechnic

- Institute and State University, Blacksburg, VA, USA, Sep. 2009. [Online]. Available: http://ossie.wireless.vt.edu/download/user_guides/OSSIE_o.8.o_User_Guide.pdf
- [23] OmniORB website. Apasphere Ltd. Cambridge, UK. [Online]. Available: <http://omniorb.sourceforge.net>
- [24] M. ROBERT, S. SAYED, C. AGUAYO, R. MENON, K. CHANNAK, C. VANDER VALK, C. NEELY, T. TSOU, J. MANDEVILLE, and J. H. REED, "OSSIE: Open source SCA for researchers," in *SDR Forum Technical Conference and Product Exposition (SDR Forum)*, Phoenix, Arizona, USA, Nov. 2004.
- [25] C. R. A. GONZALEZ, F. M. PORTELINHA, and J. H. REED, "Part 1: Design and implementation of an SCA core framework for a DSP platform," *Military Embedded Systems*, Mar. 2007. [Online]. Available: <http://www.mil-embedded.com/pdfs/VirginiaTech.Mar07.pdf>
- [26] C. R. A. GONZALEZ, F. M. PORTELINHA, and J. H. REED, "Part 2: Design and implementation of an SCA core framework for a DSP platform," *Military Embedded Systems*, May 2007. [Online]. Available: <http://www.mil-embedded.com/pdfs/VirginiaTech.May07.pdf>
- [27] P. J. BALISTER, M. ROBERT, and J. H. REED, "Impact of the use of CORBA for inter-component communication in SCA based radio," in *SDR 06 Technical Conference and Product Exposition*, 2006. [Online]. Available: http://www.sdrforum.org/pages/sdro6/sdro6_papers/1.1/1.1-4.pdf
- [28] M. CARRICK, S. SAYED, C. DIETRICH, and J. REED, "Integration of FPGAs into SDR via memory-mapped i/o," in *SDR Forum Technical Conference and Product Exposition (SDR)*, Washington, D.C., USA, Dec. 2009.
- [29] R. SATHAPPAN, M. DUMAS, L. BELANGER, and M. UHM, "New architecture for development platform targeted to portable applications," in *SDR Forum Technical Conference (SDR)*, 2006.
- [30] J. JACOB and M. DUMAS, "CORBA for FPGA the missing link for SCA radios," Lyrtech, Quebec, Canada, white paper, Feb. 2007.
- [31] *System Generator for DSP: User Guide*, Xilinx, Inc., San Jose, CA, USA, 19 Apr. 2010, UG640 v12.1. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/sysgen_user.pdf
- [32] PrismTech website. PrismTech Ltd. [Online]. Available: <http://www.prismtech.com>
- [33] Spectrum Signal Processing by Vecima. Burnaby, Canda. [Online]. Available: <http://www.spectrumsignal.com>
- [34] T. ULVERSØY and J. O. NESET, "On workload in an SCA-based system, with varying component and data packet sizes," NATO, report RTO-MP-IST-o83, 2008.
- [35] E. A. LEE and D. G. MESSERSCHMITT, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

- [36] FREE SOFTWARE FOUNDATION, INC. (2008) GNU radio - the GNU software radio. [Online]. Available: <http://www.gnu.org/software/gnuradio/>
- [37] E. BLOSSOM. (2004, Nov.) Exploring GNU radio. Initial version published in Linux Journal, Issue 122, June 2004, as "GNU Radio: Tools for Exploring the RF Spectrum.". [Online]. Available: <http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html>
- [38] (2008) Cognitive Wireless Technology (CWT). Virginia Polytechnic Institute and State University. [Online]. Available: <http://www.cognitiveradio.wireless.vt.edu>
- [39] P. MACKENZIE, "Reconfigurable software radio systems," Ph.D. dissertation, University of Dublin, Trinity College, Ireland, Oct. 2004.
- [40] K. E. NOLAN, P. D. SUTTON, L. E. DOYLE, T. W. RONDEAU, B. LE, and C. W. BOSTIAN, "Dynamic spectrum access and coexistence experiences involving two independently developed cognitive radio testbeds," in *2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, 17–20 Apr. 2007, pp. 270–275.
- [41] P. D. SUTTON, K. E. NOLAN, and L. E. DOYLE, "Cyclostationary signatures for rendezvous in OFDM-based dynamic spectrum access networks," in *2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, 17–20 Apr. 2007, pp. 220–231.
- [42] D. SARATH, K. E. NOLAN, P. D. SUTTON, and L. E. DOYLE, "Exploring the reconfigurability options of multi-carrier CDMA in cognitive radio systems," in *IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 3–7 Sep. 2007, pp. 1–5.
- [43] G. J. MINDEN, J. B. EVANS, L. SEARL, D. DEPARDO, V. R. PETTY, R. RAJBANSHI, T. NEWMAN, Q. CHEN, F. WEIDLING, J. GUFFEY, D. DATLA, B. BARKER, M. PECK, B. CORDILL, A. M. WYGLINSKI, and A. AGAH, "KUAR: A flexible software-defined radio development platform," in *2nd IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, Dublin, Ireland, 17–22 Apr. 2007.
- [44] *ISE Design Suite Product Brief*, Xilinx Inc., San Jose, CA, USA, Apr. 2010. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/pn0010867.pdf
- [45] G. MINDEN, J. EVANS, L. SEARL, D. DEPARDO, R. RAJBANSHI, J. GUFFEY, Q. CHEN, T. NEWMAN, V. PETTY, F. WEIDLING, M. PECK, B. CORDILL, D. DATLA, B. BARKER, and A. AGAH, "Cognitive radios for dynamic spectrum access - an agile radio for wireless innovation," *Communications Magazine, IEEE*, vol. 45, no. 5, pp. 113–121, May 2007.
- [46] J. D. GUFFEY, A. M. WYGLINSKI, and G. J. MINDEN, "Agile radio implementation of OFDM physical layer for dynamic spectrum access research," in *IEEE Global Telecommunications Conference (GLOBECOM)*, 26–30 Nov. 2007, pp. 4051–4055.

- [47] R. REED, "Implementation of a BPSK transceiver for use with the University of Kansas Agile Radio," BSc thesis, Department of Electrical Engineering and Computer Science, Faculty of the Graduate School of the University of Kansas, Lawrence, Kansas, USA, 2006. [Online]. Available: www.ittc.ku.edu/research/thesis/documents/ryan_reed_thesis.pdf
- [48] Z. MILJANIC, I. SESKAR, K. LE, and D. RAYCHAUDHURI, "The WINLAB network centric cognitive radio hardware platform – WiNC2R," *Mobile Networks and Applications*, vol. 13, no. 5, pp. 533–541, Oct. 2008.
- [49] RUTGERS WINLAB. (2009) Network Centric Cognitive Radio Platform (WiNC2R). [Online]. Available: <http://www.winlab.rutgers.edu/docs/focus/WiNC2R.html>
- [50] S. SRIKANTESWARA, J. H. REED, P. ATHANAS, and R. BOYLE, "A soft radio architecture for reconfigurable platforms," *IEEE Communications Magazine*, vol. 38, no. 2, pp. 140–147, Feb. 2000.
- [51] J. NEEL, "Simulation of an implementation and evaluation of the layered radio architecture," MSc thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, Dec. 2002. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-01092003-162405/unrestricted/jn_thesis_final.pdf
- [52] K. AMIRI, Y. SUN, P. MURPHY, C. HUNTER, J. R. CAVALLARO, and A. SABHARWAL, "WARP, a unified wireless network testbed for education and research," in *IEEE International Conference on Microelectronic Systems Education (MSE)*, 3–4 Jun. 2007, pp. 53–54.
- [53] (2009) WARP FPGA Board – Hardware Version 2.2. Mango Communications, Inc. [Online]. Available: <http://mangocomm.com/products/boards/warp-fpga-board-v2>
- [54] *IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Broadband Wireless Access Systems*, IEEE Std. 802.16-2009 (Revision of 802.16-2004), 2009.
- [55] *IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Broadband Wireless Access Systems Amendment 2: Improved Coexistence Mechanisms for License-Exempt Operation*, IEEE Std. 802.16h-2010 (Amendment to 802.16-2009), 2010.
- [56] *MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 10.1i*, Xilinx, Inc., San Jose, CA, USA, 17 Jan. 2008, UGo81 (v9.0). [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf
- [57] *XPS HWICAP: Product Specification*, Xilinx, Inc., San Jose, CA, USA, 16 Sep. 2009, v3.00a. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/xps_hwicap.pdf
- [58] *Programming Languages - C++*, JTC1/SC22/WG21 - The C++ Standards Committee, Std. INCITS/ISO/IEC 14 882-2003, 2003.
- [59] G. V. VAUGHAN, B. ELLISTON, T. TROMLEY, and I. L. TAYLOR, *GNU Autoconf, Automake, and Libtool*. Pearson Education, Oct. 2000. [Online]. Available: <http://sourceware.org/autobook/>

- [60] *Xilinx University Program Virtex-II Pro Development System – Hardware Reference Manual*, Xilinx, Inc., San Jose, CA, USA, 8 Mar. 2005, UG069 v1.0. [Online]. Available: http://www.xilinx.com/univ/XUPV2P/Documentation/XUPV2P_User_Guide.pdf
- [61] *ML505/ML506/ML507 Evaluation Platform: User Guide*, Xilinx, Inc., San Jose, CA, USA, 7 Oct. 2009, UG347 v3.1.1. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf
- [62] D. KEGEL. (2006) crosstool: Building and testing gcc/glibc cross toolchains. [Online]. Available: <http://www.kegel.com/crosstool>
- [63] Xilinx Git repository. Xilinx, Inc. [Online]. Available: [git://git.xilinx.com](http://git.xilinx.com)
- [64] D. VLASENKO *et al.* (2010) BusyBox: The Swiss Army Knife of Embedded Linux. [Online]. Available: <http://www.busybox.net/about.html>
- [65] *Universal Software Radio Peripheral – The Foundation for Complete Software Radio Systems*, Ettus Research LLC, Mountain View, California, USA, Nov. 2006. [Online]. Available: http://www.ettus.com/downloads/usrp_v4.pdf
- [66] J. LOTZE, S. A. FAHMY, J. NOGUERA, L. E. DOYLE, and R. ESSER, “High-level cognitive radio design using Xilinx FPGAs,” in *1st Collaborative International Software Defined Radio Workshop*, Maynooth, Ireland, May 2008.
- [67] J. LOTZE, S. A. FAHMY, J. NOGUERA, L. E. DOYLE, and R. ESSER, “An FPGA-based cognitive radio framework,” in *Irish Signals and Systems Conference (ISSC)*, Galway, Ireland, Jun. 2008, pp. 138–143.
- [68] J. LOTZE, B. ÖZGÜL, S. A. FAHMY, J. NOGUERA, L. E. DOYLE, and R. ESSER, “Spectrum sensing to achieve frequency rendezvous using Xilinx FPGAs,” in *IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, Chicago, USA, Oct. 2008, demonstration paper.
- [69] J. LOTZE, S. A. FAHMY, J. NOGUERA, B. OZGUL, L. DOYLE, and R. ESSER, “Development framework for implementing FPGA-based cognitive network nodes,” in *IEEE Global Communications Conference, Exhibition, and Industry Forum (GLOBECOM)*, Honolulu, Hawaii, USA, Dec. 2009.
- [70] S. A. FAHMY, J. LOTZE, J. NOGUERA, L. E. DOYLE, and R. ESSER, “Generic software framework for adaptive applications on FPGAs,” in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, Apr. 2009, pp. 55–62.
- [71] J. LOTZE, S. A. FAHMY, L. E. DOYLE, and J. NOGUERA, “An FPGA-based autonomous adaptive radio,” in *ACM SIGCOMM Conference*, Barcelona, Spain, Aug. 2009, demonstration paper.
- [72] J. P. COSTAS, “Synchronous communications,” *Proceedings of the IRE*, vol. 44, no. 12, pp. 1713–1718, 1956.
- [73] J. FEIGN, “Practical Costas loop design,” *RF Signal Processing*, pp. 20–36, Jan. 2002. [Online]. Available: <http://mobiledevdesign.com/images/archive/0102Feigin20.pdf>

- [74] G. R. DANESFAHANI and T. G. JEANS, "Optimisation of modified Mueller and Muller algorithm," *Electronics Letters*, vol. 31, no. 13, pp. 1032–1033, Jun. 1995.
- [75] T. YUCEK and H. ARSLAN, "A survey of spectrum sensing algorithms for cognitive radio applications," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 1, pp. 116–130, Mar. 2009.
- [76] A. GHASEMI and S. E. SOUSA, "Spectrum sensing in cognitive radio networks: Requirements, challenges, and design trade-offs," *IEEE Communications Magazine*, vol. 46, pp. 32–39, Apr. 2008.
- [77] W. A. GARDNER, "Signal interception: A unifying theoretical framework for feature detection," *IEEE Transactions on Communications*, vol. 36, pp. 897–906, Aug. 1988.
- [78] T. J. O'SHEA, T. C. CLANCY, and H. J. EBEID, "Practical signal detection and classification in GNU radio," in *SDR Forum Technical Conference (SDR)*, Denver, Colorado, USA, Nov. 2007.
- [79] *Viterbi Decoder*, Xilinx, Inc., San Jose, CA, USA, 10 Oct. 2007, DSP247, v6.1. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/viterbi_ds247.pdf
- [80] *ML505/ML506/ML507 Reference Design: User Guide*, Xilinx, Inc., San Jose, CA, USA, 23 Jun. 2009, UG249 v3.1. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug349.pdf
- [81] Xilinx open source wiki. Xilinx, Inc. San Jose, CA, USA. [Online]. Available: <http://xilinx.wikidot.com>
- [82] P. KARN. (2007, 7 Aug.) DSP and FEC Library. [Online]. Available: <http://www.kagq.net/code/fec>
- [83] D. JOHNSON, "Signal processing software tools," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 9, San Diego, CA, USA, 19–21 Mar. 1984, pp. 318–320.
- [84] E. A. LEE, "Embedded software — an agenda for research," University of California at Berkeley, UCB ERL Memorandum M99/63, Dec. 1999.
- [85] 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, Jan. 2003.
- [86] *Simulink 7*, The MathWorks, Inc., Sep. 2007, no. 9320v06. [Online]. Available: <http://www.mathworks.com/mason/tag/proxy.html?dataid=9798&fileid=43815>
- [87] D. SILAGE, *Digital Communication Systems Using SystemVue*. London, UK: Charles River Media, Jan. 2006.
- [88] *Real-Time Workshop 7*, The MathWorks, Inc., Sep. 2007, no. 9400v06. [Online]. Available: <http://www.mathworks.com/mason/tag/proxy.html?dataid=9547&fileid=43808>

- [89] P. D. SUTTON, B. OZGUL, K. E. NOLAN, and L. E. DOYLE, "Bandwidth-adaptive waveforms for dynamic spectrum access networks," in *3rd IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, 14–17 Oct. 2008, pp. 1–7.
- [90] G. NYCHIS, T. HOTTELIER, Z. YANG, S. SESHAN, and P. STEENKISTE, "Enabling MAC protocol implementations on software-defined radios," in *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, 2009, pp. 91–105.
- [91] *ChipScope Pro 12.1: Software and Tools*, Xilinx, Inc., San Jose, CA, USA, Apr. 2010, UG029 v12.1. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/chipscope_pro_sw_cores_ugo29.pdf
- [92] *The Unified Modeling Language (UML)*, Object Management Group Std., Rev. 2.3, May 2010. [Online]. Available: <http://www.omg.org/spec/UML/2.3/>
- [93] P. MILLER. (2001) SISAL Parallel Programming. Sourceforce project site. [Online]. Available: <http://sisal.sourceforge.net/>
- [94] (2009) Single Assignment C. [Online]. Available: <http://www.sac-home.org>
- [95] M. EISENRING, J. TEICH, and L. THIELE, "Rapid prototyping of dataflow programs on hardware/software architectures," in *Hawaii International Conference on System Sciences (HICSS)*, vol. 7, Jan. 6–9, 1998, pp. 187–196.
- [96] G. KAHN, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proceedings of the IFIP Congress*, J. L. ROSENFELD, Ed. New York, USA: North-Holland, 1974, pp. 471–475.
- [97] E. A. LEE and T. M. PARKS, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.
- [98] A. M. TURING, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, pp. 230–265, 1936.
- [99] A. M. TURING, "On computable numbers, with an application to the entscheidungsproblem: A correction," *Proceedings of the London Mathematical Society*, vol. s2-43, pp. 544–546, 1937.
- [100] J. E. HOPCROFT, R. MOTWANI, and J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Addison-Wesley, 2001.
- [101] E. A. LEE and D. G. MESSERSCHMITT, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
- [102] S. RITZ, M. PANKERT, V. ZIVOJINOVIC, and H. MEYR, "Optimum vectorization of scalable synchronous dataflow graphs," in *International Conference on Application-Specific Array Processors (ASAP)*, 25–27 Oct. 1993, pp. 285–296.
- [103] G. BILSEN, M. ENGELS, R. LAUWEREINS, and J. A. PEPPERSTRAETE, "Cyclostatic data flow," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 5, 9–12 May 1995, pp. 3255–3258.

- [104] J. BUCK and E. A. LEE, "The token flow model," in *Data Flow Workshop*, Hamilton Island, Australia, May 1992.
- [105] W. PLISHKER, N. SANE, M. KIEMB, K. ANAND, and S. S. BHATTACHARYYA, "Functional DIF for rapid prototyping," in *IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*, Monterey, CA, USA, 2–5 Jun. 2008, pp. 17–23.
- [106] G. BILSEN, M. ENGELS, R. LAUWEREINS, and J. A. PEPPERSTRAETE, "Static scheduling of multi-rate and cyclo-static DSP-applications," in *Workshop on VLSI Signal Processing*, 26–28 Oct. 1994, pp. 137–146.
- [107] T. M. PARKS, J. L. PINO, and E. A. LEE, "A comparison of synchronous and cycle-static dataflow," in *Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, vol. 1, 30 Oct. – 2 Nov. 1995, pp. 204–210.
- [108] W. PLISHKER, N. SANE, and S. S. BHATTACHARYYA, "A generalized scheduling approach for dynamic dataflow applications," in *Design, Automation & Test in Europe Conference and Exhibition (DATE)*, Nice, France, 20–24 Apr. 2009, pp. 111–116.
- [109] C. A. PETRI, "Communication with automata," Reconnaissance-Intelligence Data Handling Branch, Rome Air Development Center, New York, suppl. I to tech. rep. no. RADC-TR-65-377, Jan. 1966.
- [110] J. L. PETERSON, *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.
- [111] J. L. PETERSON, "Petri nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, Sep. 1977.
- [112] J. LOTZE, S. A. FAHMY, J. NOGUERA, and L. E. DOYLE, "A model-based approach to cognitive radio design," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 2, pp. 455–468, Feb. 2011.
- [113] S. HAYKIN, *Neural Networks: A Comprehensive Foundation*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.
- [114] N. BALDO, B. R. TAMMA, B. S. MANOJT, R. RAO, and M. ZORZI, "A neural network based cognitive controller for dynamic channel selection," in *IEEE International Conference on Communications (ICC)*, 2009, pp. 1–5.
- [115] D. E. GOLDBERG, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
- [116] N. CHAIYARATANA and A. M. S. ZALZALA, "Recent developments in evolutionary and genetic algorithms: theory and applications," in *Second International Conference On Genetic Algorithms in Engineering Systems: Innovations and Applications*, 1997, pp. 270–277.
- [117] S. CHEN, T. R. NEWMAN, J. B. EVANS, and A. M. WYGLINSKI, "Genetic algorithm-based optimization for cognitive radio networks," in *IEEE Sarnoff Symposium*, 2010, pp. 1–6.

- [118] L. R. RABINER, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.
- [119] A. HE, K. K. BAE, T. R. NEWMAN, J. GAEDDERT, K. KIM, R. MENON, L. MORALES-TIRADO, J. NEEL, Y. ZHAO, J. H. REED, and W. H. TRANter, "A survey of artificial intelligence for cognitive radios," *IEEE Transactions on Vehicular Technology*, vol. 59, no. 4, pp. 1578–1592, 2010.
- [120] J. T. BUCK, "Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams," in *Asilomar Conference on Signals, Systems and Computers (ACSSC)*, vol. 1, 1994, pp. 508–513.
- [121] M. C. WILLIAMSON and E. A. LEE, "Synthesis of parallel hardware implementations from synchronous dataflow graph specifications," in *Asilomar Conference on Signals, Systems and Computers (ACSSC)*, 1996, pp. 1340–1343.
- [122] C. CHOI and S. HA, "Software synthesis for dynamic data flow graph," in *IEEE International Workshop on Rapid System Prototyping (RSP)*, 24–26 Jun. 1997, pp. 72–79.
- [123] S. KWON, C. LEE, and S. HA, "Data-parallel code generation from synchronous dataflow specification of multimedia applications," in *IEEE/ACM/IFIP Workshop Embedded Systems for Real-Time Multimedia (ESTIMedia)*, Oct. 2007, pp. 91–96.
- [124] S. TRIPAKIS, D. BUI, M. GEILEN, B. RODIERS, and E. A. LEE, "Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-52, May 2010. [Online]. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-52.html>
- [125] 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," *IEEE Communications Magazine*, vol. 48, no. 9, pp. 114–122, 2010.
- [126] M. SANGRIOTIS and I. XEZONAKIS, "Digital costas loop-like PLL for the carrier recovery of a QPSK signal," *Electronics Letters*, vol. 29, no. 10, pp. 897–899, 1993.
- [127] *Convolution Encoder*, Xilinx, Inc., San Jose, CA, USA, 17 May 2007, DS248 v6.1. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/convolution_ds248.pdf
- [128] A. H. DÖNNI, *The Boost Statechart Library*, Apr. 2010, ver. 1.43, part of the Boost C++ Libraries. [Online]. Available: http://www.boost.org/doc/libs/1_43_0/libs/statechart
- [129] R. W. THOMAS, L. A. DASILVA, and A. B. MACKENZIE, "Cognitive networks," in *IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, Nov. 2005, pp. 352–360.
- [130] *ModelSim SE User's Manual*, Mentor Graphics Corporation, Wilsonville, OR, USA, May 2010, software version 6.6b.

- [131] *Annals of Internal Medicine*, American College of Physicians. [Online]. Available: <http://www.annals.org>
- [132] (2010) Annals of Internal Medicine – Information for Authors: Reproducible Research. American College of Physicians. [Online]. Available: <http://www.annals.org/site/misc/ifora.xhtml#reproducible-research>
- [133] P. GEPNER, D. L. FRASER, M. F. KOWALIK, and R. TYLMAN, “New multi-core Intel Xeon processors help design energy efficient solution for high performance computing,” in *International Multiconference Computer Science and Information Technology (IMCSIT)*, 2009, pp. 567–571.
- [134] P. CONWAY and B. HUGHES, “The AMD Opteron northbridge architecture,” *IEEE Micro*, vol. 27, no. 2, pp. 10–21, 2007.
- [135] P. CONWAY, N. KALYANASUNDHARAM, G. DONLEY, K. LEPAK, and B. HUGHES, “Cache hierarchy and memory subsystem of the AMD Opteron processor,” *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.
- [136] B. STOLT, Y. MITTLEFEHLDT, S. DUBEY, G. MITTAL, M. LEE, J. FRIEDRICH, and E. FLUHR, “Design and implementation of the POWER6 microprocessor,” *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 21–28, 2008.
- [137] D. GREENLEY, J. BAUMAN, D. CHANG, D. CHEN, R. ELTEJAEIN, P. FEROLITO, P. FU, R. GARNER, D. GREENHILL, H. GREWAL, K. HOLDBROOK, B. KIM, L. KOHN, H. KWAN, M. LEVITT, G. MATURANA, D. MRAZEK, C. NARASIMHAIAH, K. NORMOYLE, N. PARVEEN, P. PATEL, A. PRABHU, M. TREMBLAY, M. WONG, L. YANG, K. YARLAGADDA, R. YU, R. YUNG, and G. ZYNER, “UltraSPARC: the next generation superscalar 64-bit SPARC,” in *IEEE Computer Society International Conference (COMPCON) ‘Technologies for the Information Superhighway’*, 1995, pp. 442–451.
- [138] *Dual-Core Intel Xeon Processor 5000 Series*, Intel Corporation, May 2006, datasheet, document no. 313079-001. [Online]. Available: http://www.intel.com/Assets/en_US/PDF/datasheet/313079.pdf
- [139] *TMS320C6455 Fixed-Point Digital Signal Processor*, Texas Instruments Inc., Dallas, Texas, USA, Apr. 2009, rev. 1. [Online]. Available: <http://www.ti.com/lit/gpn/tms320c6455>
- [140] *NVIDIA CUDA C Programming Guide*, NVIDIA Corporation, Santa Clara, CA, USA, 21 Jul. 2010, version 3.1.1. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf
- [141] KHRONOS OPENCL WORKING GROUP, *The OpenCL Specification*, Khronos Group, 6 Oct. 2009, version 1.0, revision 48. [Online]. Available: <http://www.khronos.org/registry/cl/specs/opengl-1.0.48.pdf>
- [142] J. A. KAHLE, M. N. DAY, H. P. HOFSTEE, C. R. JOHNS, T. R. MAEURER, and D. SHIPPY, “Introduction to the Cell multiprocessor,” *IBM Journal of Research and Development*, vol. 49, no. 4, pp. 589–604, 2005.

- [143] M. SCARPINO, *Programming the Cell Processor: For Games, Graphics, and Computation*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.
- [144] (2006) BDTI Communications Benchmark (OFDM)[™] Results. Berkley Design Technology, Inc. (BDTI). [Online]. Available: <http://www.bdti.com/bdtimark/ofdm.htm>
- [145] "FPGAs for DSP, Second Edition," Berkley Design Technology, Inc. (BDTI), focus report, 2006. [Online]. Available: http://www.bdti.com/products/reports_fpga2006.html