ABSTRACT

Title of dissertation: RAPID PROTOTYPING OF HIGH PERFORMANCE SIGNAL PROCESSING APPLICATIONS

Nimish Sane, Doctor of Philosophy, 2011

Dissertation directed by: Shuvra S. Bhattacharyya (Chair/Advisor)
Professor
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies

Andrew Harris (Co-Advisor)
Professor
Department of Astronomy

Advances in embedded systems for digital signal processing (DSP) are enabling many scientific projects and commercial applications. At the same time, these applications are key to driving advances in many important kinds of computing platforms. In this region of high performance DSP, rapid prototyping is critical for faster time-to-market (e.g., in the wireless communications industry) or time-to-science (e.g., in radio astronomy). DSP system architectures have evolved from being based on application specific integrated circuits (ASICs) to incorporate reconfigurable off-the-shelf field programmable gate arrays (FPGAs), the latest multiprocessors such as graphics processing units (GPUs), or heterogeneous combinations of such devices. We, thus, have a vast design space to explore based on performance trade-offs, and expanded by the multitude of possibilities for target platforms.

In order to allow systematic design space exploration, and develop scalable and portable prototypes, model based design tools are increasingly used in design and implementation of embedded systems. These tools allow scalable high-level representations, model based semantics for analysis and optimization, and portable implementations that can be verified at higher levels of abstractions and targeted toward multiple platforms for implementation. The designer can experiment using such tools at an early stage in the design cycle, and employ the latest hardware at later stages. In this thesis, we have focused on dataflow-based approaches for rapid DSP system prototyping. This thesis contributes to various aspects of dataflow-based design flows and tools as follows:

1. We have introduced the concept of *topological patterns*, which exploits commonly found repetitive patterns in DSP algorithms to allow scalable, concise, and parameterizable representations of large scale dataflow graphs in high-level languages. We have shown how an underlying design tool can systematically exploit a high-level application specification consisting of topological patterns in various aspects of the design flow.

2. We have formulated the *core functional dataflow* (*CFDF*) model of computation, which can be used to model a wide variety of deterministic dynamic dataflow behaviors. We have also presented key features of the CFDF model and tools based on these features. These tools provide support for heterogeneous dataflow behaviors, an intuitive and common framework for functional specification, support for functional simulation, portability from several existing dataflow models to CFDF, integrated emphasis on minimally-restricted specification of actor functionality, and

support for efficient static, quasi-static, and dynamic scheduling techniques.

3. We have developed a generalized scheduling technique for CFDF graphs based on decomposition of a CFDF graph into static graphs that interact at run-time. Furthermore, we have refined this generalized scheduling technique using a new notion of "mode grouping," which better exposes the underlying static behavior. We have also developed a scheduling technique for a class of dynamic applications that generates *parameterized looped schedules* (*PLS*s), which can handle dynamic dataflow behavior without major limitations on compile-time predictability.

4. We have demonstrated the use of dataflow-based approaches for design and implementation of radio astronomy DSP systems using an application example of a *tunable digital downconverter* (*TDD*) for spectrometers. Design and implementation of this module has been an integral part of this thesis work. This thesis demonstrates a design flow that consists of a high-level software prototype, analysis, and simulation using the dataflow interchange format (DIF) tool, and integration of this design with the existing tool flow for the target implementation on an FPGA platform, called *interconnect break-out board* (*IBOB*). We have also explored the trade-off between low hardware cost for fixed configurations of digital downconverters and flexibility offered by TDD designs.

5. This thesis has contributed significantly to the development and release of the latest version of a graph package oriented toward models of computation (MoCGraph). Our enhancements to this package include support for tree data structures, and *generalized schedule trees* (*GST*s), which provide a useful data structure for a wide

variety of schedule representations. Our extensions to the MoCGraph package provided key support for the CFDF model, and functional simulation capabilities in the DIF package.

RAPID PROTOTYPING OF HIGH PERFORMANCE SIGNAL PROCESSING APPLICATIONS

by

Nimish Sane

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctory of Philosophy
2011

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Andrew Harris, Co-Advisor
Professor Steven Tretter
Professor Gang Qu
Professor Amitabh Varshney, Dean's representative

© Copyright by
Nimish Sane
2011

Dedication

To my parents and grandparents.

Acknowledgments

I express my sincere gratitude toward my advisor Prof. Shuvra S. Bhattacharyya for his academic, research, and professional guidance throughout my PhD research work. He allowed me to pursue my own research interests with great freedom, and enthusiastically supported all of my efforts. This resulted in our research collaboration with the National Radio Astronomy Observatory (NRAO). His expertise in the field of embedded signal processing and model based design tools helped me smoothly get into and learn about this exciting research field. His advice and insights throughout my PhD greatly enriched my research, and helped steer it to stay on track. I have learned a great deal from his research methods and attention to detail. I am grateful to him for providing the necessary motivation when needed. His patience and understanding ensured that my experience was enjoyable. I thank him for giving me excellent research, teaching, and mentoring opportunities. I will always cherish my experience working with him.

I am also grateful to my co-advisor Prof. Andrew Harris for his advice on the aspects of my research related to astronomy as well as academic and professional guidance. His insights into scientific, technical, and logistical aspects have been crucial in shaping my research goals and professional career. He introduced me to the field of radio astronomy instrumentation, and encouraged me to pursue my PhD research in that field. His support during my internship at the Owens Valley Radio Observatory (OVRO) and collaboration with the NRAO was very crucial. I will be always grateful to him for that.

I also thank other members of my dissertation committee Prof. Steven Tretter, Prof. Gang Qu, and Prof. Amitabh Varshney for their advice and co-operation. Their

constructive comments and feedback were valuable.

The research presented in this thesis was sponsored in part by the National Radio Astronomy Observatory, Agilent Technologies, Inc., Laboratory of Telecommunication Sciences, National Science Foundation, US Air Force Research Laboratory, and Austrian Marshall Plan Foundation. I acknowledge them with thanks for their support.

I am extremely thankful to Dr. Richard Prestage of the NRAO, Green Bank, WV, for his initial support and continued guidance in establishing a collaboration between the NRAO and Maryland Digital Signal Processing — Computer Aided Design (DSPCAD) research group. I am also grateful to Mr. John Ford of the NRAO, Green Bank, WV, for his patient supervision and guidance. He encouraged me to pursue various research directions and provided valuable technical insights. He was very supportive, understanding, and co-operative in dealing with the logistics of the collaboration, which made it a smooth one. I also thank Shilpa Bollineni, Srikanth Bussa, Randy McCullough, Scott Ransom, and Jason Ray of the NRAO for their support.

I would like to thank Dr. Chia-Jui Hsu, and Josè Pino (both of Agilent Technologies, Inc.) for their supervision and collaboration.

I acknowledge Dr. Gunasekaran Seetharaman of the US Air Force Research Laboratory with thanks for his collaboration.

It has been a memorable experience working at the Maryland DSPCAD research group. I am thankful to Dr. William Plishker for his support and guidance throughout my PhD research. I have learned a lot from him while working with him on various collaborative efforts. Some of these efforts have contributed to the research presented in this thesis. I sincerely thank him for the same. I also thank him and Dr. Chung-Ching

Shen for a number of very interesting conversations that we have had on technical and research issues as well as professional career. I also thank Mary Kiemb, Kapil Anand, Dr. Hojin Kee (now at National Instruments, Inc.), and Shenpei Wu from the Maryland DSPCAD group for their contributions.

All the current and past DSPCAD members and friends — Dr. William Plishker, Dr. Chung-Ching Shen, Dr. Sankalita Saha, Dr. Hojin Kee, Dr. Ruirui Gu, Mary Kiemb, Kapil Anand, Sebastian Puthenpurayil, George Zaki, Hsiang-Huang Wu, Soujanya Kedilaya, Inkeun Cho, Scott Kim, Kishan Sudusinghe, and Shenpei Wu — deserve a special mention. They have been of great help throughout my research work in the DSPCAD group. I specially thank to Will, Chung-Ching, Sankalita, and Hojin for their valuable guidance during my initial days as a PhD student. I also thank Will, George, Hsiang-Huang, Shenpei, and Kishan for their help toward the end. I also had an opportunity to mentor some of the high school and undergraduate student members — Saara Khan, Kelly Davis, Chima Ebinama, Giancarlo Bautista, Rainier Gomez, and Cordell Reid — in the DSPCAD group. Mentoring them has been a learning experience in itself, and I appreciate their efforts and response.

During first two years of my graduate studies at the University of Maryland, College Park, I worked in the departments of Kinesiology, Facilities Management, and Astronomy (at the OVRO). These positions provided not only the much needed financial assistance but also a wealth of experience, which I have cherished throughout my graduate studies. In particular, I would like to acknowledge and thank my supervisors Prof. John Jeka (of the Department of Kinesiology, University of Maryland), and Dr. James Lamb (of the OVRO). Dr. James Lamb facilitated my interactions with many others in the field

of radio astronomy. These interactions, along with his insights and encouragement, have been instrumental in shaping my research interests in the field of radio astronomy signal processing. I will be always grateful to him for that.

I would like to thank all the staff of the ECE department, especially that of Graduate Studies, Business, and Payroll offices, for their timely help and useful advice.

Though it would be too formal to thank my parents, grandparents, cousins, and every one in the Sane, Bhat, and Dadhich families, I, still, must do it for every one of them has been a source of inspiration and great strength. I thank all of them for always being there.

I would like to express my sincere gratitude toward families of Mr. Arun and Late Mrs. Usha Karve, Dr. Ajit and Dr. Asha Kembhavi, Dr. Naresh and Mrs. Sadhana Dadhich, Mr. Mohammad and Mrs. Fatima Khadas, Mr. Jawahar and Mrs. Pushpa Nagori, and Mr. Arun Sawant without whose support and financial assistance the journey to the United States would not have been possible. It is rather unfortunate and sad that my grandmother, Mrs. Usha Karve, is not with us when I complete my doctoral degree.

I must also remember and thank all my close friends who have been supportive throughout my graduate studies and are as much a part of life as my family.

I am grateful to Mr. Neel and Mrs. Gandhali Phadake, and Mr. Suresh and Dr. Kunda Joshi for their support.

I would like to thank Dr. Aniruddha Kembhavi, Avanti Shetye, Dr. Ayan Ghosh, and Amit Apte, who helped me immensely during my initial days in College Park, and continued to guide me throughout my graduate studies.

I should thank my room-mates during my stay in College Park — Kunal, Dhaval,

Nirankush, Mehul, Sarang, Prashant, Ashwin, Aalap, Omkar, Jagannath, Shreyas, Saurabh, Kaustubh, Harshavardhan, Sumit, and Kamal — for being such a wonderful family away from home, and of course, bearing with me. I also thank all my friends from College Park, who made my experience at the University of Maryland so enjoyable.

I am sure there are many others who have been of great help and support at various stages of my PhD. It may not be possible to list all of them here. I take this opportunity to thank each and every one of them.

Table of Contents

List of Figures xi

List of Tables xiv

1 Introduction 1
1.1 High Performance Signal Processing Applications 1
1.2 Design Tools for Rapid Prototyping 4
1.3 Contributions of the Thesis 8
1.3.1 Topological Patterns for Specification and Analysis of Dataflow Graphs 9
1.3.2 Formulation of the Core Functional Dataflow Model 11
1.3.3 Efficient Scheduling Techniques for Core Functional Dataflow Graphs 13
1.3.4 Dataflow-based Rapid Prototyping for Radio Astronomy Signal Processing 14
1.3.5 Development and Release of the latest version of the MoCGraph Package 16
1.4 Organization of Thesis 16

2 Background 18
2.1 Dataflow Modeling 18
2.1.1 Synchronous Dataflow 20
2.1.2 Cyclo-static Dataflow 21
2.1.3 Parameterized Dataflow 22
2.2 The Dataflow Interchange Format 27
2.3 Generalized Schedule Trees 29
2.4 Graph Package Oriented toward Models of Computation 30
2.4.1 MoCGraph Software Architecture and Features 31
2.4.2 `mocgsched` Plug-in 33
2.5 DICE: The DSPCAD Integrative Command-Line Environment 33
2.6 Summary 35

3 Topological Patterns for Specification and Analysis of Dataflow Graphs 36
3.1 Introduction 36
3.2 Related Work 38
3.3 Topological Patterns 42
3.3.1 Topological Patterns in Signal Processing 42
3.3.2 Parameter Propagation 44
3.4 Topological Patterns in DIF 45
3.5 Applications of Topological Patterns 47
3.5.1 Graph Analysis 48
3.5.2 Representing HSDF Graphs 49
3.5.3 Extracting Implementation-Specific Features 52

3.5.4 Exploring Implementation Trade-offs 54
3.5.5 Representing Schedules 56
3.5.6 Experimenting with Pattern-Specific Schedules 60
3.6 Summary 66

4 Prototyping Heterogeneous Dataflow Applications using Core Functional Dataflow 67
4.1 Related Work 67
4.2 Formulation of Core Functional Dataflow 69
4.3 Modeling Adaptive Modulation Scheme using Core Functional Dataflow . 71
4.4 Translation to Core Functional Dataflow 74
4.4.1 Static Dataflow 74
4.4.2 Boolean Dataflow 76
4.4.3 Representing PSDF and PCSDF Actors using CFDF 77
4.5 Functional Simulations in DIF using Core Functional Dataflow 79
4.5.1 Extensions to DIF Software Architecture 80
4.5.2 Canonical Scheduler 81
4.6 Design Examples 81
4.6.1 Programmable Polynomial Evaluation Accelerator 82
4.6.2 Design with Multiple Polynomial Evaluation Accelerators 83
4.6.3 Results 85
4.7 Summary 86

5 Efficient Scheduling Techniques for Core Functional Dataflow Graphs 88
5.1 Scheduling using Dynamic Dataflow Graph Decomposition 88
5.1.1 Dynamic Dataflow Graph Decomposition Algorithm 89
5.1.2 Simulation Results 92
5.2 Scheduling using Mode Grouping 95
5.2.1 Simulation Results 96
5.3 Parameterized Looped Schedules 97
5.3.1 Related Work 99
5.3.2 Constructing Parameterized Looped Schedules 100
5.3.3 Bounded Memory Execution 102
5.3.4 Simulation Results 104
5.4 Summary 108

6 Dataflow-based Rapid Prototyping for Radio Astronomy Signal Processing 110
6.1 Introduction 111
6.2 Related Work 113
6.3 Tunable Digital Downconverter 114
6.4 Dataflow-based Design and Implementation 117
6.4.1 Modeling and Prototyping using DIF 118
6.4.2 Integration with the CASPER Tool Flow 125
6.4.3 Platform-specific Analysis using DIF 128
6.4.4 Software Interface for the Tunable Digital Downconveter 129
6.5 Exploring Implementation Trade-off with TDD and FDD Blocks 130

6.5.1 TDD and FDD for Multistage Downconversion 131
6.6 Summary 132

7 Summary and Conclusions 134

8 Future Work 139

Bibliography 142

List of Figures

1.1 Design flow using a model based approach. 6

1.2 DIF based design flow. 7

2.1 An SDF graph. 21

2.2 An application graph with a simple decimator actor `M` using the (a) CSDF, and (b) SDF models. Actor `M` is a decimator with a *decimation factor* of 4. 22

2.3 Modeling a parameterized decimation filter (`DF`) application using PCSDF: (a) Application graph — C_N denotes a vector of FIR filter coefficients, and D denotes a decimation factor, and (b) PCSDF representation. 25

2.4 The DIF language. 28

2.5 Generalized schedule tree. 29

2.6 A hierarchy of classes in MoCGraph that implements various types of graph data structures. 32

3.1 Configuring an array of nodes with a PPP. 45

3.2 Overlapping topological patterns. 46

3.3 A sample rate converter. 50

3.4 Dataflow graph for a 4-point fast Fourier transform and the `topology` block in its DIF specification. 52

3.5 JPEG encoder and the `topology` block in its DIF specification. 53

3.6 Dataflow graphs for (a) the generic class of applications under consideration, and (b) a simplified adaptive modulation scheme. 57

3.7 A PLS for the application in Fig. 3.6(b), and the `topology` block in a corresponding DIF representation. Table 3.2 provides parameters associated with each `node` in the DIF specification. 58

3.8 (a) An SDF graph with a `butterfly` pattern. (b)-(c) two possible GST structures using schedules that are based on acyclic pairwise clustering (iteratively clustering two actors at a time). 61

3.9 (a)-(b) SDF graphs with `butterfly` patterns. (c)-(d) GSTs for minimizing buffer memory requirements of the SDF graphs in (a) and (b), respectively. 64

4.1 Mode transition behavior of the `mapper` actor. 73

4.2 Mode transition behavior of the decimator actor. 76

4.3 Application of BDF using a `Switch` actor. 77

4.4 Mode transition behavior of the `Switch` actor. 78

4.5 CFDF mode transition behavior of a PCSDF actor `Decimator` with possible decimation factors 3 and 4. 79

4.6 A pictorial representation of the dual PEA application. 85

4.7 Single appearance schedule for the dual PEA system. 86

4.8 Multiple appearance schedule for the dual PEA system. 86

5.1 Algorithm for dynamic dataflow graph decomposition. 90

5.2 Application decomposition example. 92

5.3 Dual sampling rate conversion. 93

5.4 The APGAN schedule of the sample rate conversion application. 94

5.5 Valid PLS for the application in Fig. 3.6(b). 101

6.1 Block diagram of a tunable digital downconverter. 116

6.2 Proposed dataflow-based approach. 117

6.3 Dataflow behavior of a `Decimator` actor with 4 inputs and outputs for a *decimation factor* of 6 using (a) SDF, and (b) CSDF models. 119

6.4 TDD application graph generated using DIF. 121

6.5 Partial DIF specification — `topology` block — for the TDD application graph using topological patterns. 122

6.6 Partial DIF specification — `topology` block — for the TDD application graph using topological patterns. 123

6.7 PLSs for the TDD application configured for a decimation factor of 11, and `decimator` actor employing the (a) PSDF and (b) PCSDF models of computation. 124

6.8 TDD System overview. 129

6.9 Two-stage digital downconversion. 132

List of Tables

2.1 Interactions among PDF subsystem components, dataflow inputs and outputs, and parameters. 24

3.1 Performance and resource utilization trade-offs for FPGA implementation of a JPEG encoder. 55

3.2 Actors and loop counts associated with nodes in the PLS graph representation. 59

3.3 Average simulation times for different sink control conditions (numbers of tokens consumed by the sink) for the PLS in Fig. 3.7 using (1) GST traversal, and (2) a hand-tuned pattern-specific schedule. 60

3.4 Buffer memory requirements for single appearance schedules generated from the SDF graph shown in Fig. 3.9(a). 65

3.5 Buffer memory requirements for single appearance schedules generated from the SDF graph shown in Fig. 3.9(b). 65

4.1 Valid modes for the `mapper` actor along with their corresponding production and consumption rates. 73

4.2 Valid modes in a CFDF representation of the decimator actor, `M`, in Fig. 2.2(a) along with their corresponding production and consumption values. . . . 75

4.3 The behavior of the `Switch` actor modes in terms of tokens produced and consumed. 77

4.4 The behavior of the PEA modes. 82

4.5 The behavior of the CSDF implementation of the restricted PEA used in the dual PEA application. 84
4.6 Simulation times and maximum buffer sizes for mixed-model applications. 87

5.1 Simulation times and maximum buffer sizes for mixed-model applications using dynamic dataflow graph decomposition based schedules. . . . 95
5.2 Total buffer size requirements with and without mode grouping. 97
5.3 Average simulation time for different sink control conditions (numbers of tokens consumed by the sink actor) using a canonical schedule (CS), DDFS, and PLS. 106
5.4 Total buffer requirements for different sink control conditions (numbers of tokens consumed by the sink) for a canonical schedule (CS), DDFS, and PLS. 106

6.1 Total buffer requirements from a DIF prototype for different decimation factors using parameterized looped schedules. 125
6.2 Implementation summary for TDD designs. 127
6.3 Implementation summary for FDD designs. 131
6.4 Implementation summary for designs employing two-stage downconversion using cascaded FDF or TDF blocks. 133

Chapter 1

Introduction

1.1 High Performance Signal Processing Applications

The field of signal processing has expanded to cover a wide range of application domains, such as image and video processing, acoustic and speech processing, wireless communication, software-defined radio, astronomical signal processing, biomedical signal processing, and medical imaging, to name a few, and it will continue to expand in the future. Each of these application domains deals with different types of signals that can be characterized by, for example, the signal sources, methods and speed of signal acquisition, signal strength in terms of signal to noise ratio, and frequency content of the signals. A signal processing algorithm processes these signals to derive useful information. Apart from varying application specifications, there are certain performance metrics, such as the need for real-time signal processing, speed of real-time or offline processing, resource utilization, power consumption, and cost, that provide complex design spaces for exploration, and help characterize and optimize the overall quality of a design. These requirements may be domain-specific. For example, study of certain astronomical objects or observations at certain wavelengths would determine specifications for a radio telescope and its signal processing backend; a wireless communication standard may determine signal processing requirements, while its deployment in a consumer product may constrain specifications such as cost and time-to-market; specifications of a video

surveillance system may be determined by its expected target-tracking capability; and so on.

A designer of any such signal processing system explores the design space to determine if and how the required system can be efficiently implemented using the present state of the art hardware platforms and design tools. Signal processing hardware manufacturers often classify their hardware that can be used to achieve maximum performance with respect to certain performance metrics as *"high performance"* signal processors. Given the evolving nature of hardware platforms, architectures, and their processing capabilities, the term "high performance" represents the extreme end of performance achievable or desirable under a given cost constraint using the state of the art at a given point in time. A signal processing application, which has specifications requiring the corresponding signal processing system to deliver maximal performance, possibly under pre-specified resource constraints, can be classified as a *high performance signal processing application*. This is in contrast, for example, to a commercial signal processing system that must be designed to minimize cost subject to a given performance constraint.

Radio astronomy digital signal processing (DSP) is an example of high performance signal processing. Scientific objectives drive radio astronomy telescopes toward high-speed samplers and corresponding DSP backends that need to process large volumes of data at high data rates. The single dish Green Bank Telescope (GBT) [68], for example, is used for pulsar searches and high-precision timing studies, which drive bandwidth requirements to extremes along with the computing needs to implement techniques such as coherent dedispersion. The new GBT spectrometer being conceived will have a channel bandwidth of 3 GHz (compared to the current 800 MHz), but must also support zooming

into tunable narrow bands. Many of the recent radio telescopes that have been constructed and the ones that have been conceived for construction over the next decade clearly show a trend toward synthesis array design [4]. These include telescope arrays such as the Square Kilometre Array (SKA) [23]; its precursors, the Karoo Array Telescope (MeerKAT) [40], and Australian SKA Pathfinder (ASKAP) [22] being built in South Africa, and Australia, respectively; and the Long Wavelength Array (LWA) in New Mexico [27]. In the case of synthesis array telescopes, the amount and speed of data to be processed increase due to the bandwidth required per antenna as well as the large numbers of antennas that are involved. The SKA will employ a large number of antennas of smaller diameters to be able to conduct ultrasensitive surveys over large area of the sky, and its precursor arrays being developed already pose challenging DSP applications in terms of processing enormous amounts of high-speed data [4]. The signal processing requirements for radio arrays (for example, the complexity of a correlator) scale quadratically or with higher order in the number of elements in the array. The performance metrics for these DSP backends require handling of enormous amounts of data in real time at very high rates (on the order of petabit per second). At the same time, the power and cost need to be minimized by orders of magnitudes compared to the present estimates to provide realistic solutions.

Some important commercial applications that involve high performance DSP include dynamic communication systems applications related to modern wireless technologies, such as the *worldwide interoperability for microwave access* (*WiMAX*) [3] and *3rd generation partnership project — long term evolution* (*3GPP—LTE*) [1]. The challenges posed by these applications are manifold including greater data rates, requirements for supporting dynamic configurability, need for faster simulations, and deployment in hand-

held devices. Some other domains of high performance signal processing include software defined radio, multimedia processing, and medical imaging. Applications such as these, especially those related to DSP for radio astronomy and wireless communication systems, have driven the work presented in this thesis.

1.2 Design Tools for Rapid Prototyping

The interaction between the evolving nature of high performance hardware platforms and signal processing applications will be more clear if one observes the trends in a particular application domain. For example, in the field of radio astronomy signal processing, the conventional approach has been to develop optimized custom hardware using application specific integrated circuits (ASICs) (e.g., see [20] for an ASIC solution to SKA DSP). Such designs, however, are not scalable, reconfigurable, or portable. Moreover, the design, development, and deployment process tends to be much longer and more costly than that for some of the reconfigurable hardware available. To account for these factors, DSP solutions that use field programmable gate array (FPGA) based reconfigurable hardware and modular software libraries have been developed (e.g., see the approach used by the Collaboration for Astronomy Signal Processing and Electronics Research (CASPER) group [55]). Recent years have seen the emergence of a large variety of computing platforms having general purpose or specialized muticore processors, such as graphics processing units (GPUs), the Cell, and a variety of processors by ARM, Tilera, and Intel [14]. Such platforms are gaining popularity within the radio astronomy domain (e.g., see [31, 81]). Moreover, it is quite possible that a DSP solution that uses a

combination of more than one hardware architecture for performing different signal processing tasks may perform better under a given set of constraints than a solution that uses only one kind of platform.

We, thus, have a complex design space to explore based on performance trade-offs (e.g., throughput, latency, power, cost, etc.), and expanded by the multitude possibilities of target platforms. A key to efficient implementation is the ability of design processes and tools to allow the designer to explore the design space effectively at a high-level, and make informed design choices at an early stage rather than trying to alter the design in major ways after having a platform-specific implementation. The need for rapid prototyping is of particular importance for high performance signal processing applications because it allows designers to focus on functionality and functional validation in early stages of design, and decide on target platforms in later stages.

A tool for designing a high performance signal processing system should, therefore, allow scalable high-level representations, semantics for analysis and resource estimation, functional verification, and portable implementations that can be reconfigured and retargeted toward the latest hardware technologies. This is possible using a model based design approach as shown in Fig. 1.1 that is founded in a particular model of computation (MoC). Model based design approaches for design and implementation of embedded systems for DSP applications continues to be an active and expanding research area both in the industry as well as academia due to ever expanding application domains and markets for such systems (e.g., see [8]). Model based design methods are extensively used in this field to make the design process streamlined, productive, robust, and efficient.

As shown in Fig. 1.1, the application specification is a high-level specification used

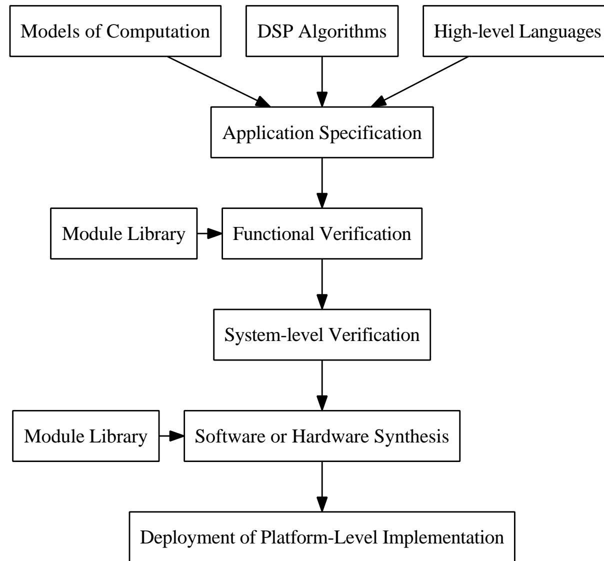


Figure 1.1: Design flow using a model based approach.

to specify only the design details that are necessary for establishing functional correctness. The high-level language used for such a specification provides syntactic features and has semantic foundations that establish the underlying choice of the MoC. A MoC provides semantics (meaning) for the interaction between functional modules in a system, and strongly influences all the significant design processes — specification, simulation, formal verification, implementation/optimization, and interfaces to environmental or external systems. A MoC essentially tries to capture the designer's intuition and effectively translates it into a model (e.g., the model that underlies "C" and other imperative languages, finite state machines, Kahn process networks, dataflow, synchronous/reactive, discrete-event, etc.) Model based design is important area for innovation in domain-specific technology and design research.

To facilitate analysis of and mapping from a model-based specification, the specification is typically converted into an intermediate representation that can be used by design

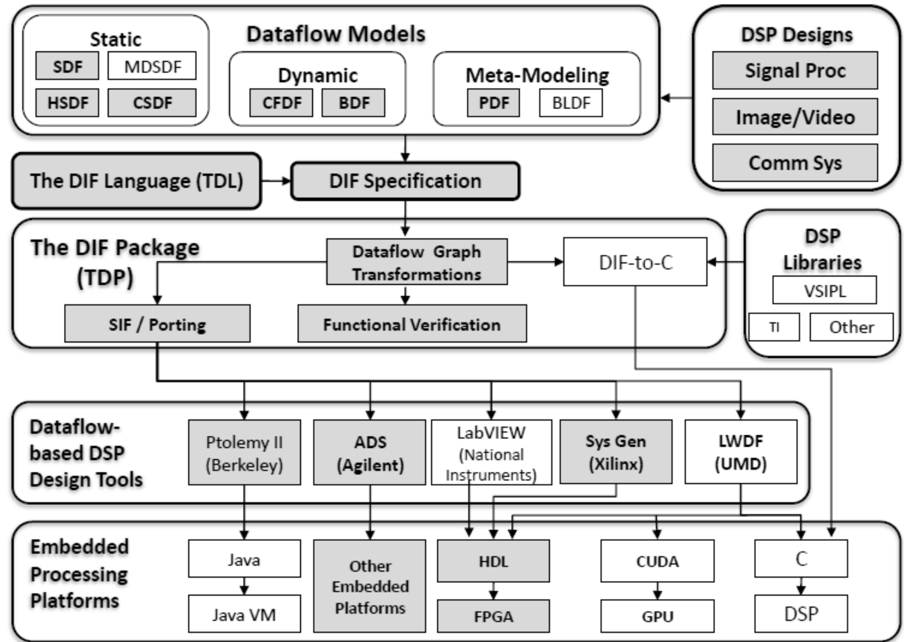


Figure 1.2: DIF based design flow.

tools during subsequent design stages. For example, in dataflow-based design, a DSP application flowgraph specification is translated into a directed graph with nodes representing functional modules and edges representing first-in-first-out communication buffers between pairs of functional modules. The subsequent stages utilize this representation for further analysis and optimization. It should be noted that module libraries that contain functional code for the modules also adhere to the prespecified interface and semantics of the selected MoC. This allows easier transcoding among various platform-specific languages, while the glue code for these modules remains platform-agnostic. This allows developing platform-independent functional prototypes that can be used for functional and system level verification as well as portability across different kinds of hardware technologies.

The work presented in this thesis primarily focuses on the dataflow MoC, which is

extensively used for developing embedded systems for DSP and communication applications, and electronic design automation. Dataflow-oriented DSP design tools typically allow high-level application specification, software simulation, and possibly synthesis for hardware or software implementation. Chapter 2 provides relevant background on dataflow modeling. There are various existing design tools with their semantic foundations in dataflow modeling such as Ptolemy [59], PeaCE [44], SysteMoc [32], StreamIt [79], CAL [26], Compaan/Laura [77], and DIF [36]. In this work, the dataflow interchange format (DIF) tool has been primarily used for demonstration along with other tools such as the Advanced Design System (ADS) from the Agilent Technologies, Inc. [60] and Xilinx System Generator (XSG) [85] (See Chapter 2 for a brief description of the DIF language and package). Fig. 1.2 provides a pictorial representation of the general DIF based design flow. This thesis significantly contributes to the components of the design flow illustrated in Fig. 1.2 that have been highlighted with a gray background. It can be seen from this figure (and also comparing with the generic model based design flow in Fig. 1.1) that this thesis addresses most of the aspects of a complete model based design flow.

1.3 Contributions of the Thesis

As mentioned earlier, this thesis deals with various aspects of a dataflow-oriented, model based design approach. This section lists the important contributions of this thesis. The research presented in this thesis, though demonstrated using specific design tools, is not restricted to those tools, and hence, can be applied to a variety of other dataflow-based design tools. Also, the applications presented, though driving this research, are

demonstrative, and the prototyping methods can be extended or adapted to other relevant DSP applications.

1.3.1 Topological Patterns for Specification and Analysis of Dataflow Graphs

Tools for designing signal processing systems with their semantic foundation in dataflow modeling often use high-level graphical user interfaces (GUIs) or text based languages that allow specifying applications as directed graphs. Such graphical representations serve as initial reference points for further analysis and optimizations that lead to platform-specific implementations. For large-scale applications, the underlying graphs often consist of smaller substructures that repeat multiple times. To enable more concise representation and direct analysis of such substructures in the context of high-level DSP specification languages and design tools, we have introduced and developed the modeling concept of *topological patterns* [71]. Topological patterns can be used to identify and concisely iterate across arbitrary structures in a dataflow application graph. We have shown how the types of flowgraph substructures that are pervasive in the DSP application domain can be effectively represented in terms of topological patterns, and thereby used to generate compact, scalable application representations.

Some earlier research efforts have employed useful techniques for deriving and exploiting various types of specialized dataflow substructures within their respective compilers (e.g., use of highly expressive constructs from procedural languages, such as recurrences, iteration, and conditionals, in dataflow-oriented languages [45], and various

textual languages for DSP system design, such as SILAGE [82], StreamIt [79], and CAL [26]). They, however, lack a general method for explicit and scalable representation of such substructures by the programmer. Such a programming interface for topological patterns is essential to capture the broad range of relevant patterns in ways that are scalable, and flexibly extensible to accommodate new types of patterns as they emerge from new applications and modeling techniques. Our concept of topological patterns is designed precisely to bridge this gap. In other prior work, higher-order functions have been shown to permit elegant construction of structured subsystems in dataflow representations [48]. Higher-order functions are functions that take functions as inputs or produce functions as outputs. Topological patterns provide a related but technically different approach since topological patterns operate on generic directed graph vertices. Furthermore, our development of topological patterns is tightly integrated with textual graph representation and arrays of graph vertices and edges, which are useful for providing scalable representations and managing large-scale designs.

We have shown how an underlying design tool can exploit a high-level application specification consisting of topological patterns in various aspects of the design flow. In particular, we have demonstrated the efficacy of topological patterns in dataflow graph analysis, concise and scalable representation of homogeneous synchronous dataflow (HSDF) graphs, and exploring implementation-specific trade-offs. We have applied the concept of topological patterns to represent schedules for application graphs. Such representations are useful, for example, when porting schedules generated using one design tool to other platform-specific tools or design languages. We have demonstrated the utility of experimentation with pattern-specific scheduling transformations, and how topological patterns

facilitate such experimentation.

1.3.2 Formulation of the Core Functional Dataflow Model

For a number of years, dataflow models have proven invaluable for application areas such as DSP. Their graph-based formalisms allow designers to describe applications in a natural yet semantically rigorous way. Such a semantic foundation has permitted the development of a variety of analysis tools. As a result, dataflow languages are increasingly popular. Their diversity, portability, and intuitive appeal have extended them to many application areas with a variety of targets (e.g., see [73], [34], and [61]). As system complexity and the diversity of components in digital signal processing platforms increases, designers are expressing more types of behavior in dataflow languages to retain these implementation benefits. This has resulted in evolution of various dataflow models with varying degrees of expressive power. On one extreme lies the synchronous dataflow (SDF) model, which is the most restrictive form of dataflow [46] that is in widespread use in the DSP design community. On the other hand, Turing complete models such as boolean dataflow (BDF) [16] can express deterministic dynamic dataflow behaviors (i.e., dynamic behaviors in which a given set of input streams always produces a unique set of output streams). There exist numerous dataflow models with intermediate levels of expressive power, such as the cyclo-static dataflow (CSDF) model [13], which allows statically specified periodic dataflow behavior, and a meta-modeling technique called parameterized dataflow (PDF), which allows limited forms of data-dependent dynamic behavior [6]. Each of these models also offer differing capabilities to analyze and estimate

resources at compile-time. The designer must explore this design space to find the model that can best capture the application behavior and allow sufficient analysis.

While the semantic range of dataflow has expanded to cover quasi-static and dynamic interactions, it is often challenging to map such interactions reliably and efficiently into implementations. The recently developed *core functional dataflow* (*CFDF*) model can be used to model a wide variety of deterministic dynamic dataflow behaviors [64], and thereby helps to unify the processes of analysis and scheduling of quasi-static and dynamic dataflow interactions. CFDF supports flexible and efficient prototyping of dataflow-based application representations and permits natural description of both dynamic and static dataflow actors. I have significantly contributed to the conception and development of the CFDF MoC. This thesis presents its mathematical formulation and modeling features. This model provides an interface for actor specification that is intuitive for a DSP system designer and allows specification of heterogeneous dataflow applications. We have applied this model to develop various application prototypes that demonstrate its effectiveness for rapid DSP system prototyping, as presented in [64] and [66].

The CFDF model, and tools based on it, has the following unique set of features: 1) support for heterogeneous dataflow behaviors, 2) intuitive and common frameworks for functional specification, 3) support for functional simulation, 4) portability from many well-known dataflow models to CFDF, 5) integrated emphasis on minimally-restricted specification of actor functionality, and 6) support for efficient static, quasi-static, and dynamic scheduling techniques. These features distinguish CFDF from a related model called stream based functions (SBF) [41], and other frameworks such as Ptolemy II, which

offers diverse MoCs [25], SystemC [32], actor-oriented languages like CAL [26], "S-functions" in Simulink [51], and GUI based tools like LabVIEW [39].

1.3.3 Efficient Scheduling Techniques for Core Functional Dataflow Graphs

The problem of scheduling dynamic dataflow applications has been studied earlier, and important results have been established regarding bounded memory execution and compile-time scheduling (e.g., see [16, 53]). Most of these approaches employ scheduling schemes that suffer from significant run-time overhead, difficulties in code generation, and lack of compile-time predictability (e.g., for validating real-time signal processing performance). We have developed generalized scheduling techniques for CFDF graphs based on decomposition of a CFDF graph into static graphs that interact at run-time [62]. Furthermore, we have conceived the mode grouping based scheduling technique that achieves more efficient simulations compared to previous approaches for dynamic dataflow applications [63].

The scheduling techniques for generalized CFDF graphs in [62] and [63], however, do not in general guarantee bounded memory execution for the entire input application. For dynamic communication applications that exhibit a particular kind of dataflow graph structure, I have developed a technique to construct *parameterized looped schedules* (*PLS*s). Well-constructed PLSs allow expressing dynamic dataflow behavior and enabling faster simulations without significantly compromising compile-time predictability [70]. This class of quasi-static schedules allows for flexible, compact specification of nested loop structures, where loop iteration counts can be either constant values or

symbolic expressions in terms of dynamic parameters in the underlying dataflow graph. While it may be possible to use a meta-modeling technique called PDF (e.g., by integrating it with SDF to give a model called parameterized SDF (PSDF)) to express limited forms of dynamic behavior and construct PLSs [6], such an approach imposes significant restrictions on how applications are modeled (e.g., in terms of hierarchies of cooperating *init*, *subinit*, and *body* graphs [6]), and in general, major changes in the user interface are required to provide direct support for PDF in a design tool. In contrast, a CFDF based approach provides PLS-based bounded memory scheduling while operating within a semantic framework that can be integrated more directly into existing design tools compared to the more hierarchical semantic structure of PDF representations.

This thesis presents analysis and scheduling techniques for CFDF graphs based on those in [62], [63], and [70], which I have contributed to.

1.3.4 Dataflow-based Rapid Prototyping for Radio Astronomy Signal Processing

Application of dataflow modeling to the field of radio astronomy DSP is a significant contribution of this thesis. The model based approach for designing large scale signal processing systems with focus on radio telescopes has been previously studied (e.g., see [2, 49]). Several frameworks have been proposed for model based high-level abstractions of architectures along with performance/cost estimation to guide the designer throughout the development cycle (e.g., see [2]). However, the focus of these approaches has been on architecture exploration. There have also been attempts to derive

implementation-level specifications starting from system-level specifications by segregating signal processing and control flow into an application specification and architecture specification, respectively (e.g., see [49]). However, the choice of models of computation has been made primarily from control flow considerations rather than dataflow considerations. These approaches, though relevant, do not specifically address the issue of high-level application specification for platform-independent prototyping and use of models of computation for abstraction of heterogeneous or hybrid dataflow behaviors. This issue is critical to efficient prototyping of high performance signal processing applications, which are typically dataflow dominated, and include increasing levels of dynamic dataflow behavior (e.g., see [8]).

I demonstrate the use of dataflow-based approaches for design and development of radio astronomy DSP systems using an application example of a *tunable digital downconverter* (*TDD*) for spectrometers at the *National Radio Astronomy Observatory* (*NRAO*), *Green Bank*. Design and development of this module has been an integral part of this thesis work. This thesis demonstrates a design flow that consists of (i) application specification and modeling using parameterized SDF and CSDF models, (ii) software prototyping, analysis, and simulation using the DIF tool, and (iii) integration of this benchmark design with the current CASPER tool flow for the target implementation on an FPGA platform, called the *interconnect break-out board* (*IBOB*) [55]. My experiments show how formal understanding of the dataflow behavior from the software prototype allows more efficient prototyping along with estimating and accounting for some of the key resource requirements (e.g., throughput, hardware duplication, pipelining, buffer memory requirements) at much earlier stages in the design cycle, unlike conventional design approaches.

1.3.5 Development and Release of the latest version of the MoCGraph Package

The DIF package makes use of and extends a graph package oriented toward MoC (MoCGraph). MoCGraph is a Java-based package of generic graph data structures and algorithms with emphasis on supporting graph-theoretic analysis for MoCs. MoCGraph has evolved from the graph package in Ptolemy II [25].

This thesis has contributed significantly to the development and release of the latest version of the MoCGraph package. My contributions to MoCGraph include support for tree data structures, and generalized schedule trees (GSTs). A GST is a data structure used for representing schedules of dataflow graphs [42]. This extension to the MoCGraph package has provided important support for the CFDF model, and functional simulation in the DIF package. Chapter 2 provides a detailed description of the MoCGraph and DIF packages, and the GST data structure.

1.4 Organization of Thesis

The remainder of this thesis is organized as follows. Chapter 2 provides background relevant to the research presented in this thesis. It specifically deals with foundations of dataflow modeling, and features of the MoCGraph and DIF packages as well as the GST data structure. Chapter 3 describes the concept of using topological patterns for specification of dataflow graphs and its application to analysis of dataflow graphs. Chapter 4 discusses formulation of CFDF model, its connections to some of the existing dataflow models, and its use in modeling dynamic dataflow applications. Chapter 5 describes var-

ious scheduling techniques for efficient simulation of CFDF graphs. Chapter 6 demonstrates the application of dataflow modeling techniques to radio astronomy DSP systems using the TDD application. A summary of findings, and conclusions from this work are presented in Chapter 7, while Chapter 8 lists useful directions for future research.

Chapter 2

Background

2.1 Dataflow Modeling

Dataflow modeling involves representing an application using a directed graph $G(V, E)$, where V is a set of vertices (nodes) and E is a set of edges. Each vertex $u \in V$ in a dataflow graph is called an *actor*, and represents a specific computational block, while each directed edge $(u, v) \in E$ represents a first-in-first-out (FIFO) buffer that provides a communication link between the *source* actor u and the *sink* actor v. A dataflow graph edge e can also have a non-negative integer *delay*, $\mathrm{del}(e)$, associated with it, which represents the number of initial data values (*tokens*) present in the associated buffer. Dataflow graphs operate based on *data-driven execution*, where an actor can be executed (*fired*) whenever it has sufficient amounts of data (numbers of "samples" or data "tokens") available on all of its inputs.

In the context of a dataflow graph, a *source* actor is an actor in the topology that has no input edges (for example, actor `W` in Fig. 2.1), and *sink* actor is an actor in the topology that has no output edges (for example, actors `Y` and `Z` in Fig. 2.1).

During each firing, an actor consumes a certain number of tokens from each input and produces a certain number of tokens on each output. When these numbers are constant (over all firings), we refer to the actor as an SDF actor [46]. For an SDF actor, the numbers of tokens consumed and produced in each actor execution are referred to as

the *consumption rate* and *production rate* of the associated input and output, respectively. If the source and sink actors of a dataflow graph edge are SDF actors, then the edge is referred to as an SDF edge, and if a dataflow graph consists of only SDF actors, and SDF edges, the graph is referred to as an SDF graph.

For a dataflow graph edge e, $\mathrm{src}(e)$ and $\mathrm{snk}(e)$, denote its source and sink actors, and if e is an SDF edge, then $\mathrm{prd}(e)$ denotes the production rate of the output port of $\mathrm{src}(e)$ that is connected to e, and similarly, $\mathrm{cns}(e)$ denotes the consumption rate of the input port of $\mathrm{snk}(e)$ that is connected to e.

A *static schedule* for a dataflow graph G is a sequence of actors in G that represents the order in which actors are fired during an execution of G.

Usually, production and consumption information — in particular, the number of tokens produced and consumed (production/consumption *volume*) — by individual firings is characterized in terms of individual input and output ports so that each port of an actor can in general have a different production or consumption volume characterization. Such characterizations can involve constant values as in SDF [46] (as described above); periodic patterns of constant values, as in CSDF [13]; or more complex forms that are data-dependent (e.g., see [16, 6, 64]). A meta-modeling technique called PDF allows limited forms of dynamic behavior [6] in terms of run-time changes to dataflow graph parameters. The BDF [16] and CFDF [64] models are highly expressive (Turing complete) dynamic dataflow models.

The following sections elaborate more on SDF, CSDF, and PDF models.

2.1.1 Synchronous Dataflow

An SDF graph is characterized by its compile-time predictability through the statically known consumption and production rates, as defined above. Fig. 2.1 shows a simple SDF graph having actors `W`, `X`, `Y`, and `Z`. Each edge is annotated with the number of tokens produced on it by the source actor and that consumed from it by the sink actor during every invocation of the source and sink actors, respectively. For example, actor `X` can be fired when there are at least two tokens on its input. Whenever actor `X` is fired, it consumes two tokens from its input buffer, and produces three tokens onto the output buffer connected to `Y` and two tokens onto the output buffer connected to `Z`.

In case of SDF graphs, it is possible to construct a periodic schedule that repeats itself during the application execution. For an SDF graph $G(V, E)$, each actor $u \in V$ fires exactly $q(u)$ times in a periodic schedule, where $q(u)$ is its repetition count obtained by solving the balance equation

$$q(\mathrm{src}(e)) \times \mathrm{prd}(e) = q(\mathrm{snk}(e)) \times \mathrm{cns}(e) \tag{2.1}$$

for each edge $e \in E$. For example, repetition counts for actors `W`, `X`, `Y`, `Z` in the SDF graph shown in Fig. 2.1 are 2, 1, 3, and 1, respectively. One of the ways to execute this SDF graph is to fire the actors in the order `WWXYYYZ`. This sequence represents one of the valid schedules for this SDF graph and can also be represented as an equivalent looped schedule `(2 W)X(3 Y)Z`.

More information on SDF graphs, conditions for having a valid schedule for an SDF graph, and various techniques of scheduling SDF graphs can be found in [11].

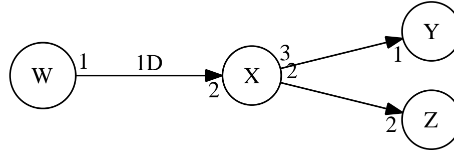


Figure 2.1: An SDF graph.

2.1.2 Cyclo-static Dataflow

Many signal processing applications involve behaviors in which production and consumption rates may change during run-time. In some cases, these changes may, however, be known at compile-time. For example, consider the CSDF graph shown in Fig. 2.2(a), which has a *decimator* actor `M` in it. This actor consumes one token from its input on each invocation, but produces a token onto its output only on every fourth invocation. This behavior has been depicted using the varying production volumes denoted by $[1\,0\,0\,0]$. The numbers of tokens produced by the decimator `M` follow this cyclic pattern with a period of 4. This sequence of varying production volumes, though not leading to constant output rates like an SDF actor, is still completely deterministic and known at the compile-time. This kind of dataflow behavior, where actors exhibit token production and consumption volumes (in terms of tokens per firing on specific actor ports) that are either constant or expressible as cyclic sequences of constant volumes, is referred to as CSDF. Thus, CSDF can be viewed as a generalization of SDF in which token production and consumption volumes may be different across different firings of an actor, but follow cyclic patterns that are completely specified at the compile-time.

We refer readers to [13] for more details on the CSDF model. As shown in Fig. 2.2(a) and Fig. 2.2(b), it may be possible to transform a CSDF actor into an SDF actor.

Figure 2.2: An application graph with a simple decimator actor M using the (a) CSDF, and (b) SDF models. Actor M is a decimator with a *decimation factor* of 4.

In general, when feedback loops are present in a dataflow graph, such a transformation may introduce deadlock, and therefore should be attempted with caution. Such a transformation, when admissible (not leading to deadlock), generally has trade-offs in terms of relevant metrics including latency, throughput, and code size. More detailed comparisons between the SDF and CSDF models of computation are presented in [54, 10].

2.1.3 Parameterized Dataflow

Though CSDF provides enhanced expressive power compared to SDF, it is still unable to specify patterns in token consumption and production volumes that are not fully known at compile time. A meta-modeling technique called PDF has been proposed to represent certain kinds of dataflow application dynamics [6]. This model can be used with any arbitrary dataflow graph format that has a well-defined notion of a *schedule iteration*. For example, the PDF meta-model, when combined with an underlying SDF model, results in the PSDF model. A PSDF graph behaves like an SDF graph during one schedule iteration, but can assume different configurations across different schedule iterations.

The PDF meta-model supports semantic and syntactic hierarchy. Syntactic hierar-

chy is used, as in other forms of dataflow, to decompose complex designs in terms of smaller components. On the other hand, semantic hierarchy in PDF is used to apply specific features in the meta-model that are associated with dynamic parameter reconfiguration. A hierarchical actor that encapsulates such semantic hierarchy in PDF encapsulates a *PDF subsystem*. A PDF subsystem in turn has three underlying graphs called the *init*, *subinit*, and *body* graphs, which interact with each other in structured ways. Intuitively, the init and subinit graphs can capture data-dependent, dynamic behavior at certain points during the execution of the graph and configure the body graph to adapt in useful ways to such dynamics. Similarly, the init graph can be used to dynamically configure parameters in the subinit graph, which, in general, executes more frequently relative to the init graph. Intuitively, the init graph is designed to capture parameter configuration that is driven by higher, system-level processing, while the subinit graph is designed to capture the parameter changes occurring across different iterations of the corresponding body graph.

A PDF actor may have a set of parameters associated with it. The PDF model allows the behavior of a subsystem to be controlled by allowing such parameters to change during run-time. These parameters can control functional behavior as well as dataflow behavior (the rates at which actors produce and consume data to and from their output and input ports). In the context of PDF, these parameters can be classified as dataflow or non-dataflow parameters depending on whether or not they control dataflow behavior in the corresponding actors. In general, the functionality of a PDF actor depends on both dataflow and non-dataflow parameters. Depending upon the visibility of the parameters outside a PDF subsystem (possibly, to the enclosing PDF graph or subsystem), it is possible to classify them as *external subsystem parameters* or *internal subsystem parameters*.

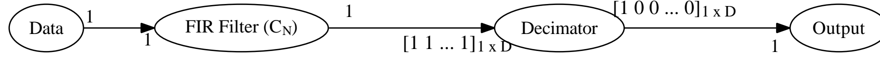
Table 2.1: Interactions among PDF subsystem components, dataflow inputs and outputs, and parameters.

Feature	PDF Subsystem Component Graphs		
	body	init	subinit
Input port connected to a dataflow edge	Yes	No	Yes
Output port connected to a dataflow edge	Yes	No	No
Sets external dataflow subsystem parameters	No	Yes	No
Sets external non-dataflow subsystem parameters	No	Yes	No
Sets internal subsystem parameters	No	Yes	Yes

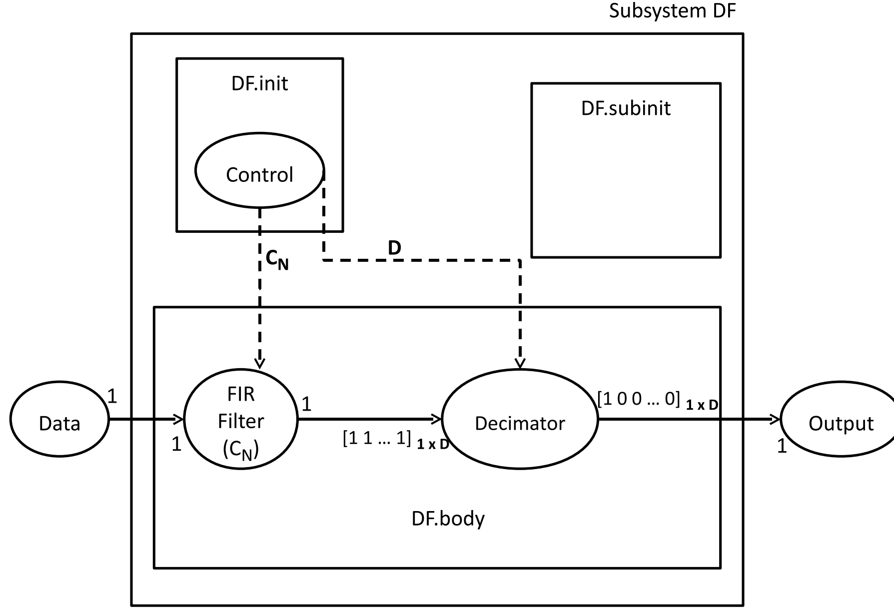
While external subsystem parameters are visible to the enclosing PDF graph, internal subsystem parameters (for example, state information of a PDF actor) are not. These parameters, collectively, are referred to as *immediate parameters*.

Details of interactions among init, subinit and body graphs and various types of parameters are described in [7]. Table 2.1 highlights some of the most important aspects of this interaction.

Intuitively, the init and subinit graphs can capture data-dependent, dynamic behavior at certain points during the execution of the graph and configure the body graph to adapt in useful ways to such dynamics. Similarly, the init graph can be used to dynamically configure parameters in the subinit graph, which, in general, executes more frequently relative to the init graph. Intuitively, the init graph is designed to capture parameter configuration that is driven by higher, system-level processing, while the subinit graph is designed to capture parameter changes occurring across different iterations of the corresponding body graph. For details related to parameter configuration, and semantics



(a)



(b)

Figure 2.3: Modeling a parameterized decimation filter (DF) application using PCSDF: (a) Application graph — C_N denotes a vector of FIR filter coefficients, and D denotes a decimation factor, and (b) PCSDF representation.

of invocations of PDF graphs, we direct the reader to [6], and [7].

To further illustrate the PDF modeling technique, we consider the application example shown in Fig. 2.3(a). This example involves a finite impulse response (FIR) filter with filter taps or coefficients given by $C_N = [c_0, c_1, \ldots, c_{N-1}]$ followed by a decimator with a tunable decimation factor of D. The values of D and C_N are set either through a higher level system or user interface. We skip the details of this mechanism for the sake of simplicity and conciseness. Such behavior can be modeled using PDF with an

underlying CSDF model. Such a modeling approach is referred to as the *parameterized cyclo-static dataflow* (*PCSDF*) model [69]. Fig. 2.3(b) shows one of the possible PCSDF graphs corresponding to the application shown in Fig. 2.3(a). The subsystem `DF` is a PCSDF subsystem with its component graphs as shown in the figure. It can be seen here that the `control` actor in the `DF.init` graph of `DF subsystem` sets the required external and internal parameters, D, and C_N, respectively. This actor models the required parameter control through either a higher level system or some form of user interface. In this particular case, the `DF.subinit` graph is empty (in general, the init, subinit and body graph do not all have to be used for a given subsystem).

The PCSDF model allows CSDF actors for which the cyclic patterns of token production and consumption volumes can be parameterized in terms of their *periods*, the actual numbers of tokens consumed or produced in the cyclo-static sequences, or both. Such a model is of particular interest for modeling multirate DSP systems that exhibit parameterizable sample rate conversions. PCSDF allows desginers to systematically explore design spaces across static, quasi-static, and dynamic implementation techniques. Here, by *quasi-static* implementation techniques, we mean techniques where relatively large portions of the associated software or hardware structures are fixed at compile-time with minor adjustments allowed at run-time (e.g., in response to changes in input data or operating conditions). A variety of quasi-static dataflow techniques are discussed, for example, in [8].

2.2 The Dataflow Interchange Format

To describe dataflow applications for a wide range of DSP applications, application developers can use the DIF language, which is a standard language founded in dataflow semantics and tailored for DSP system design [36]. DIF provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification. From a dataflow point of view, DIF is designed to describe mixed-grain graph topologies and hierarchies as well as to specify dataflow-related and actor-specific information. The dataflow semantic specification is based on dataflow modeling theory and independent of any design tool.

Fig. 2.4 illustrates some of the available constructs in the DIF language along with the syntax used for application specification. More details on the DIF language can be found in [35]. The `topology` block of the specification specifies the graph topology, which includes all of the `nodes` and `edges` in the graph. DIF supports *built-in attributes* such as `interface`, `refinement`, `parameter`, and `actor`, which identify specifications related to graph interfaces, hierarchical subsystems, dataflow parameters, and actor configurations, respectively. DIF also allows *user-defined attributes*, which have a similar syntax as built-in attributes except that they need to be declared with the `attribute` keyword.

The DIF package (TDP) (see Fig. 1.2) facilitates use of the DIF language. Along with the ability to transform DIF descriptions into manipulable internal representation, it contains graph utilities, optimization engines, verification techniques, a comprehensive functional simulation framework, and a software synthesis framework for generating C

```
[dataflowModel] graphID {
    basedon {
        graphID;
    }

    [topology] {
        nodes = nodeID, ...;
        edges = edgeID(srcNodeID, snkNodeID), ...;
    }

    [builtInAttribute] {
        elementID = value;
        elementID = id;
        elementID = id1, id2, ...;
    }

    [attribute] userDefinedAttribute {
        elementID = value;
        elementID = id;
        elementID = id1, id2, ...;
    }
}
```

Figure 2.4: The DIF language.

code [36, 64]. These facilities make DIF an effective environment for modeling dataflow applications, providing interoperability with other design environments, and developing and experimenting with new tools and dataflow techniques. Beyond these features, DIF is also suitable as a design environment for implementing dataflow-based application representations. Describing an application graph is done by listing nodes and edges, and then annotating dataflow specific information.

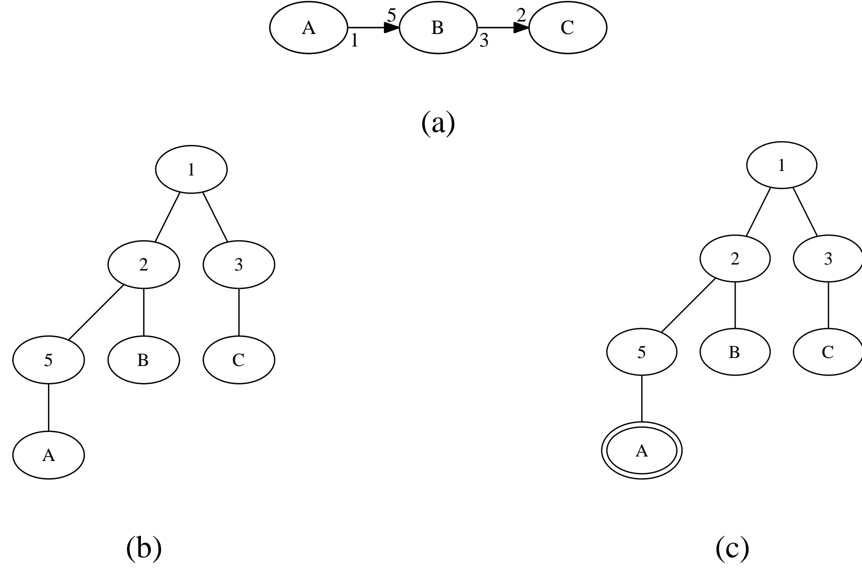


Figure 2.5: (a) An SDF graph for a sample rate converter; (b) one of the schedules for it represented as a GST; and (c) GST representing guarded and unguarded execution of actors — a GST node with two concentric ellipses or circles denotes guarded execution of the corresponding actor.

2.3 Generalized Schedule Trees

The GSTs provide a dataflow model independent representation of schedules, which can then be utilized as an input to the subsequent stages of the toolflow such as simulation and code synthesis [42]. GSTs are ordered trees with leaf nodes pointing to the actors of an associated dataflow graph. An internal node of a GST denotes a loop count (an iteration construct to be applied when executing the schedule). In this thesis, I denote the loop count and actor associated with a node u in a GST by $\mathrm{count}(u)$ and $\mathrm{actor}(u)$, respectively. The GST representation allows exploiting topological information and algorithms for ordered trees in order to access and manipulate schedule elements. The execution of a schedule involves traversing the GST in a depth-first manner, and during this traversal, the sub-schedule rooted at any internal node is executed as many times as

specified by the loop count of that node. Fig. 2.5(b) shows a GST for a valid schedule for the SDF graph shown in Fig. 2.5(a). This particular GST represents a firing sequence `(2 (5 A)B)(3 C)`, where `(n X)` implies n successive invocations of a schedule element (possibly an actor) `X`.

For sake of completeness, it must be noted that GSTs can be used to represent guarded and unguarded execution of actors as shown in Fig. 2.5(c). A GST node with two concentric ellipses or circles denotes guarded execution of the corresponding actor. A schedule represented by the GST in Fig. 2.5(c) involves guarded execution of actor `A`, and unguarded execution of actors `B` and `C`. The term "guarded execution" refers to invoking an actor following some actor-specific run-time checks, and only if certain conditions (e.g., with respect to availability of the required number of data tokens on all the actor inputs) are satisfied. We will revisit this concept and elaborate more on it in Chapter 4 in the context of CFDF model.

2.4 Graph Package Oriented toward Models of Computation

MoCGraph is a Java-based package of generic graph data structures and algorithms with emphasis on supporting graph-theoretic analysis for MoCs. MoCGraph has been developed by the Maryland DSPCAD Research Group ("DSPCAD Group"), which focuses on computer-aided design (CAD) techniques for DSP systems. The DIF package builds on the MoCGraph package to provide representations and analysis techniques that are specialized for dataflow graphs, and provide foundations for model-based design flows targeted to embedded DSP systems [36, 64].

It must be noted, however, that the features provided in MoCGraph are generally not specific to DSP or CAD-for-DSP applications, and can be used for many other kinds of graph-theoretic specification and analysis features. In fact, MoCGraph has evolved from the graph package in Ptolemy II [25], which allows experimentation with a wide variety of MoCs.

This thesis has contributed significantly to the development and release of the latest version of the MoCGraph package. This version extends the previous released version of MoCGraph to include support for tree data structures, and GSTs, in particular. These developments have provided a key foundation for supporting capabilities in the DIF package that are associated with CFDF modeling, and functional simulation. By linking to features of the *dot* package [29], this version also allows visualization of graphs that are constructed using the graph representations provided in MoCGraph.

2.4.1 MoCGraph Software Architecture and Features

The MoCGraph package provides for the following important features and facilities through its software architecture:

1. MoCGraph allows creating and manipulating generic graph data structures along with special types of graphs, such as directed graphs, directed acyclic graphs (DAGs), trees, and rooted trees among many others [18]. Fig. 2.6 shows a partial hierarchy of classes in MoCGraph. Creating a graph typically involves instantiating the appropriate graph elements — nodes and edges — along with specifying the graph topology.

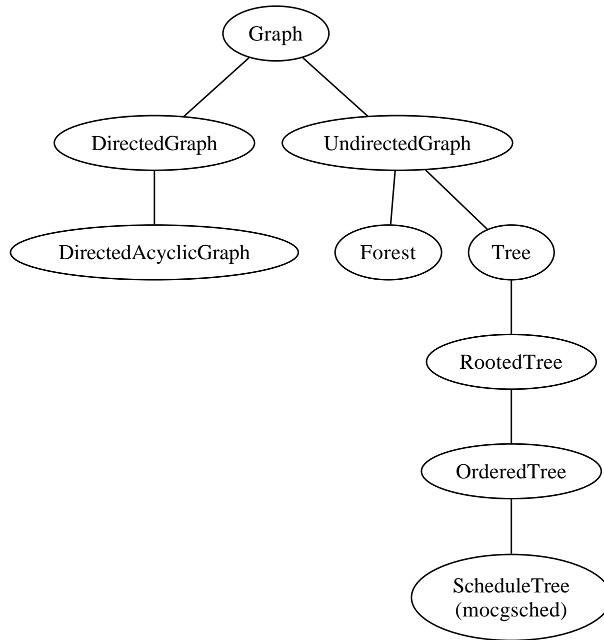


Figure 2.6: A hierarchy of classes in MoCGraph that implements various types of graph data structures.

2. MoCGraph allows assigning application-specific information associated with a graph element — a graph node or edge — through a construct known as a "*weight*," which has a generic data type. Such weights can be utilized to extend the MoCGraph package for use in specific applications or domains. For example, the DIF package makes use of this facility to assign dataflow-specific information to the nodes and edges in a dataflow graph.

3. MocGraph provides implementations of algorithms for graph analysis — for example, algorithms to determine existence of cycles in a graph, and compute all-pairs shortest path results, to name a few.

The latest version of MoCGraph extends its earlier released version by providing the following new graph types — tree, rooted tree, ordered tree, and k-ary tree [18].

An important plug-in to MoCGraph is a package called `mocgsched`, which stands for *MoCG*raph *sched*uling support. Section 2.4.2 describes this plug-in.

2.4.2 `mocgsched` Plug-in

This plug-in augments the MoCGraph package by providing data structures used to represent schedules for dataflow graphs. The scheduling strategies, which are graph transformations, can make use of these data structures to represent the schedules derived from the given application graphs. It must, however, be noted that scheduling transformations that are specific to dataflow are not part of the MoCGraph package — instead, packages that are intended for features specific to dataflow can build on features of MoCGraph and `mocgsched` to provide such dataflow-specific transformations. This is the approach taken in DIF for many of its scheduling features.

The latest version of `mocgsched` includes the representation that supports looped schedules (as in the previous version of MoCGraph [35]), along with a data structure for storing and manipulating GSTs. GSTs make use of various types of tree data structures, which have been added recently as mentioned in Section 2.4.1. The support for GSTs in `mocgsched` has provided an important foundation for CFDF-based modeling and functional simulation features in the DIF package [64].

2.5 DICE: The DSPCAD Integrative Command-Line Environment

The DSPCAD Group has developed the *DSPCAD integrative command line environment* (*DICE*), which is a package of utilities that facilitates efficient management of

software projects [9]. The objective of DICE is to provide a flexible, lightweight environment for the design, implementation, testing, and integration of engineering software, with a specific orientation towards projects that employ heterogeneous programming languages and cross-platform design methods. The DICE package has been used extensively as an environment for software development for many of the projects involved in this thesis.

DICE is implemented as a collection of utilities that are in the form of Bash scripts, C programs, and Python scripts. Therefore, facilities for interpreting/compiling these languages must be available to use all of the capabilities in DICE. DICE is developed with significant attention to cross-platform operation. Platforms on which DICE is used actively include Linux, MacOS, Solaris, and Windows (equipped with Cygwin).

Apart from DICE utilities that facilitate working in a command line environment, such as those for directory navigation, another important aspect of the DICE is to provide a lightweight and flexible unit testing environment. This environment is lightweight in that it requires minimal learning of new syntax or specialized languages, and flexible in that it can be used to test source code in many languages, including C, Java, Verilog, and VHDL [9].

In DICE, the test suite for a project consists of an *individual test subdirectory* (*ITS*) for each of the unit tests in the suite. An ITS in general contains files that provide documentation related to the test, a script to perform any compilation steps necessary to build the test, a script to execute the test, and files that contain the expected output and errors resulting from correct execution of the test. The output and errors resulting from the actual execution of the test are also stored in the same ITS, and compared with the expected

behavior of the test. More information regarding ITS structure, and DICE utilities for unit testing can be found in [9].

Although testing is an established concept in the software engineering field, integrating testing rigorously with software development instead of testing being applied as an afterthought to the coding process is a relatively new paradigm. DICE facilitates the application of this paradigm.

A companion package of DICE, called `dicelang`, provides a collection of language-specific plug-ins that extend the features of DICE, and provide new features to facilitate efficient software project design, implementation, and testing for selected programming languages [9].

2.6 Summary

In this chapter, we have provided background information on dataflow modeling, the dataflow interchange format, generalized schedule trees, the MoCGraph package, and the DICE package. This background and the corresponding tools are fundamental to the work developed and presented in the remainder of this thesis.

Chapter 3

Topological Patterns for Specification and Analysis of Dataflow Graphs

Tools for designing signal processing systems with their semantic foundation in dataflow modeling often use high-level GUIs or text based languages that allow specifying applications as directed graphs. Such graphical representations serve as an initial reference point for further analysis and optimizations that lead to platform-specific implementations. For large-scale applications, the underlying graphs often consist of smaller substructures that repeat multiple times. To enable more concise representation and direct analysis of such substructures in the context of high-level DSP specification languages and design tools, we have developed the modeling concept of *topological patterns*, and proposed ways for supporting this concept in a high-level language. This chapter shows how the DIF language can be augmented with constructs for supporting topological patterns, and topological patterns can be effective in various aspects of embedded signal processing design flows using specific application examples.

3.1 Introduction

As mentioned in Section 1.2, DSP-oriented dataflow design tools typically allow high-level application specification, software simulation, and possibly synthesis for hardware or software implementation. These tools employ high-level description languages for application specification. These languages, which may be either GUI or text based,

provide syntactic and semantic constructs for specifying graphical representations of DSP applications. Such graphical representations are then parsed and converted into intermediate representations suitable for further processing.

In this work, we address the problem of representing large-scale and scalable dataflow graphs that have complex topologies. Such graphs comprise of various kinds of functional substructures that are parameterizable and can be represented in terms of concise, scalable specifications.

For example, the dataflow graph of an N-point fast Fourier transform (FFT) algorithm consists of a combination of scaled versions of a well-known pattern called the *butterfly diagram* [52], and a systolic array is a *mesh* of computing elements having a specific dataflow structure that can solve problems such as QR-decomposition based recursive least square adaptive filtering, and minimum variance distortionless response beamforming [43]. We identify such common structures in dataflow graphs as *topological patterns*, and treat this kind of pattern as a first class citizen in the modeling process. Furthermore, we demonstrate and experiment with the use of topological patterns in the DIF, a textual design language and associated software package for specification, analysis, and synthesis based on DSP-oriented dataflow models of computation [36], [64].

Topological patterns not only permit scalable specifications of dataflow substructures but also expose the underlying graph structure explicitly to the corresponding design tool. This allows design tools to exploit any analysis or optimization advantages offered by the substructures without having to "discover" those structures through additional levels of pre-processing analysis. Some of the key components of the design flow that can potentially benefit from explicitly exposed patterns include various kinds of scheduling

transformations, and techniques for buffer memory optimization. Furthermore, by making it easier and more efficient to apply substructure-specific analysis techniques, programming support for topological patterns encourages the development of such analysis techniques, and provides a natural interface for reusing them across different applications and tools.

The concept of topological patterns is elaborated in Section 3.3. In Section 3.4, we describe how we extend the DIF language to integrate topological patterns as a first class modeling construct. In Section 3.5, we show how topological patterns can be used by dataflow based design tools for dataflow graph analysis and transformations. We show how topological patterns can be used for graph analysis; representing equivalent HSDF graphs of application graphs modeled using SDF and CSDF models; extracting implementation-specific features; exploring trade-offs for an FPGA implementation of a JPEG image compression application; representing schedules; and experimenting with pattern-specific schedules.

3.2 Related Work

Block diagrams are a natural and convenient way of describing DSP algorithms, and hence, DSP systems designers find it intuitive to have a high-level application specification that captures such a description. GUI based dataflow languages try to capture this intuition using visually appealing representations, while text based languages provide syntax that looks similar to common procedural languages, such as C, but with semantic constructs that model the dataflow structure of DSP block diagrams. To effectively han-

dle the increasing complexity of signal processing system design, these languages must provide frameworks for modular and scalable representations with sufficient expressive power.

Earlier research efforts have focused on supporting commonly used and highly expressive constructs from procedural languages, such as recurrences, iteration, and conditionals, in dataflow-oriented languages [45]. Subsequent work includes evolution of various textual languages for DSP system design, such as SILAGE [82], StreamIt [79], and CAL [26]. The StreamIt language provides high-level, architecture-independent abstractions for streaming applications geared toward large-scale program development. The CAL language is an actor-oriented language, which has been applied actively for field programmable gate array (FPGA) implementation and reconfigurable video coding applications. The SILAGE language has been developed with an emphasis on support for high-level synthesis and multidimensional signal processing.

While these previous efforts have employed useful techniques for deriving and exploiting various types of specialized dataflow substructures within their respective compilers, they lack a general method for explicit and scalable representation of such substructures by the programmer. Such a programming interface for topological patterns is essential to capture the broad range of relevant patterns in ways that are scalable, and flexibly extensible to accommodate new types of patterns as they emerge from new applications and modeling techniques. Our concept of topological patterns is designed precisely to bridge this gap.

In other prior work, higher-order functions have been shown to permit elegant construction of structured subsystems in dataflow representations [48]. Higher-order func-

tions are functions that take functions as inputs or produce functions as outputs. Topological patterns provide a related but technically different approach since topological patterns operate on generic directed graph vertices (e.g., `nodes` in DIF), where the actual binding to actor functionality and associated actor parameter values is specified separately, possibly through additional *parameter propagation patterns* (*PPP*s). Thus, unlike higher-order functions that take functions as arguments, topological patterns take only generic graph vertices (or arrays of such vertices) as arguments. Furthermore, our development of topological patterns is tightly integrated with textual graph representation and arrays of graph vertices and edges, which are useful for providing scalable representations and managing large-scale designs.

Perhaps the most closely related prior work is that on support for arrays of vertices and edges in the DIF language with array construction syntax and semantics similar to those in the C language [19]. These constructs provide a useful shorthand notation for specifying related groups of graph elements (nodes or edges) as arrays in which individual elements can be easily indexed. A typical `elementID` in the DIF specification (see Fig. 2.4) when referred to as `baseName[N]`, generates an array of N elements. For example, `tap[N]` in DIF specifies an array `tap` of N nodes. The ith node, where $i = 0, 1, \ldots, N-1$, can be accessed using its index as `tap[i]`. However, in this *first-version* array support within DIF, there is no mechanism for instantiating (declaring) collections of related edges automatically as structured mappings among corresponding subsets of nodes. It is also not possible to configure parameters across arrays of actors as functions of the array indices. These two features — scalable, programmatic instantiation of graphical substructures, and association of parameter values — are provided by our development of

topological patterns.

This development is orthogonal to the existing support for syntactic and semantic hierarchy in the DIF language, which allows constructing hierarchical dataflow graphs. The focus here is to allow the designer to specify already identified topological patterns in the design and expose such patterns to the enclosing design tool or design process, which is generally not achieved through conventional methods for using hierarchical dataflow graphs.

This chapter presents formulation of the concept of topological patterns and its application to dataflow modeling. To prototype this concept in DIF, we build upon the first-version framework of arrays in DIF, and introduce new modeling and language constructs that are dedicated to topological patterns. We also demonstrate the use of topological patterns to derive efficient implementations.

A preliminary version of this work was presented in [71], while the extended work was presented in [72]. The work presented here, and in [72] goes beyond the developments of [71] by significantly extending the development of applications of topological patterns. Specifically, we explore the utility of topological patterns in analyzing dataflow graphs and extracting implementation-specific features. We also use topological patterns to represent schedules obtained after applying scheduling transformations to dataflow graphs, and derive more efficient implementations from such representations. Additionally, we show how specific topological patterns can be exploited to construct structured schedules, and how designers can experiment with corresponding scheduling trade-offs.

3.3 Topological Patterns

We have developed the concept of topological patterns for concise specification of functional structures at the dataflow graph (inter-actor) level. Topological patterns provide a scalable approach to specifying regular functional structures in a manner that is analogous in some ways to the use of design patterns in object oriented software [28], but with additional properties associated with being formally integrated with the framework of dataflow. This integration allows not only for specification of functional patterns but also for their analysis and optimization as part of the larger framework of dataflow.

Topological patterns build on the concepts of *graph element arrays*, which allow indexed families of graph elements to be declared and treated as single units for purposes of graph construction and analysis. As with arrays in conventional programming languages, graph element arrays can be single- or multi-dimensional. Additionally, they can be parameterized in terms of dataflow graph attributes so that their sizes and other characteristics can be conveniently adapted.

3.3.1 Topological Patterns in Signal Processing

We motivate the utility of incorporating topological patterns into dataflow frameworks for DSP system design by illustrating the pervasive nature of these patterns in the domain of DSP. We have already discussed a few such patterns in Section 3.1 — in particular, the `butterfly` and `mesh` patterns, which have applications in FFTs and systolic arrays, respectively. Additionally, the `chain` pattern is one of the most commonly found topological patterns. This pattern finds applications in modeling multi-stage sample rate

converters, delay lines in FIR filters, or configurations of pipeline stages. A chain of delay blocks, a chain of adders, and an `array` of filter taps collectively specify a complete FIR filter when connected together. A natural extension of this pattern is a 2-dimensional mesh structure. Such a structure is of particular use to model DSP architectures in which data flows across a network of processing elements connected to form a 2-D grid such as a systolic array, as discussed earlier in Section 3.1 [43].

A `ring` pattern represents a cycle in a graph as may be introduced by a phase-locked loop [47] or more generally, a `feedback loop` in the system. The FFT block is one of the most abundantly found blocks in DSP systems. An N-point FFT computation involves FFT computation stages of smaller dimensions that can be implemented as scaled versions of the 2-point FFT. These FFT stages resemble a butterfly-like pattern [52]. Such patterns can also be found in other applications, such as sorting networks [18]. Entropy encoding algorithms such as Huffman coding make use of the `binary tree` structure, a commonly found data structure in many computer algorithms [38]. A pattern in which edges connect a source node to multiple sink nodes can be termed as a `broadcast` pattern. This pattern finds use in applications that have computation blocks in multiple stages with blocks in one stage connected to those in the subsequent stage. Such patterns are observed in multi-layer neural networks used for pattern classification [24] and trellis coding algorithms used in digital communication [47]. It is also common to find its dual, the `merge` pattern, which connects multiple source nodes to a single sink node. Applications may also have parallel connections between corresponding nodes in adjacent stages. We identify this pattern as a `parallel` pattern in which edges form a one-to-one correspondence between nodes in two different sets. We also identify a pattern called

`multiedge` that creates multiple edges between a given pair of nodes.

3.3.2 Parameter Propagation

An important feature to support in conjunction with topological patterns is a mechanism for structured *parameter propagation*, whereby any parameters associated with the vertices in a topological pattern can be set as a function of the vertex indices (i.e., indices associated with the underlying vertex ordering that is input to the pattern instance). For example, Fig. 3.1 shows an array of 5 actors identified as `A_0_`, `A_1_`, . . . , `A_4_`, where each actor has a parameter `angle` associated with it that is an affine function of its index in the array. Such a parameter assignment can be implemented in a scalable, reusable, and explicitly-recognizable form as a designated PPP — in particular, a PPP for affine mappings of parameter values across ordered vertices. Such an affine PPP can find use in specifying elements of a *steering vector* corresponding to each sensor in a sensor array while estimating the direction of arrival of the received signal [33].

One of our important motivations for using topological patterns is to provide for compact, scalable representations for large dataflow graphs. It is common for such large graphs to have actors with the same functionality that scale in number with the size of the application graph. These actors may have functional parameters (for example, the parameter *angle* associated with the actors in Fig. 3.1) that determine some of their functional aspects and also distinguish them from the other actors of the same (parameterized) functionality. It may be inconvenient to specify such parameters individually for all of the actors with growing size of an application graph (in fact, such individualized parameter

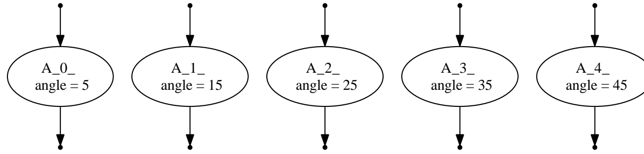


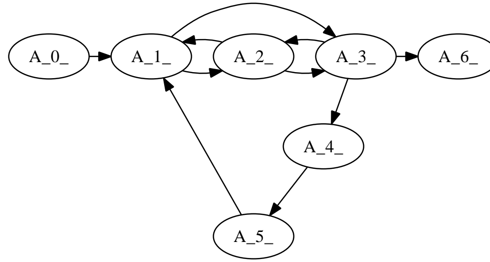
Figure 3.1: Configuring an array of nodes with a PPP.

specification violates the compactness objective of topological patterns). PPPs can help here by providing a compact representation format that can be used to set parameters associated with actors in the large graphs that are represented by the associated topological patterns.

In terms of implementation in the DIF language, just as component attributes and topological patterns can be either user-defined or built-in, similarly commonly-used PPPs can be absorbed into the language as built-in PPPs, while users have the flexibility to incorporate specialized PPPs by linking their interpretation (propagation functionality) to segments of customized Java code.

3.4 Topological Patterns in DIF

We extend the DIF language by supporting topological patterns as first class citizens in the modeling framework. These patterns can be defined as built-in patterns, which are recognized and processed through corresponding keywords in the language. To enable more flexible application of patterns, we also support declaring arbitrary (user-defined) patterns, whose associated graph construction functionality can be carried out through procedural language code (Java or C in the case of DIF) that is linked with the graph specification.



(a)

```
topology {
    nodes = A[7];
    edges = e0(A[0], A[1]), e1(A[3], A[6]),
            ring_0[5] -> ring(A[1:1:5]),
            ring_1[3] -> ring(A[1], A[3], A[2]);
}
```

(b)

Figure 3.2: Overlapping patterns: (a) a graph topology having two `ring` patterns that have three nodes common to them, and (b) a corresponding DIF representation.

We have added, as built-in topological pattern specifiers, new keywords in DIF corresponding to topological patterns that are relatively common in signal processing systems. These keywords, such as `ring`, `parallel`, `merge`, `butterfly`, `broadcast`, and `chain`, allow specifying patterns explicitly as part of the `topology` block in a DIF specification. When declaring an instance of such a pattern, the designer must provide a sequence of vertices and an optional set of parameter values. The pattern construct, when parsed, generates the required edges, inserting the new edges into the graph that is being constructed. The pattern construct also configures the underlying nodes using the parameter propagation mechanism explained in Section 3.3.2.

A typical way to specify a sequence of nodes is through the use of DIF notation

for representing nodes in an array. For example, for an array of 7 nodes, specified as `A[7]`, we can specify that 5 of its elements form a ring structure using the construct `ring(A[1:1:5])` in the `topology` block of the DIF code as shown in Fig. 3.2. The argument `A[1:1:5]` to the construct `ring`, specifies an array of nodes starting from `A[1]`, ending at `A[5]`, and having an array index increment of 1. In general, the syntax `baseName[i:j:k]` denotes an array of elements in an array `baseName` starting from the index i, ending with the index k, and having an array index increment of j. Note that, outside of the pattern instantiation construct, the nodes in the array `A` can be accessed by their indices to create edges that are not part of the `ring` pattern. Thus, one can flexibly embed patterns within arbitrary structures including structures that contain other patterns.

It is also possible to generate multiple patterns that have one or more nodes common to them, as shown in Fig. 3.2. It is, thus, possible for the designer to effectively identify one or more types of overlapping topological patterns in the application graph.

3.5 Applications of Topological Patterns

As described earlier, we envision topological patterns to offer a wide range of advantages at various stages of the design flow from modeling to platform-specific implementation. In Sections 3.3 and 3.4, we have identified topological patterns in various DSP system specifications. In the following subsections, we examine other aspects of the design flow where topological patterns can be effectively used.

3.5.1 Graph Analysis

The explicit specification of known graphical structures as topological patterns can significantly facilitate various types of dataflow graph analysis algorithms. For example, one of the first and most important steps in many dataflow graph scheduling strategies is to analyze the input graph to identify strongly connected components (SCCs). An SCC is a maximal subgraph in which every pair of distinct nodes is connected through a cyclic path. It is often useful to cluster SCCs — for example, SCCs can be clustered to improve scheduling of SDF graphs (e.g., see [37]). Such clustering of SCCs is typically performed in order to obtain a top-level DAG. For a directed graph $G = (V, E)$, SCCs can be identified using an algorithm with a time complexity of the order of $\Theta(|V| + |E|)$ (see [18] for more details on the definition of the Θ-notation as well as algorithm to find SCCs in a directed graph).

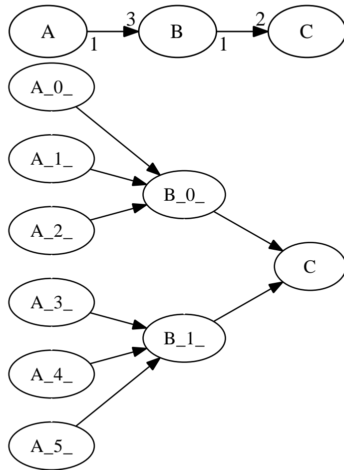
Consider an application graph that contains multiple feedback paths that can be modeled and specified using the `ring` pattern. A `ring` represents a cycle in the graph and hence, a subset of vertices that form an SCC. Such a cycle, when directly specified as a `ring` can be readily reduced into a single clustered actor. A `ring` with n nodes in it, when clustered into a single node, effectively reduces the number of nodes in the graph G by $n-1$. Suppose that a graph G has many `ring` patterns that have been identified in the graph specification. Then by identifying these rings in constant time, which an analysis tool can do easily from explicit topological pattern specifications, the number of nodes and edges in the graph can be reduced significantly. This can lead to more efficient SCC computation, especially for large graphs.

3.5.2 Representing HSDF Graphs

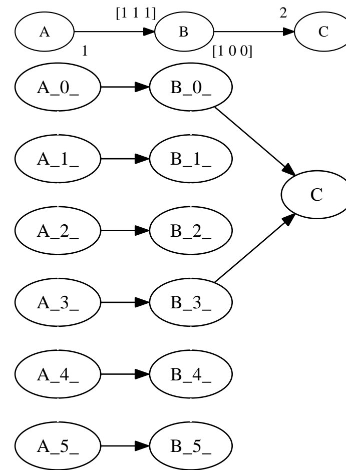
Many techniques devised for generating multiprocessor schedules from SDF graphs require that the given dataflow graph be transformed into an equivalent HSDF graph (e.g., see [76]). An HSDF graph is an SDF graph in which every actor consumes (produces) a single token from (on) each input (output) port. Techniques for converting an SDF graph into equivalent (for scheduling purposes) HSDF graphs have been developed in [46]. Such techniques are useful, because equivalent HSDF graphs can expose parallelism much more effectively compared to their more compact SDF counterparts.

Unfortunately, equivalent HSDF representations can scale very inefficiently — the size of an equivalent HSDF graph is in general not polynomially bounded in the size of the corresponding SDF graph [58]. Representing such HSDF graphs becomes a cumbersome exercise, as such representations require large amounts of storage to maintain and large amounts of computation time to process them. For a large HSDF representation, it is difficult for a design tool to traverse the HSDF representation and make effective use of it within a reasonable amount of time. Topological patterns can help in this situation by providing concise representations to expose repetitive structures within HSDF representations, thereby improving the efficiency of HSDF-based schedulers.

For example, Fig. 3.3(a) shows an SDF graph that models a simple sample rate converter, and its equivalent HSDF graph (below). Here, actor `B` is a decimator with a decimation factor of 3. Fig. 3.3(c) shows a DIF specification of this HSDF graph using topological patterns. Fig. 3.3(b) and (d) show an equivalent CSDF graph model with its HSDF graph and associated topological-pattern-based DIF specification. In the CSDF



(a) SDF graph to HSDF graph



(b) CSDF graph to HSDF graph

```
topology {
    nodes = A[6], B[2], C;
    edges = e0[3] -> merge(A[0:2], B[0]),
            e1[3] -> merge(A[3:5], B[1]),
            e2[2] -> merge(B[0:1], C);
}
```

(c) DIF `topology` block for HSDF graph in (a)

```
topology {
    nodes = A[6], B[6], C;
    edges = e_par[6] -> parallel(A[0:5], B[0:5]),
            e_mrg[2] -> merge(B[0:3:3], C);
}
```

(d) DIF `topology` block for HSDF graph in (b)

Figure 3.3: A sample rate converter.

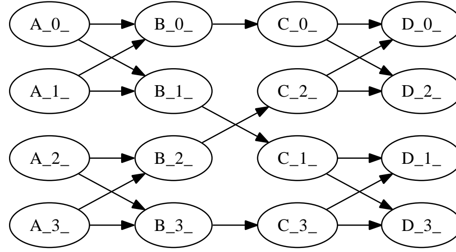
representation, actor B provides a decimation by a factor of 3. Actor B consumes input tokens on every firing while producing an output token only on every third firing, starting with the first firing. As this example illustrates, topological patterns can provide a concise and scalable representation of equivalent HSDF graph representations for SDF and CSDF graphs.

It should, however, be noted that a graph representation using topological patterns is in general not unique. Depending on the set of available topological patterns, it may be possible to have multiple functionally-equivalent representations of a given dataflow graph using topological patterns. In the case of Figure 3.3(a), for example, it may be possible to use a `tree` pattern if the associated design tool supports it.

Structured representations of HSDF graphs can also enable efficient tuning of HSDF graph representations in terms of application parameters. For example, for the dataflow graph in Figure 3.3(b), it can be observed that if the decimation factor of actor B is changed, then the DIF representation for the HSDF graph can be updated by simply changing the numeric arguments to the topological patterns used in its representation. In general, for a decimation factor of D, the production rate of actor B in Fig. 3.3(b) is $[1\ 0\ 0\ \cdots\ 0]_{1\times D}$ and the equivalent HSDF graph for this CSDF graph has the following specification, where D is a suitably-declared parameter.

```
topology {
    nodes = A[2D], B[2D], C;
    edges = e_par[2D] -> parallel(A[0:2D-1], B[0:2D-1]),
            e_mrg[2] -> merge(B[0:D:D], C);
}
```

Thus, topological patterns provide streamlined representations that are concise, tunable, and scalable, and are particularly useful for complex graph structures, such as those




```
topology {
    nodes = A[4], B[4], C[4], D[4];
    edges = fft2_0[4] -> butterfly(A[0:1], B[0:1]),
            fft2_1[4] -> butterfly(A[2:3], B[2:3]),
            fft4[8] -> butterfly(C[0:3], D[0:3]),
            e_par[4] -> parallel(B[0:3], C[0:3]);
}
```

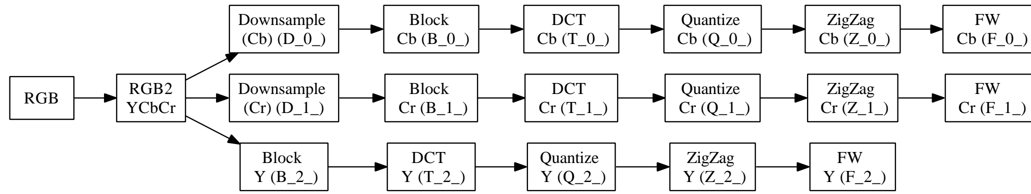
Figure 3.4: Dataflow graph for a 4-point fast Fourier transform and the `topology` block in its DIF specification.

found in equivalent HSDF graphs arising from multirate SDF, and CSDF models.

3.5.3 Extracting Implementation-Specific Features

Fig. 3.4 shows an HSDF graph that models a 4-point FFT application [52], and the `topology` block in its DIF specification. Note the underlying topological patterns — `butterfly` and `parallel` — in the graph. It should also be noted that `butterfly(C[0:3], D[0:3])` is a scaled version of a `butterfly` pattern with just 4 nodes, and is equivalent to two `butterfly` patterns formed by the node subsets {`C_0_, C_2_, D_0, D_2_`} and {`C_1_, C_3_, D_1_, D_3_`}.

Apart from scalability, there is another useful feature in this HSDF graph representation. In particular, the bi-partite nature of both the patterns — `butterfly` and `parallel` — allows us to generate a pipelined implementation of this application. Here,




```
topology {
    nodes = RGB, RGB2YCbCr, D[2], B[3], T[3], Q[3],
            Z[3], F[3];
    edges = e0(RGB, RGB2YCbCr),
            e1[3] -> broadcast(RGB2YCbCr, D[0:1:1], B[2]),
            Cb[5] -> chain(D[0], B[0], T[0], Q[0], Z[0],
            F[0]),
            Cr[5] -> chain(D[1], B[1], T[1], Q[1], Z[1],
            F[1]),
            Y[4] -> chain(B[2], T[2], Q[2], Z[2], F[2]);
}
```

Figure 3.5: JPEG encoder and the `topology` block in its DIF specification.

segments `A`, `B`, `C`, and `D`, consisting of nodes `A[0:3]`, `B[0:3]`, `C[0:3]`, and `D[0:3]`, respectively, may be considered as pipeline stages of the FFT implementation. This inherent pipelined nature of the FFT application can be identified easily using the bi-partite nature of the underlying topological patterns. Of course, for FFTs, many efficient implementations have been developed in the literature, and the use of topological patterns does not add any obvious value to the large library of existing FFT implementation techniques. However, this example succinctly illustrates the general potential of topological patterns for exposing useful implementation options more clearly and efficiently to designers and to analysis modules within design tools.

3.5.4 Exploring Implementation Trade-offs

Fig. 3.5 shows a JPEG encoder along with the `topology` block in its DIF specification [83]. It effectively employs the `broadcast` and `chain` patterns in its representation. The JPEG compression algorithm downsamples both the chroma (C_b and C_r) components before processing them. Except for this, all three components (both chroma and luma Y) are processed through functionally similar chains of blocks. The input pixels are grouped into blocks that are then transformed using the discrete cosine transform (DCT), quantized, and scanned in a zigzag order. In particular, the chroma components may be processed using shared functional modules that are clearly exposed by the `topology` block. Without the use of topological patterns, this observation may not be clear to a designer until the entire graph is carefully traced. For a design tool, this observation may go entirely unexploited because such high-level structure can be difficult to extract automatically from unstructured specifications.

The problem of identifying such graph structure is related to the *graph isomorphism problem*, which is the problem of detecting whether two graphs (or two subgraphs from the same or different graphs) can have their vertices and edges placed in one-to-one correspondence with one another in a manner that maintains edge-vertex connectivity relationships. There are no known polynomial time algorithms for the graph isomorphism problem (e.g., see [30]).

For the JPEG encoder example, we can exploit the potential for resource sharing — which is exposed explicitly at a high-level through the use of topological patterns — to develop a streamlined FPGA implementation. Awareness of the high-level topological

Table 3.1: Performance and resource utilization trade-offs for FPGA implementation of a JPEG encoder.

JPEG Encoder	Throughput (samples /cycle)	FPGA Resource Utilization		
		Slices (out of 13696)	18kB BRAM	18x18 MULT
Non-shared	0.159	8070 (58%)	41	30
Shared	0.159	6088 (44%)	37	22

pattern in this application allows for systematic trade-off analysis between two design options — one with shared resources for chroma component processing, and another without shared resources.

An analysis of the high-level dataflow specification suggests that downsampling of chroma components would ensure that the chain processing Y component is the bottleneck and hence, the throughput should remain unaffected even when the C_b and C_r components are processed using shared functional modules. Precise modeling of the shared-resource implementation of the JPEG encoder requires that the SDF design in Fig. 3.5 be transformed to expose more detail. For example, the design can be converted into an equivalent CSDF design in which buffers between functional modules are duplicated and alternate buffers are used in successive schedule iterations. For more background on this form of CSDF-based structural modeling, we refer the reader to [10].

From inspection of the CSDF intermediate model, it can be reasoned that the buffer requirement would remain unchanged across both designs (shared- versus separate-resource). However, we expect that the shared-resource version of the JPEG encoder would result in a net reduction in BRAM (*block random access memory*) utilization.

This analysis can be confirmed from the resource utilization and throughput for

shared- and separate-resource JPEG encoder implementations on the Xilinx Virtex-II Pro FPGA, as shown in Table 3.1 [71]. The base clock rate for our experiments is 40 MHz. Even though actor-level resource sharing is often avoided in FPGA implementation due to the relatively high costs of multiplexing and routing resources (e.g., see [78]), resource sharing for a subgraph in a dataflow representation can result in conservation of FPGA resources that overrides the multiplexing overhead. The shared-resource JPEG encoder uses less BRAM than the separate-resource version, which can be attributed to the shared DCT block. Also, the shared-resource version uses fewer (18×18) multiplier units by employing shared downsampling, DCT, and quantization modules.

As expected — from the aforementioned bottleneck analysis — both versions of the JPEG encoder achieve the same throughput. In particular, the Y component remains as the system bottleneck even when the C_b and C_r components are processed using shared FPGA resources. Our experiments thus demonstrate concretely how topological patterns can provide a *formal path* from scalable application analysis to the systematic exploration of implementation trade-offs in the design and implementation of signal processing systems on a relevant target platform.

3.5.5 Representing Schedules

The utility of topological patterns is not limited to representation of application graphs alone. Their utility can be extended to create concise and parameterizable representations of structures typical to schedules for certain application graphs. This can be of particular importance in functionally simulating application graphs, and porting schedules

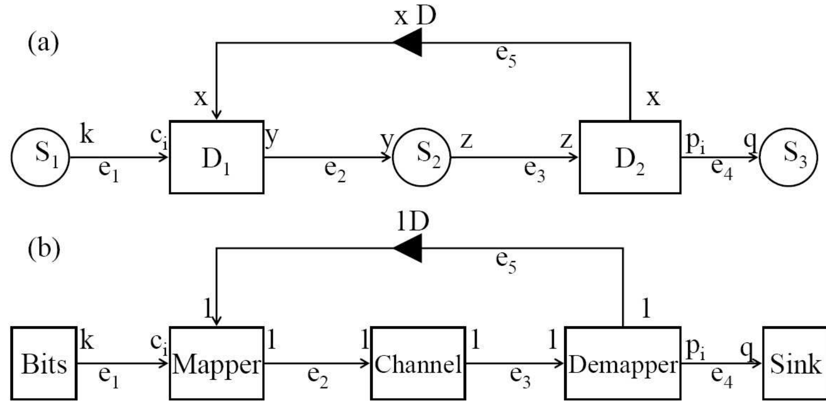
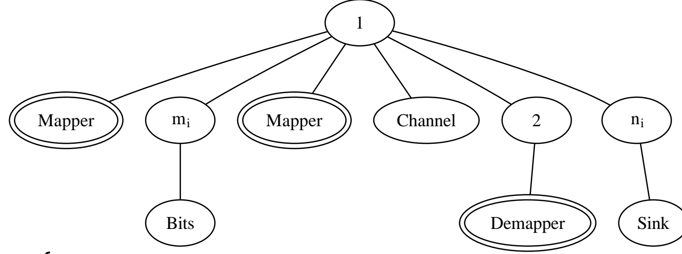


Figure 3.6: Dataflow graphs for (a) the generic class of applications under consideration, and (b) a simplified adaptive modulation scheme.

across design tools or languages. We elaborate on this using the following example.

We consider a class of applications typically found in the domain of wireless communications, and signal processing systems that exhibit dataflow graph structures similar to the one shown in Fig. 3.6(a). A typical example of this type is that of the adaptive modulation scheme (*AMS*) shown in Fig. 3.6(b). The AMS is a dynamic communication application, which is an important part of modern wireless standards such as the *world-wide interoperability for microwave access* (*WiMAX*) [3] and *3rd generation partnership project — long term evolution* (*3GPP—LTE*) [1] standards. For details of AMS, we refer readers to [70]. There exist other applications that exhibit the general dataflow structure illustrated in Fig. 3.6(a), such as prediction error filters [33] and systems for frequency domain block adaptive filtering [75]. Such dataflow graphs can be efficiently simulated by constructing parameterized looped schedules (*PLS*s) as described in [70] and [42]. We will revisit the AMS application, and show how it can be modeled using CFDF model in Section 4.3. We will further elaborate on constructing a PLS for the AMS application in




```
topology {
    nodes = Root, N[6], B, D, Snk;
    edges = e0[6] -> broadcast(Root, N[0:5]),
            e1(N[1], B), e2(N[4], D), e3(N[5], Snk);
}
```

Figure 3.7: A PLS for the application in Fig. 3.6(b), and the `topology` block in a corresponding DIF representation. Table 3.2 provides parameters associated with each `node` in the DIF specification.

Section 5.3.

Fig. 3.7 shows a PLS for the AMS application. A PLS of this type is of particular importance since it can capture the dynamic dataflow behavior inherent in the application without compromising compile-time analysis. It is possible to perform useful analysis (e.g., estimation of upper bounds on total buffer memory requirements) for PLSs at compile-time.

In Fig. 3.6(a), the consumption rate c_i and production rate p_i can vary over finite ranges of positive integer values with known upper bounds $c_{\max}$ and $p_{\max}$, respectively. The subscript i in the symbols p_i and c_i represents the dependence of this production and consumption rate pair on the actor execution index i — thus, p_i represents the number of tokens produced onto e_4 in the ith execution (*firing*) of D_2, and c_i represents the number of tokens consumed from e_1 during the ith firing of D_1. In Fig. 3.7, the loop counts m_i and n_i are computed dynamically.

Table 3.2: Actors and loop counts associated with nodes in the PLS graph representation. Here, NULL indicates an internal node in the GST that does not have any actor associated with it.

Node	Actor	Loop Count
Root	NULL	1
N[0]	Mapper	1
N[1]	NULL	m_i
N[2]	Mapper	1
N[3]	Channel	1
N[4]	NULL	2
N[5]	NULL	n_i
B	Bits	1
D	Demapper	1
Snk	Sink	1

In the context of this AMS example, topological patterns help not only in specification of the application dataflow graph using the `ring` pattern, which can be used to identify the pair of dynamic actors easily, but also representation of generated PLSs using `broadcast` patterns with hierarchical nodes for SDF-schedules, as shown in Fig. 3.7. For such a well-structured schedule representation, it is possible to hand-tune an implementation and use that representation explicitly for applications having similar dataflow behavior instead of traversing the GST using a generic process to derive a software or hardware implementation. In this case, topological patterns provide a framework by which hand-tuned schedules can be formally specified and reused across different applications or target platforms.

Table 3.3 shows a comparison between simulation times using GST traversal and hand-tuned pattern-specific implementation for the PLS in Fig. 3.7. These simulation experiments — the results of which are presented in Table 3.3 — differ from related experiments that we have reported on previously (e.g, in [72]) in that we have eliminated

Table 3.3: Average simulation times for different sink control conditions (numbers of tokens consumed by the sink) for the PLS in Fig. 3.7 using (1) GST traversal, and (2) a hand-tuned pattern-specific schedule.

Sink control condition	Average simulation time (ms)		Improvement
(Number of tokens)	(1)	(2)	(%)
10000	73	32	56.16
20000	90	47	47.78
50000	148	62	58.11
100000	248	93	62.50

some of the common overheads by suppressing printing of routine debug and status information. This allows us to determine the extent of effect of these two simulation strategies on simulation speed, and compare them more precisely. It can be seen that the hand-tuned software implementation results in faster simulations by a factor of up to 62%. Furthermore, through its formulation in the framework of topological patterns, the hand-tuned implementation can be analyzed, maintained, ported, and reused effectively across different design contexts.

3.5.6 Experimenting with Pattern-Specific Schedules

When specifying signal processing systems, an important motivation for using topological patterns is to facilitate application of pattern-specific transformations, such as pattern-specific scheduling transformations. In such a context, it can be useful for a design tool to provide features that allow the designer to experiment with various "scheduling patterns" at a high level of abstraction. Since topological patterns provide well-defined, scalable topological information, one can generate a structured schedule from a given pattern. We demonstrate this application of topological patterns through an example of a

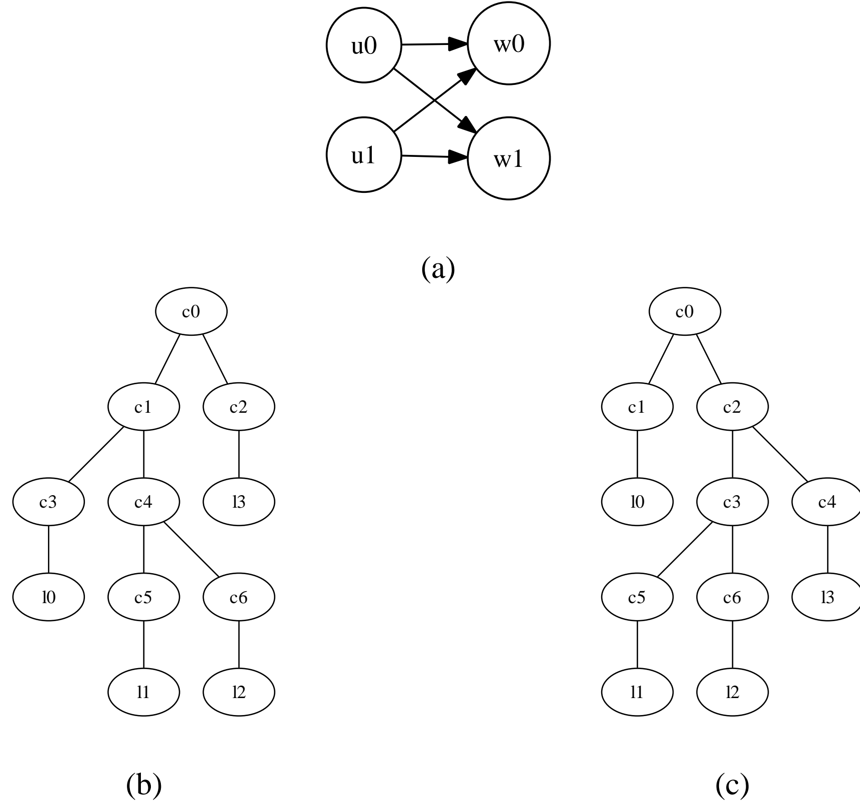


Figure 3.8: (a) An SDF graph with a `butterfly` pattern. (b)-(c) two possible GST structures using schedules that are based on acyclic pairwise clustering (iteratively clustering two actors at a time).

commonly used `butterfly` pattern.

Consider an SDF graph having a `butterfly` pattern, as shown in Fig. 3.8(a). One commonly used scheduling transformation involves applying clustering transformations on one pair of connected actors at a time such that no cycle is introduced in the resultant graph, and then generating a hierarchical schedule for the given application graph by iteratively applying such acyclic pairwise clustering (APC) [11]. In case of SDF graphs, a group of actors can be *SDF-clustered* if its component actors can be scheduled together (i.e., the group can be scheduled as a single unit in the overall schedule for the graph) without introducing deadlock [11]. It can be observed that more than one schedule can be

generated using APC depending on the pair of actors clustered at every stage of scheduling. In case of SDF graphs, the total buffer memory requirements depend upon the choice of a schedule, and in general, a schedule that has minimum total buffer memory requirements is desirable in many applications. A scheduling technique based on APC called acyclic pairwise grouping of adjacent nodes (APGAN) has been described in [11] that chooses a pair of actors to be clustered at every stage of scheduling using a metric based on repetition counts of the actors in the graph. This heuristic is widely used and attempts to minimize the total buffer memory requirements. We refer readers to [11] for more information on SDF-clustering, and SDF scheduling heuristics that are based on APC including APGAN.

A useful class of SDF schedules is that of single appearance looped schedules, as described in Section 2.1. Let $G(V, E)$ denote the graph in Fig. 3.8(a), where

$$V = \{u_0, u_1, w_o, w_1\}, \text{ and } E = \{(u_0, w_0), (u_0, w_1), (u_1, w_0), (u_1, w_1)\}, \tag{3.1}$$

and suppose that we apply APC to the graph. Based on the steps involved in APC, there are only two possible GST structures for this example. These two structures are shown in Fig. 3.8(b) and (c). Here, each c_i, $i = 0, 1, \cdots, 6$, denotes a loop count, while each l_i, $i = 0, 1, \cdots, 3$, denotes the actor pointed to by a leaf node in the GST. The existence of exactly two unique GST structures for this example can be verified from the following observations regarding the operation of APC (see [11] for further details on the operation of APC for SDF graphs).

1. Let $U = \{u_0, u_1\}$, and $W = \{w_0, w_1\}$. Then we can describe the graph $G(V, E)$ as

$$V = U \cup W, \text{ and } E = U \times W. \tag{3.2}$$

2. Let $e \in E$ denote the group of actors clustered during the first clustering step. Then, $l_1 \in U$, and $l_2 \in W$. This follows from the bipartite nature of the `butterfly` pattern.

3. Following the first APC step, operation of APC ensures that $l_0 \in (U \setminus \{l_1\})$, and $l_3 \in (W \setminus \{l_2\})$. This is because clustering actors a and b such that $a \in U$ and $b \in W$ at this stage would amount to adding a cycle into the clustered graph, which is not permitted by APC.

4. Loop counts c_i, $i = 0, 1, \cdots, 6$, can be accordingly determined using the SDF repetitions vector (the vector of minimal repetition counts in a periodic schedule) for the application graph.

Given that each of the 4 pairs of actors can be grouped in the first-step, which, in turn, results in possibly two different schedules upon further grouping, we observe that there are at most 8 different single appearance looped schedules generated using this approach. Such different schedules can in general have different buffer memory requirements [11]. Thus, it can be useful for a designer to experiment with alternative schedules, estimate the buffer memory requirements for these schedules, and identify the schedule that best matches the application requirements and resource constraints.

For the `butterfly` pattern shown in Fig. 3.9(a), Table 3.4 shows 9 different schedules, including a flat schedule for comparison. It can be seen that each of these

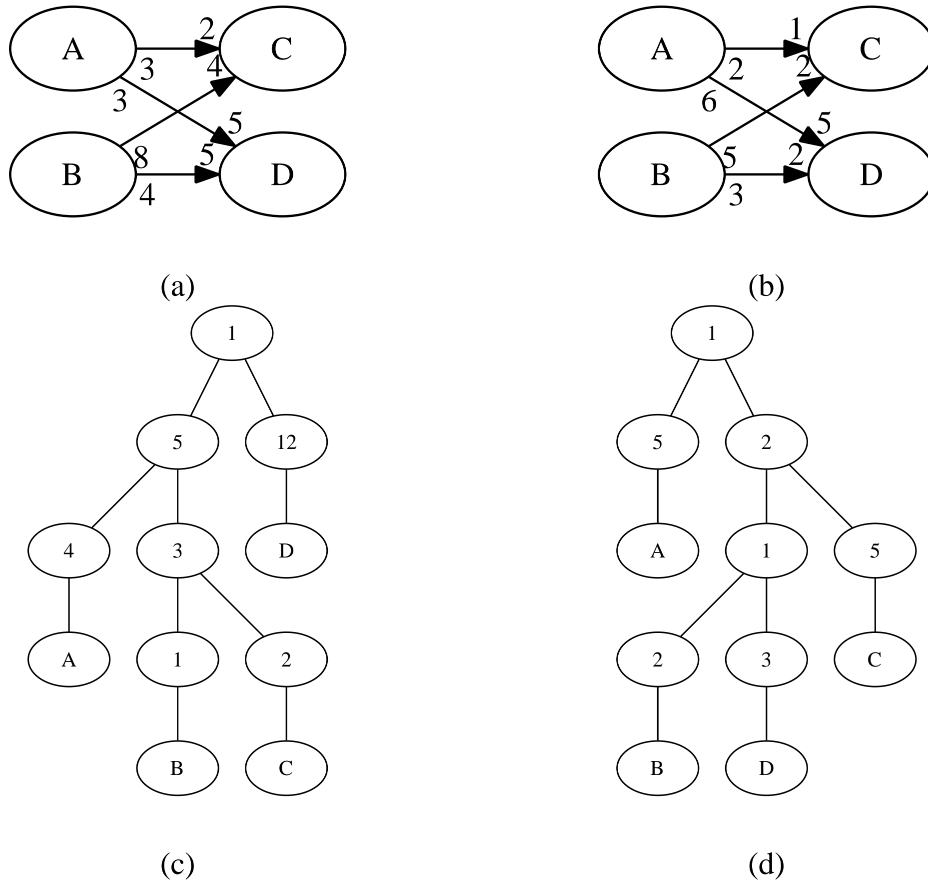


Figure 3.9: (a)-(b) SDF graphs with `butterfly` patterns. (c)-(d) GSTs for minimizing buffer memory requirements of the SDF graphs in (a) and (b), respectively.

schedules has different buffer memory requirements. In a given design context, a designer may want to experiment with all schedules that fit the available resources in the target platform. The optimal schedule from the viewpoint of total buffer memory cost (schedule (1)) has a total buffer memory cost of 140 memory units, and is generated using the APGAN strategy.

However, APGAN is in general a heuristic and is therefore not always guaranteed to derive an optimal solution. For example, consider the `butterfly` pattern shown in Fig. 3.9(b). Table 3.5 shows 6 different schedules for this graph, including, again, a flat

Table 3.4: Buffer memory requirements for single appearance schedules generated from the SDF graph shown in Fig. 3.9(a).

Schedule	Single Appearance Schedule	Total buffer requirement (number of tokens)
Flat	(20 A)(15 B)(30 C)(12 D)	300
1	(5 (4 A)(3 B(2 C)))(12 D)	140
2	(20 A)(3 (5 B(2 C))(4 D))	148
3	(5 (3 B)(2 (2 A)(3 C)))(12 D)	150
4	(15 B)(2 (5 (2 A)(3 C))(6 D))	216
5	(15 B)(4 (5 A)(3 D))(30 C)	255
6	(15 B)(2 (2 (5 A)(3 D))(15 C))	225
7	(20 A)(3 (5 B)(4 D))(30 C)	260
8	(20 A) (3 (5 B)(4 D)(10 C))	180

Table 3.5: Buffer memory requirements for single appearance schedules generated from the SDF graph shown in Fig. 3.9(b).

Schedule	Single Appearance Schedule	Total buffer requirement (number of tokens)
Flat	(5 A)(4 B)(10 C)(6 D)	72
1	(4 B)(5 A(2 C))(6 D)	64
2	(5 A)(2 (2 B)(5 C)(3 D))	56
3	(5 A)(2 (2 B)(5 C))(6 D)	62
4	(5 A)(2 (2 B)(3 D))(10 C)	66
5	(5 A)(2 (2 B)(3 D)(5 C))	56

schedule, and 5 different looped schedules. Here, schedule (1) is the one generated by applying the APGAN strategy, and it can be seen that schedules (2), (3), and (5) outperform this schedule in terms of total buffer memory requirements.

This example demonstrates the utility of experimenting with alternative schedules even if established heuristics, such as APGAN, are available. Topological patterns facilitate such experimentation through their capabilities for schedule representation. In particular, topological patterns allow designers to construct structured patterns of schedules,

which can then be examined separately to determine which one is most suitable in a given design context. Furthermore, topological pattern representations can be used to maintain libraries of subsystem-specific schedules, which can then be drawn upon efficiently when constructing larger applications that employ those subsystems.

3.6 Summary

We have introduced the concept of topological patterns, which can be used to identify and concisely iterate across arbitrary structures in a dataflow application graph. We have shown how the types of flowgraph substructures that are pervasive in the DSP application domain can be effectively represented in terms of topological patterns, and thereby used to generate compact, scalable application representations.

We have also shown how an underlying design tool can exploit a high-level application specification consisting of topological patterns in various aspects of the design flow. In particular, we have demonstrated the efficacy of topological patterns in dataflow graph analysis, concise and scalable representation of HSDF graphs, and exploring implementation-specific trade-offs. We have also shown the use of topological patterns in graph analysis and extracting implementation-specific features. We have applied the concept of topological patterns to represent schedules for application graphs. Such representations are useful, for example, when porting schedules generated using one design tool to other platform-specific tools or design languages. We have demonstrated the utility of experimentation with pattern-specific scheduling transformations, and how topological patterns facilitate such experimentation.

Chapter 4

Prototyping Heterogeneous Dataflow Applications using Core Functional Dataflow

We have provided a brief summary of dataflow models relevant to the work presented in this thesis in Chapter 2. These dataflow models with varying degrees of expressive power can model dataflow behaviors that range from being completely static to highly dynamic, where production and consumption volumes can vary on a per-invocation basis. In Section 1.3.2, we introduced the CFDF model, which we have formulated and developed as part of this thesis. The CFDF model can be used to model a wide variety of deterministic dynamic dataflow behaviors [64]. At the same time, it supports flexible and efficient prototyping of dataflow-based application representations and permits natural description of both dynamic and static dataflow actors. In this chapter, we present the semantics of CFDF. We demonstrate how various heterogeneous dataflow applications can be modeled using CFDF. We show how various existing dataflow models can be represented using CFDF semantics. We also present application of CFDF to rapid prototyping of heterogeneous dataflow applications.

4.1 Related Work

A number of development environments utilize dataflow models to aid in the capture and optimization of functional application descriptions. Ptolemy II encompasses a

diversity of dataflow-oriented and other kinds of MoCs [25]. To describe an application subsystem, developers employ a *director* that controls the communication and execution schedule of an associated application graph. If an application developer is able to write the functionality of an actor in a prescribed manner, it will be polymorphic with respect to other MoCs. To describe an application with multiple MoCs, developers can insert a "composite actor" that represents a subgraph operating with a different MoC (and therefore its own director). In such hierarchical representations, directors manage the actors only at their associated levels, and directors of composite actors only invoke their actors when higher level directors execute the composite actors. This paradigm works well for developers who know a priori the modeling techniques with which they plan to represent their applications.

One of the other techniques employs SystemC to capture actors as composed of input ports, output ports, functionality, and an execution finite state machine (FSM), which determines the communication behavior of the actor [32]. There exist languages, such as CAL [26], that specifically target actor descriptions. For complete functionality in Simulink [51], actors are described in the form of "S-functions." By describing them in a specific format, actors can be used in continuous, discrete-time, and hybrid systems. LabVIEW [39] even gives designers a way of programmatically describing graphical blocks for dataflow systems.

Semantically, perhaps the most related work is that of a MoC called the SBF [41]. In SBF, an actor is represented by a set of functions, a controller, state, and transition function. Each function is sequentially enabled by the controller, and uses on each invocation a blocking read for each input to consume a single token. Once a function is done

executing, the transition function defines the next function in the set to be enabled.

CFDF semantics, and features in the DIF tool based on those, differ from these related efforts in dataflow based design in their integrated emphasis on minimally-restricted specification of actor functionality, and support for efficient static, quasi-static, and dynamic scheduling techniques. Each may be critical to prototyping overall dataflow graph functionality. Compared to models such as SBF, CFDF allows a designer to describe actor functionality in an arbitrary set of fixed modes, instead of parceling out actor behavior as side-effect free functions, a controller, and a transition function. CFDF is also more general than SBF as it permits multi-token reads and can enable actors based on application state. As designers experiment with different dataflow representations with different levels of actor dynamics, they need corresponding capabilities to experiment with compatible scheduling techniques. This is a key motivation for the integrated actor- and scheduler-level prototyping considerations in CFDF and its support in DIF. The material presented in this chapter is based on the work in [64], [66], and [65].

4.2 Formulation of Core Functional Dataflow

CFDF semantics can be viewed as a "deterministic dataflow subset" of *enable-invoke dataflow* (EIDF) semantics, which require that actor specification be divided into separate *enable* and *invoke* functions [64] (described below). A CFDF actor a also has a set of valid modes M_a in which it can execute. When the actor a executes in a mode $m \in M_a$, it consumes (produces) a fixed number of tokens from its inputs (onto its outputs), but the number of tokens consumed and produced by an actor can vary across different modes

in M_a. The separation of *enable* and *invoke* capabilities helps in prototyping efficient scheduling techniques.

The *enable* function is designed to be used as a "hook" for dynamic or quasi-static scheduling techniques to rapidly query actors at run-time, and check whether or not they are executable. The *enable* function only checks for the availability of sufficient input data to allow an actor to fire in its current mode, and does not consume any tokens from the actor inputs. The current mode of an actor is always unique in CFDF, so this check of "data sufficiency" is unambiguous. Given an actor $a \in V$ in a dataflow graph $G(V, E)$, the *enabling function* for a is defined as:

$$\varepsilon_a : (T_a \times M_a) \rightarrow B, \tag{4.1}$$

where $T_a = \aleph^{|\text{in}(a)|}$ is a tuple of the number of tokens on each of the input edges to actor a (here, $|\text{in}(a)|$ is the number of input edges to actor a); M_a is the set of modes associated with actor a; and $B = \{true, false\}$ is $true$ when an actor $a \in V$ has an appropriate number of tokens for mode $m \in M_a$ available on each input edge, and $false$ otherwise. An actor can be executed in a given mode at a given point in time if and only if the enabling function is true-valued.

The *invoke* function, on the other hand, consumes as many tokens from the inputs as specified by its mode of execution, and correspondingly produces the specified numbers of tokens onto the actor outputs. The *invoke* function can generally change the mode of the actor by returning a valid mode of execution in which the actor should be fired during its next invocation. Thus, actors proceed deterministically to a unique "next mode" of

execution whenever they are enabled. The *invoking function* for an actor a is defined as:

$$\kappa_a : (I_a \times M_a) \rightarrow (O_a \times M_a), \tag{4.2}$$

where $I_a = X_1 \times X_2 \times \cdots \times X_{|\mathrm{in}(a)|}$ is the set of all possible inputs to a, where X_i is the set of possible tokens on the edge on input port i of actor a. After a executes, it produces outputs $O_a = Y_1 \times Y_2 \times \cdots \times Y_{|\mathrm{out}(a)|}$, where Y_i is the set of possible tokens on the edge connected to port i of actor a, and $|\mathrm{out}(a)|$ is the number of output ports. If no mode is returned (i.e., an empty mode set is returned), the actor is forever disabled.

We further illustrate these CFDF semantics by applying those to model the dynamic dataflow behavior of actors in applications such as the AMS in Section 4.3.

For use of EIDF in modeling applications that cannot be modeled using CFDF, such as the Gustav function [5], we refer readers to the discussion in [65].

4.3 Modeling Adaptive Modulation Scheme using Core Functional Dataflow

As mentioned in Section 3.5.5, the AMS is an example of a useful, restricted class of dataflow-based applications or subsystem modules, the graphical representation of which can be reduced to the form shown in Fig. 3.6(a). In Fig. 3.6(a), S_1, S_2, and S_3 denote regions consisting of SDF actors that can be SDF-clustered, while actors D_1 and D_2 have dynamic behavior. A group of actors can be said to be *SDF-clustered* if its component actors can be scheduled together (i.e., the group can be scheduled as a single unit in the overall schedule for the graph) without introducing deadlock [11]. The dataflow edges $e_1, e_2, \ldots, e_5$ denote FIFO buffers. The D on edge e_5 denotes the delay, del(e_5), associated with it. In this targeted class of applications, it is assumed that the production and

consumption rates k, x, y, z, q are positive integer constants, while the consumption rate c_i and production rate p_i can vary over finite ranges of positive integer values with known upper bounds $c_{\max}$ and $p_{\max}$, respectively. The subscript i in the symbols p_i and c_i represents the dependence of this production and consumption rate pair on the actor execution index i — thus, p_i represents the number of tokens produced onto e_4 in the ith execution (*firing*) of D_2, and c_i represents the number of tokens consumed from e_1 during the ith firing of D_1. Such a class of applications is of particular importance since useful compile-time analysis can be performed while handling the dynamic behavior as explained later in Section 5.3.

Fig. 3.6(b) shows a simplified representation of the AMS with a source of input `bits`, dynamic `mapper`, `channel`, dynamic `demapper`, feedback path (e_5), and output `sink` actor. The `mapper` maps the bit(s) from the input bitstream to a symbol for transmission over the `channel`, while the `demapper` outputs one or more bits for each of the symbols received from the `channel`. The number of bits per symbol depends upon the modulation and demodulation schemes (e.g., QPSK or 64QAM) used by the `mapper` and `demapper`, respectively. The `mapper` receives feedback from the `demapper` indicating the result of channel estimation and accordingly selects one of the modulation schemes to be employed, which in turn determines the number of bits per symbol. Hence, the number of tokens consumed (produced) by the `mapper` (`demapper`) from the buffer e_1 (e_4) in general can vary from one invocation to the next.

We use CFDF to model dataflow behavior of the actors in the AMS application, and in particular, the dynamic `mapper` and `demapper` actors. The remaining actors are SDF actors and can be modeled easily as CFDF actors with just one valid mode

Table 4.1: Valid modes for the `mapper` actor along with their corresponding production and consumption rates.

Mode	Consumption rate		Production rate
	e_1	e_5	e_2
control	0	1	0
QPSK	2	0	1
16QAM	4	0	1
64QAM	6	0	1

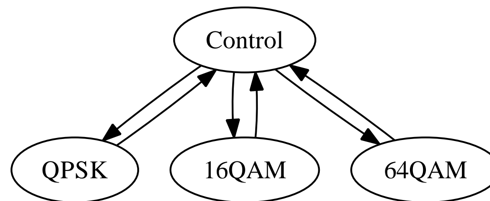


Figure 4.1: Mode transition behavior of the `mapper` actor.

each. Table 4.1 shows the possible modes for a generic dynamic `mapper` actor with their respective production and consumptions rates. It has a mode corresponding to each of the possible modulation schemes being employed (here QPSK, 16QAM, and 64QAM), and an additional mode called `control`. In the `control` mode, the mapper actor reads a channel quality indicator token from the feedback edge e_5 (see Fig. 3.6). This information is then used to determine the modulation scheme to be employed, and the *invoke* function returns (as the next mode value) the mode that corresponds to this scheme. The `demapper` actor can be modeled in a similar manner. Fig. 4.1 shows the mode transition behavior for the `mapper` actor.

4.4 Translation to Core Functional Dataflow

Many of the commonly used dataflow models can be directly translated to CFDF in an efficient and intuitive manner. In this section we show such constructions that demonstrate the expressibility of CFDF, and how the existing designs can be readily represented using CFDF semantics.

4.4.1 Static Dataflow

SDF, CSDF, and other static dataflow-actor behaviors can be translated into finite sequences of CFDF modes for equivalent operation. Consider, for example, CSDF, in which the production and consumption behavior of each actor a is divided into a finite sequence of periodic phases $P = (1, 2, ..., n_a)$. Each phase has a particular production and consumption behavior. The pattern of production and consumption across phases can be captured by a function ϕ_a whose domain is P_a. Given a phase $i \in P_a$, $\phi_a(i) = (G_i, H_i)$, where G_i and H_i are vectors indexed by the input and output ports of a, respectively, that give the numbers of tokens produced and consumed on these edges for each port during the ith phase in the execution of actor a.

To construct a CFDF actor from such a model, a mode is created for each phase, and we denote the set of all modes created in this way by M_a. Given a mode $m \in M_a$ corresponding to phase $p \in P_a$, the enable method for this mode checks the input edges of the actor for sufficient numbers of tokens based on what the phase requires in terms of the associated CSDF semantics. Thus, for each input port z of a, mode m checks for the availability of at least $G_p(z)$ tokens on that port, where $\phi(p) = (G_p, H_p)$. For

Table 4.2: Valid modes in a CFDF representation of the decimator actor, M, in Fig. 2.2(a) along with their corresponding production and consumption values.

Mode	Consumption rate in	Production rate out
mode_0	1	1
mode_1	1	0
mode_2	1	0
mode_3	1	0

the complementary invoke method, the consumption of input ports is fixed to G_p, the production of output ports is fixed to H_p. The next mode returned by the invoke method must be the mode corresponding to the next phase in the CSDF phase sequence.

For example, consider the decimator actor, M, in Fig. 2.2(a), which has a decimation factor of 4. Its dataflow behavior can be modeled using the CSDF model, as explained in Section 2.1.2. We can construct an equivalent CFDF representation of this actor using the process explained above. Table 4.2 shows modes of the decimator actor with their corresponding consumption and production values. In this table, `in` and `out` refer to the input and output ports, respectively, of the decimation actor. Fig. 4.2 shows the mode transition behavior of the decimator actor.

Since any SDF actor can be viewed as a single-phase CSDF actor, the CFDF construction process for SDF is a specialization of the CSDF-to-CFDF construction process described above in which there is only one mode created.

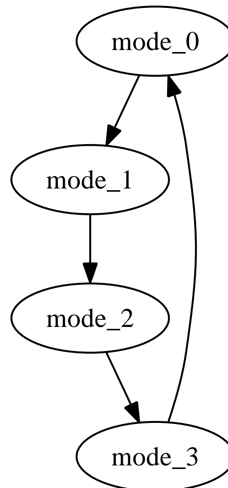


Figure 4.2: Mode transition behavior of the decimator actor. See Table 4.2 for the dataflow behavior of the actor in each mode.

4.4.2 Boolean Dataflow

BDF adds dynamic behavior to dataflow [16]. The two fundamental elements of BDF are `Switch` and `Select`. `Switch` routes a token from its input to one of two outputs based on the Boolean value of a token on its control input. The concept of a control input is also utilized for `Select`, in which the value of the control token determines which input port will have a token read and forwarded to its one output.

Consider an application shown in Fig. 4.3 that uses a `Switch` actor. To construct a CFDF actor that implements BDF semantics, we create a mode that is dedicated to reading that input value, which we call the `Control` mode. The result of this examination sends the actor into either a `True` mode or a `False` mode that corresponds to that control input. In the case of `Switch`, this implies three modes with behavior described in Table 4.3. The mode transition behavior of the `Switch` actor is shown in Fig. 4.4. For a strict construction of BDF, only the `Switch` and `Select` actors are needed for imple-

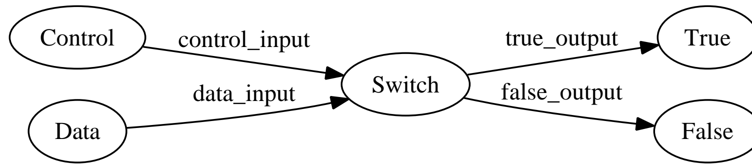


Figure 4.3: Application of BDF using a `Switch` actor.

Table 4.3: The behavior of the `Switch` actor modes in terms of tokens produced and consumed.

mode	consumption rate		production rate	
	control_input	data_input	true_output	false_output
Control	1	0	0	0
True	0	1	1	0
False	0	1	0	1

mentation, but CFDF does permit more flexibility, allowing designers to specify arbitrary behavior of `True` and `False` modes as long as each mode has a fixed production and consumption behavior.

4.4.3 Representing PSDF and PCSDF Actors using CFDF

It is also possible to construct an equivalent CFDF representation of a PSDF or PCSDF actor in ways similar to those described in Sections 4.4.1 and 4.4.2. For constructing a CFDF representation, a PSDF or PCSDF actor can be considered as a combination of static and dynamic dataflow behaviors. A PSDF or PCSDF actor mode corresponding to parameter configuration can be viewed as a control-oriented mode, while other modes model different static behaviors for specific settings of actor parameters. We illustrate this using an example of a TDD.

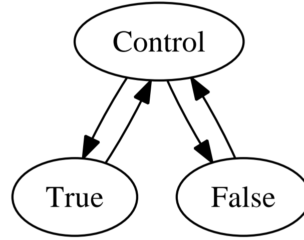


Figure 4.4: Mode transition behavior of the `Switch` actor. See Table 4.3 for dataflow behavior of the actor in each mode.

Consider the application graph shown in Fig. 2.3(a) along with its PCSDF description in Fig. 2.3(b). In particular, consider the `Decimator` actor. Suppose the decimation factor of this actor can be set to either 3 or 4 depending upon a parameter set from its control port. Fig. 4.5 shows the mode transition behavior of the `Decimator` actor in one of its possible CFDF representations. Here, modes `mode_4_0`, `mode_4_1`, `mode_4_2`, and `mode_4_3` together represent a behavior similar to a CSDF `Decimator` actor with a decimation factor of 4, while `mode_3_0`, `mode_3_1`, and `mode_3_2` together represent a behavior similar to a CSDF `Decimator` actor with a decimation factor of 3. The control mode models the parameter configuration. Note that actor does not consume or produce any dataflow tokens in this mode. Also, note the transitions out of modes `mode_4_3` and `mode_3_2`. It is possible that parameters associated with the `Decimator` are configured after every CSDF cycle (for example, if configured by a subinit graph) or multiple CSDF cycles (for example, if configured by an init graph). The scheduler can exploit this behavior depending upon the application under consideration to construct efficient PLSs.

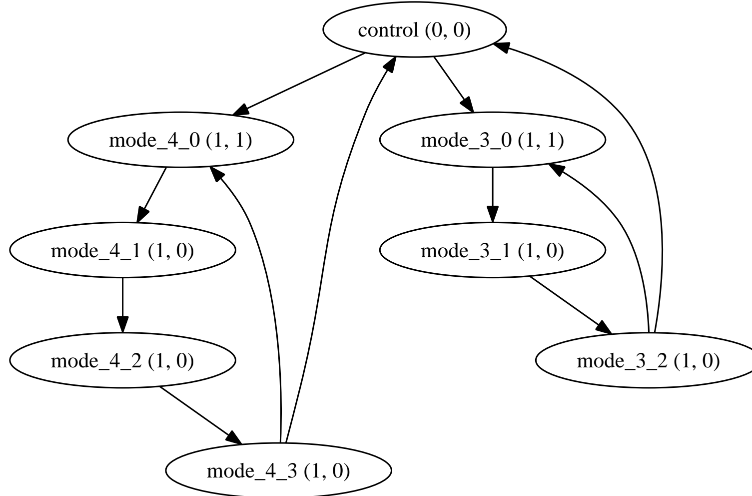


Figure 4.5: CFDF mode transition behavior of a PCSDF actor `Decimator` with possible decimation factors 3 and 4. The annotation (i, j) for a CFDF mode indicates the corresponding numbers of tokens consumed from the input (i), and produced onto the output (j) by the actor.

4.5 Functional Simulations in DIF using Core Functional Dataflow

The CFDF semantics allow the DIF package to support functional simulations. The segregation of enable and invoke functions allows use of a guarded execution of an actor. The term guarded execution refers to a scheme of firing an actor in which the actor is fired (i.e. its invoke function is called) only if its enable function asserts the availability of sufficient data to fire the actor in its current mode by returning *true*. This feature allows using a *canonical schedule* (described below in Section 4.5.2) in which every actor is fired once in every schedule iteration. Such simulations can also be used for analyzing and estimating the buffer requirements in terms of number of tokens accumulated in those buffers. This provides an estimate of total memory requirements as well as individual buffers when porting an application to the target platform. It is also possible to use more

sophisticated scheduling techniques such as those we will describe in Chapter 5.

4.5.1 Extensions to DIF Software Architecture

The DIF package has been restructured to support functional simulations. We have introduced a library of actors, which adhere to CFDF semantics, described in Java programming language for use in the applications. Actors are objects derived from a base class that provides each actor with mode and edge interfaces along with base methods for the enabling and invoking functions, called the enable method and the invoke method respectively. Modes can be created either by a user through an API or automatically, based on other information about the application (e.g., the sequence of phases in a CSDF representation). While a designer will redefine an actor's class methods to define the proper functionality, the enable method is always restricted to only checking the number of tokens on each input (as per the enabling function definition). The invoke method may read values from inputs, but it must consume them as tokens. In other words, when a mode is invoked on an actor, the actor consumes a fixed number of tokens that is associated with that mode, and no more values are read. In either case, we expect designers to effectively construct a case statement of all of the possible modes for a given actor, and fill in the functionality of each mode in a case.

A scheduler uses GST representation to represent the generated schedule, which is then used to simulate the application graph.

4.5.2 Canonical Scheduler

We can always construct a canonical schedule for an application graph. This is the most trivial schedule that can be constructed from the application graph. The canonical schedule is a single appearance schedule (a schedule in which actors of the application graph appear once) which includes all actors in some order. In terms of the GST representation, a canonical schedule has a root node specifying the loop count of 1 with its child nodes forming leaves of the schedule tree. Each leaf node points to a unique actor in the application graph. The ordering of leaf nodes determines the order in which actors of the application graph are traversed. When the simulator traverses GST, each actor in the graph is fired, if it is enabled.

4.6 Design Examples

Polynomial evaluation is a commonly used primitive in various domains of signal processing, such as wireless communications and cryptography. Polynomial functions may change whenever senders transmit data to receivers. The kernel is the evaluation of a polynomial $F_i(x) = \sum_{k=0}^{n_i} c_k \times x^k$, where $c_1, c_2, \ldots, c_n$ are coefficients, x is the polynomial argument, and n_i is the degree of the polynomial. Since the coefficients may change at runtime, a programmable *polynomial evaluation accelerator* (*PEA*) is useful for accelerating the computation of multiple F_i's.

Table 4.4: The behavior of the PEA modes.

mode	consumes		produces	
	Control	Data	Result	Status
Normal	1	0	0	0
Reset	0	0	0	0
Store Poly	0	1	0	1
Evaluate Poly	0	1	1	1
Evaluate Block	0	1	1	1

4.6.1 Programmable Polynomial Evaluation Accelerator

Since the degree and coefficients of a polynomial may change at run-time (e.g., for different communications standards or different subsystem functions), a programmable PEA is useful for accelerating the computation of multiple F_i 's in a flexible way. To this end, we design a PEA with the following instructions: *reset*, *store polynomial* (STP), *evaluate polynomial*, and *evaluate block* (EVB). *Evaluate polynomial* is for a single evaluation, and EVB is for a bulk evaluation of the same polynomial.

Since data consumption and production behavior for the PEA depends on the specific instruction, a PEA actor cannot follow the semantics of conventional dataflow models, such as SDF. However, if we define multiple modes of operation, we can capture the required dynamic behavior as a collection of CFDF modes. Following this principle, we have implemented the PEA as a single CFDF actor. In our functional description of the actor, we defined different modes according to the four PEA instructions. These modes are summarized in Table 4.4.

The normal mode (like the "decode" stage in a typical processor) reads an instruction and determines the next operating mode of the data path. Of particular note here

is the behavior of STP in which the number of coefficients read varies. Each individual mode is restricted to one particular consumption rate, so when the STP mode is invoked, it reads a single coefficient, stores it, and updates an internal counter. If the counter is less than the total number of coefficients to be stored, invoke returns STP as the next mode, so it will continue reading until done. Note that persistent internal variables ("actor state variables"), such as a counter, can be represented in dataflow as self-loop edges (edges whose source and sink actors are identical), and thus, the use of internal variables does not violate the pure dataflow semantics of the enclosing DIF environment.

We find that functional simulations using the high-level DIF prototype of a PEA application based on CFDF model are faster by a factor of 4.9 compared to those using an implementation in lower level language, such as Verilog [64].

4.6.2 Design with Multiple Polynomial Evaluation Accelerators

To illustrate the problem of heterogeneous complexity, we suppose that a DSP application designer might use two PEA actors customized for different length polynomials. For this application, we restrict the PEA's functionality to be a CSDF actor with two phases: reading the polynomial coefficients and then processing a block of x's to be evaluated, as shown in Table 4.5. The overall PEA system is shown in Fig. 4.6. Two PEA actors are in the same application and we made them selectable by bracketing them with a `Switch` and a `Select` block. To manage these two PEA actors properly, this design requires control to select the `PEA1` or `PEA2` branch. In this system, the CSDF PEA actors consume a different number of polynomial coefficient tokens, so the control tokens

Table 4.5: The behavior of the CSDF implementation of the restricted PEA used in the dual PEA application.

Actor	mode	consumes Data	produces Result
PEA1	Store Poly	4	0
	Evaluate Block	15	15
PEA2	Store Poly	7	0
	Evaluate Block	15	15

driving the `Switch` and `Select` on the data path must be able to create batches of 19 and 22 tokens, respectively for each path. If the designer is restricted to only `Switch` and `Select` for BDF functionality, the balloon with `CONTROLLER` shows how this can be done.

This design can certainly be captured with model oriented approach, pulling the proper actors into super-nodes with different models. But like many designs, this application has a natural functional hierarchy in it with the refinement of `CONTROLLER`, and with `PEA1` and `PEA2`. We believe that competing design concerns of functional and model hierarchy will ultimately be distracting for a designer. With this work, we focus designers on efficient application representation and not model related issues.

Immediate simulation of the dual PEA application is possible to verify correctness by using the canonical schedule. We simulated the application with a random control source and a stream of integer data. A nontrivial schedule tree can significantly improve upon the canonical performance. Given that the probability of a given PEA branch being selected is uniform, we can derive a single appearance schedule shown in Figure 4.7, where each leaf node is annotated with an actor and each interior node is annotated with

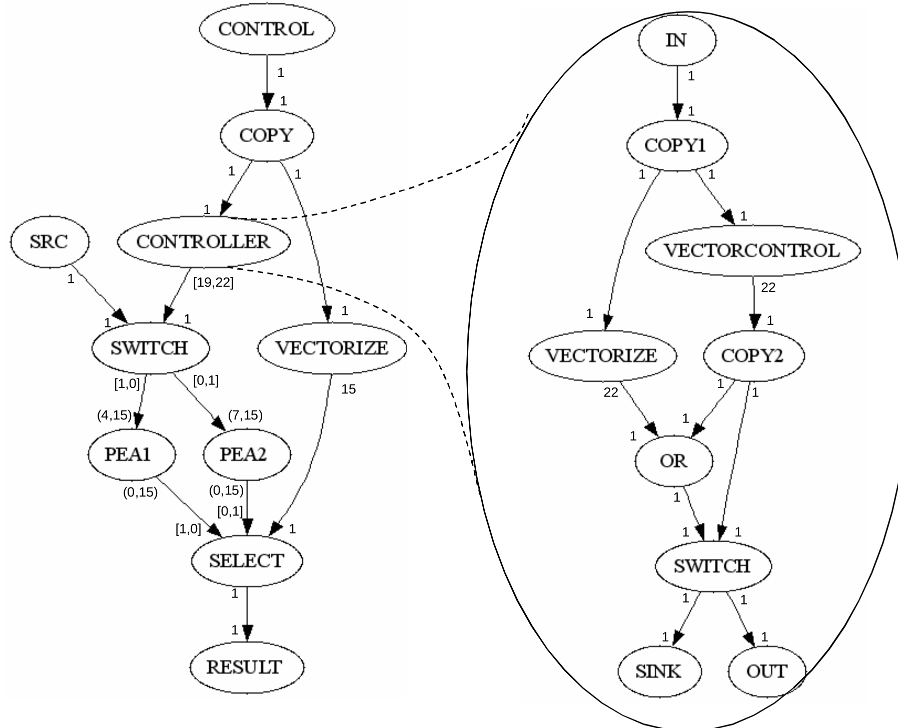


Figure 4.6: A pictorial representation of the dual PEA application.

a loop count. Figure 4.8 shows a manually designed multiple appearance schedule (a schedule in which actors may appear more than once) that attempts to process polynomial coefficients first, before queuing up data to be evaluated, to reduce buffering. Note that the `SRC` and `CONTROL` actor are unguarded as they require no input tokens to successfully fire.

4.6.3 Results

We also implemented an polyphase uniform discrete Fourier transform (DFT) filter bank and a sample rate conversion application. We constructed the decimated uniform DFT filter bank using a mixed-model consisting of CSDF and SDF actors [54]. The sam-

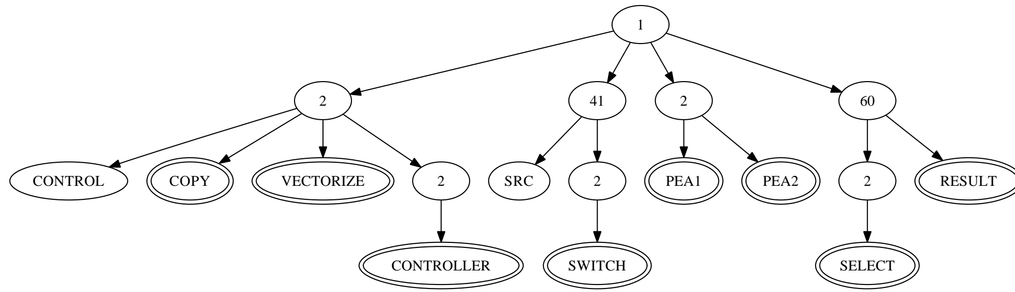


Figure 4.7: Single appearance schedule for the dual PEA system.

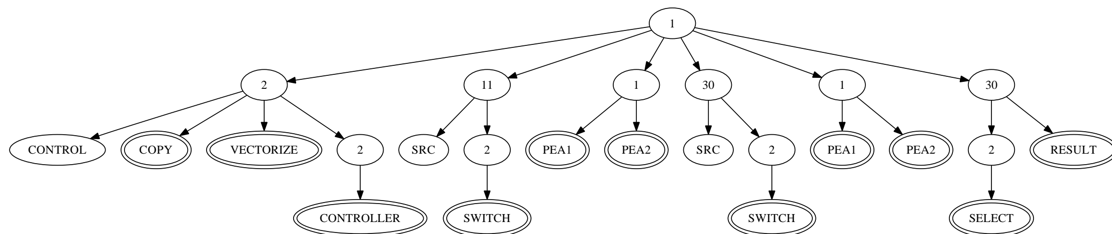


Figure 4.8: Multiple appearance schedule for the dual PEA system.

ple rate conversion application is based on concepts found in [21] and [36]. Results for these different implementations with different schedules are summarized in Table 4.6. We simulated 10000 evaluations running on a 1.7GHz Pentium with 1GB of physical memory. We measured the time it took to complete enough iterations to complete all of the evaluations and maximum total queue size. The manually designed schedules performed notably better than the canonical schedule. Such insight can be invaluable when considering the final implementation of the controller logic.

4.7 Summary

We have formulated the CFDF model, which can be used to model a wide variety of deterministic dynamic dataflow behaviors, and used to capture various well known

Table 4.6: Simulation times and maximum buffer sizes for mixed-model applications.

Application	Schedule	Simulation Time (s)	Maximum observed buffer size (tokens)
Dual PEA - BDF Strict	Canonical	6.88	2,327,733
	Single appearance	1.72	1,729
	Multiple appearance	1.59	1,722
Dual PEA - CFDF	Canonical	3.57	1,018,047
	Single appearance	0.95	1,791
	Multiple appearance	0.99	1,800
DFT Filter	Canonical	0.91	17
	Single appearance	1.02	24
Sample Rate Converter	Canonical	9.15	9,394
	Single appearance	1.43	2,408

forms of dataflow in a single, unified formulation. We have also presented the features of CFDF model and tools based on it, such as support for heterogeneous dataflow behaviors, intuitive and common framework for functional specification, support for functional simulations that allows designers to model and verify interactions between those models, portability from most of the existing dataflow models to CFDF, and integrated emphasis on minimally-restricted specification of actor functionality. With this CFDF modeling approach integrated into DIF, we have demonstrated its use with various applications. Such an approach has allowed us to functionally simulate the design immediately, and then focus on experimenting on schedules and dataflow styles to improve performance.

Chapter 5

Efficient Scheduling Techniques for Core Functional Dataflow Graphs

In Chapter 4, we described various features of the CFDF model of computation, and applied this model to specify and prototype different heterogeneous dataflow applications. For functional simulations of CFDF application graphs, we used the CFDF canonical scheduler, as explained in Section 4.5.2. It is, however, possible to use more efficient scheduling techniques that are applicable to general or specialized classes of CFDF graphs. In this chapter, we focus on scheduling techniques for CFDF graphs, and present three different scheduling techniques that employ, respectively, (1) decomposition of dynamic dataflow graphs, (2) mode grouping, and (3) parameterized looped schedules.

5.1 Scheduling using Dynamic Dataflow Graph Decomposition

We proposed this technique as a generalized scheduling strategy in [62]. It is based on decomposing a dynamic dataflow graph into a set of static interacting graphs. It makes use of the fact that every CFDF mode has a fixed production and consumption behavior. To construct a static graph based on these modes, it finds the combination of modes in which one mode from each actor in the subgraph is producing or consuming on an edge that has a consuming or producing mode at the other end of the edge. Since every actor can potentially provide many modes, there are an exponential number of combinations to be considered. To avoid exploring this entire space, a reachability analysis is performed

to consider only those modes that are connected to each other. For this, an extension of depth first search (DFS) graph traversal with the concept of mode traversal to arrive at the set of static subgraphs is employed as shown in Fig. 5.1.

We have mentioned the relevant research efforts related to modeling and simulating heterogeneous dataflow applications, and highlighted novelty of CFDF approach in Section 4.1. In the context of efficient scheduling and simulations of applications, our generalized scheduling framework differs from these related efforts in dataflow-based design in that our framework uses top-down analysis of (explicitly-specified) application structure combined with integration of static dataflow sub-behaviors (actor modes) across groups of dataflow actors. This approach to analysis and integration systematically extends the reach of static scheduling techniques so that they can be used across significant portions of dynamic dataflow designs. The approach is driven by the modeling semantics of CFDF, which provides the explicit decomposition of actors into static dataflow sub-behaviors, and efficiently exposes to the scheduler the design spaces associated with separating sub-behaviors of individual actors, and grouping subsets of sub-behaviors across different actors.

5.1.1 Dynamic Dataflow Graph Decomposition Algorithm

The key addition to the traditional DFS is that the next nodes to be added to the working stack S are found by following a mode from the current node. Another stack of nodes T keeps track of what order the nodes have been visited, so that the graph visited state may be unwound. When a static subgraph has been completed or an invalid

```
Function DecomposeCFDFGraph
  Data: CFDF Graph G
  Result: Returns set of static graphs
  Graphs Gs ← {} ;
  foreach source mode in G do
      /* we use stacks for both the DFS and ensuring all modes are
         visited                                                        */
      Stack S ← {} ;
      Stack T ← {} ;
      SDFGraph sdfG ← empty graph;
      T.push(node that contains the source mode) ;
      while T has elements do
          S.push(T.pop()) ;
          while S has elements do
              Actor A ← S.pop() ;
              if A not visited then
                  mark A as visited ;
                  foreach mode M in A do
                      if M not visited matches the connecting edge then
                          S.push(actors on inputs and outputs of M) ;
                          sdfG.add(A) ;
                          sdfG.annInEdges(M.cons) ;
                          sdfG.annOutEdges(M.prod) ;
                      end
                      mark M as visited ;
                  end
                  T.push(A)
              end
          end
          /* when the stack is empty, one static graph is complete    */
          if sdfG is a valid graph then
              if Gs.doesNotContain(sdfG) then
                  Gs.add(sdfG) ;
              end
          end
          /* in every case, unwind graph                              */
          while T has elements do
              if T.peek().allModesVisited() then
                  Actor B ← T.pop() ;
                  B.resetNodeVisitedFlag() ;
                  B.resetAllModeVisitedFlags() ;
              else
                  T.peek().resetNodeVisitedFlag() ;
                  break ;
              end
          end
      end
  end
  return Gs ;
```

Figure 5.1: Algorithm for dynamic dataflow graph decomposition.

graph has been found in the course of DFS, nodes are popped off of T until a node is found that has another mode to be considered (i.e. the potential of another unique static subgraph). Each of the popped nodes have their mode and node visited flags cleared, thus unwinding the graph state by making them available for the mode at the top of T. Therefore multiple graphs maybe constructed from the same source mode. Currently, we only consider directed acyclic graphs, so DFS is started at the source modes in the application (i.e., those that do not need input tokens to execute). Note that mode transition edges are not considered as edges to be traversed in DFS, effectively separating the graph at mode boundaries.

For example, consider the decomposition of the `Switch` application in Fig. 5.2(a) as shown in Fig. 5.2(b). Two source modes were found in `A` and `B`. The DFS from the mode of `A` ended immediately in the `control` mode of `Switch`, but the DFS from `B` found two matching modes in `Switch` (namely `true` and `false`). After a full run of DFS from `B`, the graph visited state unwinds back to `Switch` and DFS restarts again from the other `Switch` mode. Thus, the single dynamic BDF application graph has been transformed into three static subgraphs. Note that for a complete iteration of the original application to finish, more than one of the subgraphs must be run to completion. Indeed, because mode transitions may be arbitrary, we have no *a priori* way in general of exactly balancing the execution of these three graphs, and we must rely on the dynamic GSTs for proper simulation.

All graphs in the set of graphs that are created by this algorithm must be subgraphs of the original graph. Edges of this subgraph are annotated with the corresponding production and consumption numbers described by the modes used in a given run of DFS.

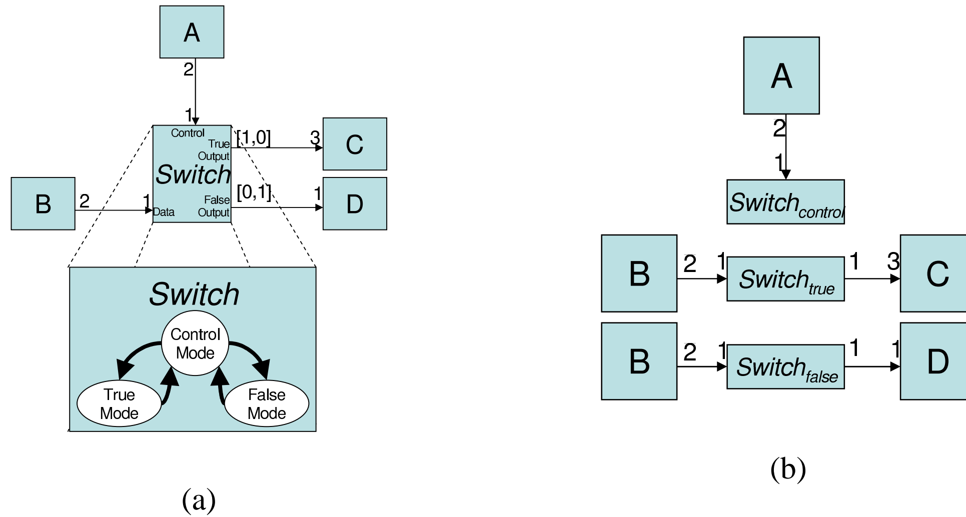


Figure 5.2: Application decomposition example.

Since the decomposition algorithm is based on DFS, the complexity of this algorithm is founded on it as well, but mode combinations make it exponential in the number of modes. Fortunately, in practice, this approach is efficient, since modes tend to be connected together in a structured way.

5.1.2 Simulation Results

To demonstrate this approach, we chose representative mixed-model applications to experiment with: a CSDF data distribution of audio streams to be sample-rate-converted, a polyphase decimated DFT filter bank, and an application with multiple polynomial evaluation accelerators.

Figure 5.3 shows a pictorial representation of the sample rate conversion application based on concepts found in [21] and [36]. Two audio channels are to be converted on two different subsystems. The input streams are interleaved, such as how multiple audio

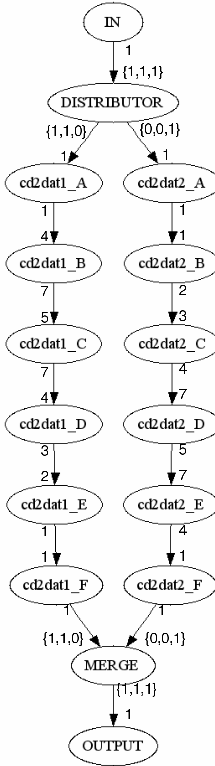


Figure 5.3: Dual sampling rate conversion.

channels might come over a single digital input. With a fixed interleaving, the CSDF `DISTRIBUTOR` actor distributes them to the appropriate multirate data path. In this case, a series of FIR filters is dedicated to sample rate conversion.

As in the case of Section 4.6.3, we also implemented an polyphase uniform DFT filter bank. We constructed a decimated uniform DFT filter bank using a mixed-model consisting of CSDF and SDF actors [54]. In addition, to show the dynamic capability of our approach, we used an application with PEAs, which utilizes both CSDF, SDF, and BDF elements. Polynomial functions may change when senders transmit data to receivers, so the application employs `Switch` and `Select` to dynamically change between the two data paths.

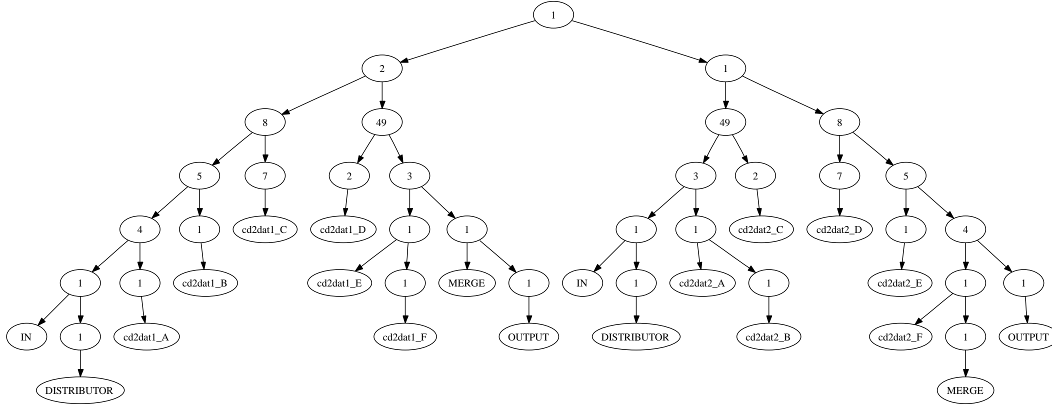


Figure 5.4: The APGAN schedule of the sample rate conversion application.

We applied our generalized scheduling approach to each of these applications and compared it to a CFDF *canonical schedule*. We compared this to the static subgraphs generated by our approach, which were scheduled with both a flat scheduler based on the repetition vectors of the SDF clusters and an APGAN-based scheduler [11]. The resulting GSTs were combined into a single GST by profiling the number successful executions, to balance the execution rates.

As an example, Fig. 5.4 shows the APGAN-generated schedule derived from our design flow on the sample rate conversion application. Two unique schedule trees resulted from the two subgraphs from the original application, and (based on the distributor element) one was executed twice as often as the other. Thus, these two trees were merged with iteration counts that balanced their execution (2 and 1, respectively).

Results for these different styles of implementation with different schedules are summarized in Table 5.1. We simulated thousands of tokens for each application on a 1.7GHz Pentium with 1GB of memory. The results show the utility of being able to apply the generalized scheduling approach presented in this work.

Table 5.1: Simulation times and maximum buffer sizes for mixed-model applications using dynamic dataflow graph decomposition based schedules.

Application	Schedule Strategy	Average Simulation Time (ms)	Maximum Observed buffer size (tokens)
Sample Rate Converter	Canonical	9,148	9,394
	Flat	1,425	2,408
	APGAN	1,462	2,278
Polyphase DFT Filter Bank	Canonical	910	17
	Flat	1,017	24
	APGAN	1,117	24
Multi-PEA	Canonical	2,163	11,198
	Flat	586	57
	APGAN	548	57

5.2 Scheduling using Mode Grouping

While static subgraphs can be successfully found by a generalized scheduling approach to dynamic applications in [62], some static behaviors are not considered. For example, in the decomposition of the switch application from Fig. 5.2 the `True` and `False` modes act predictably, always returning to `Control`, which is the mode that transitioned to them in the first place. The repeatable nature of these branches are the kind of static behavior that is exploitable. To this end, the actor description is augmented with the concept of *mode grouping*, in which application writers can refine their original application by grouping modes together [63]. For an actor a with modes M_a, we define a mode grouping, $D_a \subseteq M_a$, as a set of modes with a static relationship. The static mode behavior exposed in this work is a cyclic mode transitions in which all modes in the grouping return exactly one mode as the next mode, except for one mode, called the *entrance mode*. The entrance mode may have multiple transitions out, as it marks the single

point of dynamic behavior in the grouping, but after it is fired, the modes that follow it do so in a static sequence. The mode grouping can be considered by the scheduler as a single mode that has production and consumption behavior equal to the sum of the individual modes in it. The resulting schedule then includes a repeated firing the size of the mode grouping.

In the switch example, two mode groupings are

$$D_a = \{\{\text{Control}, \text{True}\}, \{\text{Control}, \text{False}\}\},$$

each with `Control` as the entrance mode. This exposes that `Control` always precedes a `True` or a `False`, allowing a larger schedule tree to be formed. For this small example performance benefits are slight, but for more complex applications, the assertion that a set of modes execute in a static sequence can lead to notably smaller buffer requirements.

5.2.1 Simulation Results

To evaluate the benefits of mode grouping, a set of both static and dynamic applications with actors that had mode groupings to exploit were considered. These application include B-spline interpolator, a CSDF data distribution of audio streams to be sample-rate-converted, a polyphase decimated DFT filter bank, and multiple PEAs. Apart from the applications that we have used for previously described experiments in this thesis, B-spline interpolator is a new one. We refer readers to [63] for detailed information regarding this application and how it can be modeled using CFDF model. For each application, the generalized scheduling strategy employing decomposition of dynamic dataflow graphs (using APGAN as the static scheduler) with and without mode groupings was em-

Table 5.2: Total buffer size requirements with and without mode grouping.

Application	Without Groups	With Groups	Percentage Improvement
BSpline Interpolation	479	304	37%
Sample Rate Converter	2,278	2,278	0%
Polyphase DFT Filter Bank	24	24	0%
Multi-PEA	3,802	2,976	22%

ployed. The resulting schedule trees were balanced and ordered based on the known input conditions, be it static patterns or probability distributions (see [63] for more details).

As seen from the simulation results in Table 5.2, the two purely static applications showed no benefit of using mode grouping. While mode groups were identified in CSDF actors, the original generalized scheduler performed equally well with and without groups. Once any dynamic behavior was inserted (i.e. the B-spline `Controller` and the PEA dynamic `Switch` and `Select` pair), mode grouping showed a significant improvement finding more (and larger) static schedule trees, which provided a direct savings in buffering by more optimal actor firings. Generalized scheduling with and without groups for each of these examples took less than 5 seconds on a modern CPU [63].

5.3 Parameterized Looped Schedules

The latest communication technologies invariably consist of modules with dynamic behavior. There exists a number of design tools for communication system design with their foundation in dataflow modeling semantics. These tools must not only support the functional specification of dynamic communication modules and subsystems but also provide accurate estimation of resource requirements for efficient simulation and implemen-

tation. We explore this trade-off — between flexible specification of dynamic behavior and accurate estimation of resource requirements — using a representative application employing an AMS. We propose an approach for precise modeling of such applications based on a recently-introduced form of dynamic dataflow called core functional dataflow. From our proposed modeling approach, we show how parameterized looped schedules can be generated and analyzed to simulate applications with low run-time overhead as well as guaranteed bounded memory execution. We have presented some of this work in [70] using the Advanced Design System from Agilent Technologies, Inc., which is a commercial tool for design and simulation of communication systems, for demonstration. In this thesis, We use DIF for demonstrating this technique.

There is generally a trade-off between the expressive power of the dataflow model being used and the compile-time (i.e., prior to execution or simulation) predictability that is available when analyzing specifications in that model. Although it is desirable to have as much expressive power as possible to best capture the dynamic nature of modern DSP and communication applications, this can lead to significant reductions in the ability to predict hardware and software resource requirements when targeting simulation or efficient implementation. Many of these applications are "mostly-static" hybrids in that they involve static dataflow components along with a relatively small proportion of dynamic components. We show an approach to modeling and scheduling of such hybrid communication system applications using CFDF.

The scheduling technique presented in this section is applicable to a class of applications that we have already explained in Sections 3.5.5 and 4.3. In these sections, we have also introduced the AMS, which we use as a case study, and modeled it using

CFDF model. In this work, we show how efficient PLSs can be derived from the CFDF representations. for this restricted class of the CFDF applications. This restricted form is defined in a way that introduces a new trade-off point between expressive power and analysis potential that is useful for modeling of modern communication systems.

5.3.1 Related Work

We have explained in previous chapters use of dataflow models like BDF to model dynamic dataflow behavior. The problem of scheduling such dynamic dataflow applications has also been studied, and important results have been established regarding bounded memory execution and compile-time scheduling (e.g., see [16, 53]). Most of these approaches employ scheduling schemes that suffer from significant run-time overhead, difficulties in code generation, and lack of compile-time predictability (e.g., for validating real-time signal processing performance). The scheduling techniques described in earlier sections — using dynamic dataflow graph decomposition and mode grouping — do not in general guarantee bounded memory execution for the entire input application.

A meta-modeling technique such as PDF [6] supports limited forms of dynamic behavior and has more compile-time predictability than more general kinds of dynamic dataflow models such as BDF. A useful feature of PSDF, for example, is its capability of efficient quasi-static scheduling in terms of PLSs [6]. This class of schedules allows for flexible, compact specification of nested loop structures, where loop iteration counts can be either constant values or symbolic expressions in terms of dynamic parameters in the underlying dataflow graph. While PDF is useful for many kinds of signal processing

applications, it imposes significant restrictions on how applications are modeled (e.g., in terms of hierarchies of cooperating *init*, *subinit*, and *body* graphs [6]), and in general, major changes in the user interface are required to provide direct support for PDF in a design tool.

In contrast to the approaches for scheduling BDF or PDF graphs, the approach that we present here provides PLS-based bounded memory scheduling while operating within a semantic framework that can be integrated more directly into existing design tools compared to the more hierarchical semantic structure of PDF representations.

5.3.2 Constructing Parameterized Looped Schedules

This technique, developed in [70], is applicable to a class of applications shown in Fig. 3.6(a). It seeks to generate efficient PLSs to reduce the run-time overhead associated with dynamic scheduling. Such quasi-static schedules are also useful from a code generation perspective as the only dynamic components of such schedules are the loop iteration counts. Our approach finds static regions in the application graph that can be clustered and completely scheduled at compile-time. It then proceeds to identify the dynamic components along with the corresponding static components, which must execute varying numbers of times in relation to the dynamic components. We then merge the appropriately-iterated static and dynamic components into a single PLS.

The following sequence of steps outlines our algorithm for PLS construction (see Fig. 3.6(a)):

1. Identify SDF components in the dataflow graph and cluster them individually to

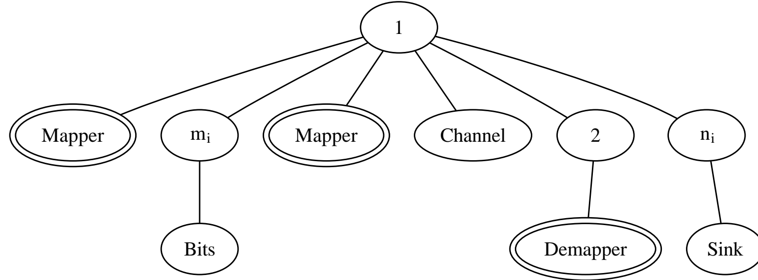


Figure 5.5: Valid PLS for the application in Fig. 3.6(b).

obtain SDF-clustered regions S_1, S_2, and S_3. This step can be performed efficiently since the specification of a CFDF actor mode includes the associated production or consumption rate for each actor port.

2. Use established SDF scheduling techniques [11, 37] for scheduling the SDF regions identified in step 1, assuming that a valid consistent schedule exists for each of the SDF subgraphs [46].

3. Identify the pair D_1 and D_2 of actors with dynamic behavior, and determine which of the SDF sub-schedule loop (iteration) counts are dependent on D_1 and D_2.

4. Combine static sub-schedules into a PLS in which parameterized loop count expressions are set up at compile time, and symbolic parameters in these expressions are varied at run-time.

Fig. 5.5 illustrates a PLS for the dataflow graph in Fig. 3.6(b) that is derived using our approach to PLS construction.

As can be seen from the GST in Fig. 5.5, CFDF actors use *guarded execution*, while other SDF actors are fired using *unguarded execution* in which the actor is fired

without checking if it is enabled (enabling is guaranteed through a carefully constructed PLS). The values of m_i and n_i are determined dynamically by the simulator based on the current modes of the `mapper` and `demapper` actors, respectively. Since the mode of an actor is visible to the simulator (through a flexible mode-querying mechanism in our implementation of CFDF), it can be used to set loop counts based on dynamically-changing execution state of the actor.

For the class of applications targeted in this section of the thesis (see Sections 3.5.5 and 4.3) and PLSs generated using the algorithm described in this section, the number of tokens accumulated on the edge e_1 (e_4) after the ith iteration is related to m_i and c_i (n_i and p_i). These expressions can then be used to prove that the numbers of tokens accumulated on edges e_1 and e_4 are bounded by $k + c_{\max} - 1$ and $q + p_{\max} - 1$, respectively (see Section 5.3.3). Together with bounds that are derived based on the static dataflow properties of the other edges, this leads to a bound on total buffer memory requirement that can be computed at compile-time. Such bounds provide for more efficient execution or simulation (since dynamic memory allocation is not required) as well as enhanced predictability and reliability.

5.3.3 Bounded Memory Execution

Since the CFDF model is Turing complete, the problem of determining whether a CFDF graph can be scheduled within bounded memory in finite time is undecidable [16]. However, for a class of applications, the graphical representations of which can be reduced to the topology shown in Fig. 3.6(a), we can guarantee a bounded memory execu-

tion, if one exists.

Consider the dataflow graph shown in Fig. 3.6(a). We assume that there exist valid, consistent schedules for the SDF clusters in this dataflow graph [46, 11]. Note that such schedules are periodic schedules that execute with bounded memory. From the graph topology, it is clear that we have a simple cycle, where the feedback edge and all other edges in the cycle are single-rate. Hence, the only buffer edges where we can have unbounded accumulation of data tokens are the edges that connect SDF cluster S_1 with the actor D_1 and the actor D_2 with SDF cluster S_3. With these assumptions and observations, it suffices to show that these two buffers can be bounded in order to establish bounded memory execution for the application.

Consider the edge e_1. Let t_i denote the number of tokens accumulated on an edge e_1 after the ith iteration of the entire graph schedule, for $i = 0, 1, \ldots$ with $t_0 = 0$. Let m_i and c_i denote the value of the parameterized loop counts and the number of tokens consumed from edge e_1 during the ith iteration, respectively, for $i = 1, 2, \ldots$. From the dataflow graph topology shown in Fig. 3.6(a) and (b), and the schedule shown in Fig. 5.5, we have

$$t_i \quad = \quad k \cdot m_i + t_{i-1} - c_i, \tag{5.1}$$

where

$$m_i \quad = \quad \lceil (c_i - t_{i-1})/k \rceil \,. \tag{5.2}$$

Substituting (5.2) in (5.1) and using the following relation [18]

$$\lceil (a/b) \rceil \leq (a + b - 1)/b, \ \forall \, \text{integers} \, a, b > 0,$$

it can be shown that

$$t_i \leq k - 1. \tag{5.3}$$

Now the maximum number of tokens that is ever queued on the buffer e_1 is bounded above by the sum of the maximum number of tokens that remain after a schedule iteration and the maximum number of tokens that can be consumed from the buffer during an iteration. Thus, the number of tokens that are accumulated in the buffer for edge e_1 is bounded above by $(k + c_{\max} - 1)$. Similarly, we can show that the buffer for edge e_4 has an upper bound on the number of tokens accumulated in it that is given by $(q + p_{\max} - 1)$.

We thus have upper bounds on the numbers of tokens accumulated in dataflow buffers for which dataflow behavior varies during run-time. These bounds, together with bounds that are derived based on the static dataflow properties of the other edges, provide a bound on the total buffer memory requirement. Moreover, this aggregate bound can be computed at compile-time, which has advantages as mentioned in Section 5.3.2.

5.3.4 Simulation Results

We have implemented the approaches to CFDF modeling and PLS construction using the *Advanced Design System* (*ADS*) tool from Agilent Technologies, Inc. [60]. We have employed the CFDF model for dynamic actors along with the existing SDF based actors in Agilent ADS. Using such a design approach, we implemented the AMS application shown in Fig. 3.6(b). Results for simulations of PLS-based execution of the AMS application, as implemented in Agilent ADS, are presented in [70].

In this thesis, we use DIF to prototype the AMS application using the CFDF model

and generate a PLS using the algorithm described in [70] as well as Section 5.3.2. In [70], we compared the performance of a PLS with that from a canonical schedule. Extending [70] further, we also compare the performance of a PLS with that of the *dynamic dataflow schedule* (*DDFS*) employed in Ptolemy II [16, 15]. For this, we have implemented the algorithm used in Ptolemy II to generate a DDFS in DIF. A DDFS is widely used to simulate dynamic dataflow applications. In a DDFS, at any given time during a simulation, the scheduler determines whether each of the actors is enabled or deferrable. A deferrable actor is the one that is enabled, but does not require to be fired in order for any of its downstream actors to be enabled. Correspondingly, we may also have actors that are *enabled but not deferrable* (*EBND*) — that is, actors that are enabled, and must be fired in order to have sufficient number of tokens produced at their outputs for one or more of their downstream actors to be enabled. Among the enabled actors, a DDFS first attempts to fire an actor only from the set of actors that are EBND. If there is no such actor, it proceeds to fire one of the enabled actors. After every invocation of an enabled actor, the DDFS checks if the invocation has changed the status of any of the other actors in the graph in terms of being enabled or deferrable. The execution proceeds until the control condition (a pre-specified condition for terminating graph execution) is reached or a deadlock condition is reached. A DDFS is designed to fire actors only when required for enabling downstream actors. By doing so, it aims to minimize the total buffer requirements for the application graph.

The results of our experiments for different sink control conditions (the total number of tokens that must be consumed by the sink actor during the simulation) are shown in Table 5.3 and Table 5.4.

Table 5.3: Average simulation time for different sink control conditions (numbers of tokens consumed by the sink actor) using a canonical schedule (CS), DDFS, and PLS.

Sink control condition	Average simulation time (ms)			Reduction (%) compared to	
	CS	DDFS	PLS	CS	DDFS
10000	78	186	73	6.41	60.75
20000	111	301	90	18.92	70.10
50000	222	665	148	33.33	77.74
100000	401	1313	248	38.15	81.11

Table 5.4: Total buffer requirements for different sink control conditions (numbers of tokens consumed by the sink) for a canonical schedule (CS), DDFS, and PLS.

Sink control condition	Total buffer requirement (number of tokens)					
	CS Minimum	CS Average	CS Maximum	DDFS	PLS Experiments	PLS Theory
10000	25	28	37	19	19	19
20000	27	31	37	19	19	19
50000	29	35	45	19	19	19
100000	31	34	37	19	19	19

As evident from the results, the PLS method exhibits significant reductions in run-time overhead over a canonical schedule and DDFS, which leads to improvements in average simulation time — up to 38% over a canonical schedule and 81% over a DDFS in our DIF implementation. Speed-up in simulation over a DDFS can be attributed to elimination of run-time overheads corresponding to determining the status of each actor (enabled or deferrable) in the graph. For a canonical schedule, total buffer requirements vary from one simulation of an application to another owing to dynamically changing dataflow behavior. We have reported minimum, maximum, and average buffer requirements from our experiments using a canonical schedule. It should be noted that the total buffer requirements for a canonical schedule not only vary significantly across different

simulations but also are much higher compared to that of a PLS or DDFS. These experiments not only confirm the theoretical buffer bounds for PLSs estimated using the results mentioned above but also demonstrate significant reductions in the total buffer memory requirements over the canonical schedule, especially for larger values of sink control conditions (i.e., longer simulations). Since a DDFS employs a strategy in which actors are fired only when needed, it tries to optimize the total buffer requirements. For the AMS application in our experimental study, our PLS can achieve this optimized buffer requirement, but with much higher simulation speeds.

In summary, from our study, trade-offs among the scheduling techniques that we have examined in this section can be summarized by listing the advantages of each of the techniques as follows:

- **CFDF canonical scheduler** offers simplicity of implementation (e.g., for fast, early stage prototyping) and generality (arbitrary topologies can be handled).
- **DDFS** offers buffer size minimization, and generality.
- **PLS** offers buffer size minimization with compile-time analysis, and fast simulation performance.

Intuitively, while the canonical schedule and DDFS offer generality for the associated dynamic dataflow modeling techniques, PLSs typically require significant amounts of static or quasi-static structure to be useful — however, when such structure can be found, their benefits can be significant, as shown in our case study using the AMS application. Furthermore, it is conceivable that the performance benefits presented here for

PLSs can be extended beyond simulation to implementation/synthesis scenarios; this is a useful direction for further investigation.

5.4 Summary

We have presented a generalized scheduling strategy for scheduling dynamic dataflow applications that leverages CFDF semantics, which structures dynamic actors as a set of modes with fixed behavior. We presented an algorithm that decomposes dynamic dataflow graphs into a set of dynamically interacting static dataflow graphs. We demonstrated this on mixed-model applications with existing schedulers, which gave a positive indication of the utility of this approach for software implementations of such dynamic dataflow applications. An immediate direction of future work is to improve the sophistication of the simulator. With a more intelligent way of dynamically switching between the resulting static schedule trees, we should achieve better run times and smaller maximum buffer sizes. A limitation of our approach, compared to related techniques, is that special attention is required by the designer to explicitly specify the dataflow properties associated with individual modes, and attention is also needed during testing to validate that the declared and observed behaviors match. An interesting direction for future work is the integration of our proposed scheduling methods with more formal reasoning about actor sub-behaviors, such as those being developed in conjunction with languages and models such as CAL and SysteMoc.

We have presented a generalized scheduling approach with mode grouping that exposes more static behavior of a dynamic application graph. By identifying static groups

of "modes" inside actors, we expose more of the static nature of the application, allowing traditional scheduling techniques to improve on memory requirements by up to 37%. Developing dynamic schedule tree selector so that a simulator or a final implementation may strategically switch between the known static behaviors at run-time is a useful direction for future work.

Our PLS approach identifies the underlying static components in the application, systematically integrates the well-established compile-time scheduling techniques for SDF graphs with more flexible CFDF semantics, and uses combined CFDF/SDF analysis to generate PLSs that have significantly reduced run-time overhead, guaranteed memory bounds, and reduced memory requirements. Our approach, therefore, provides robust simulation of dynamic communication applications without major limitations on compile-time predictability and efficient scheduling.

Chapter 6

Dataflow-based Rapid Prototyping for Radio Astronomy Signal Processing

There is a growing trend toward using high-level tools for design and implementation of radio astronomy DSP systems. Such tools, for example, those from the CASPER group, are usually platform-specific, and lack high-level, platform-independent, portable, scalable application specifications. This limits the designer's ability to experiment with designs at a high-level of abstraction and early in the development cycle. We address some of these issues using a model based design approach employing dataflow modeling, which is extensively used in design of embedded DSP systems. We use an application employing a TDD to allow narrow band modes in spectrometers as a driving and demonstrative application. Our design is targeted toward an FPGA platform, called the IBOB, that is available from the CASPER group. By a TDD, we imply a hardware digital downconverter design that can be reconfigured without the need for regenerating the hardware code. Such a design is currently not supported in the CASPER DSP library. The work presented in this chapter focuses on two aspects. Firstly, we introduce and demonstrate a dataflow-based design approach using the DIF tool for high-level application specification, and we integrate this approach with the CASPER tool flow. Secondly, we explore the trade-off between the flexibility of TDD designs and the low hardware cost of fixed-configuration digital downconverter (FDD) designs that use the available CASPER DSP

library. We further explore this trade-off in the context of a two-stage downconversion scheme employing a combination of TDD or FDD designs.

6.1 Introduction

Key challenges in designing DSP systems employed in the field of radio astronomy arise from the need to process very large amounts of data at very high rates arriving from one or more telescopes. It is also desirable to have scalable and reconfigurable designs for shorter development cycles and faster deployment. Moreover, these designs should be portable to different platforms to keep up with advances in new hardware technologies. However, conventional design methodologies for signal processing systems in the field of radio astronomy focus on custom designs that are platform-specific. Such designs, by virtue of being platform-specific, are highly specialized, and thus difficult to retarget. The design approaches also lack high-level platform-independent application specifications that can be experimented with, and later ported to and optimized for various target platforms. This limits the scalability, reconfigurability, portability, and evolvability across varying requirements and platforms of such DSP systems.

A model based approach for design and implementation of a DSP system can effectively exploit the semantics of underlying models of computation for precise estimation and optimization of system performance and resource requirements (e.g., see [8]). Though approaches for scalable and reconfigurable design based on modular FPGA hardware and software libraries have been developed (e.g., see [55, 56]), they neither allow for high-level abstraction nor provide linkage to formal models of computation.

We propose an approach using DSP-oriented dataflow models of computation to address some of these issues [46]. Dataflow modeling is extensively used in developing embedded systems for signal processing and communication applications, and electronic design automation [8]. Our design methodology involves specifying the application in DIF [36] using an appropriate dataflow model. This application specification is transformed into an intermediate, graphical representation, which can be further processed using graph transformations. The DIF tool allows designers to verify the functional correctness of the application, estimate resource requirements, and experiment with various dataflow graph transformations, which help to analyze or optimize the design in terms of specific objectives. The DIF-based dataflow specification is then used as a reference while developing a platform-specific implementation. We show how formal understanding of the dataflow behavior from the software prototype allows more efficient prototyping and experimentation at a much earlier stage in the design cycle compared to conventional design approaches.

As mentioned earlier, we demonstrate our approach using the design of a TDD that allows fine-grain spectroscopy on narrow-band signals. The TDD, which was originally designed for the *Green Bank Ultimate Pulsar Processing Instrument* (*GUPPI* at the NRAO, Green Bank, finds its use in the spectrometers currently under development for the GBT and 20m telescope at the NRAO, Green Bank. One of the motivations has been to have a TDD design, where by a TDD, we mean a digital downconversion system that supports changes to the targeted downconversion ratio without requiring regeneration of the corresponding hardware code. Development of such a TDD is a significant contribution of this work. We compare our TDD with the FDD designs that use the current DSP

library from the CASPER group. Through this kind of comparison, we explore trade-offs between the flexibility offered by TDD designs and their hardware cost.

6.2 Related Work

There exist high-end reusable, modular, scalable, and reconfigurable FPGA platforms such as the *Berkeley Emulation Engine 2* (*BEE2*) and IBOB, which have been introduced specifically for DSP systems [17]. The BEE2 uses SDF as a unified computation model for both the microprocessor and the reconfigurable fabric. It uses a high-level block diagram design environment based on The Mathworks' Simulink and the XSG. This design environment, however, does not expose the underlying dataflow model. In particular, the designer has little or no scope to make use of the underlying dataflow model for experimentation. Also, the SDF model used for programming the BEE2 is a static dataflow model in that all the dataflow information is available at compile-time (i.e., before executing or running the application). Though this feature provides maximal compile-time predictability, it has limited expressive power. It does not allow for data-dependent, dynamic behavior, which is exhibited by many modern DSP applications, such as the TDD application introduced in Section 6.3. Other forms of dataflow models that can capture more application dynamics with acceptable levels of compile-time predictability may better exploit the features offered by platforms such as the BEE2.

Model based approaches for designing large scale signal processing systems with a focus on radio telescopes has been previously studied (e.g., see [2, 50, 49]). Several frameworks have been proposed for model based, high-level abstractions of architec-

tures along with performance/cost estimation methods to guide the designer throughout the development cycle (see [2]). However, the focus of these approaches has been on architecture exploration. There have also been attempts to derive implementation-level specifications starting from system-level specifications by segregating signal processing and control flow into an application specification and architecture specification, respectively (see [50, 49]). However, the choice of models of computation has been made primarily from control flow considerations rather than dataflow considerations. These approaches, though relevant, do not specifically address the issue of high-level application specification for platform-independent prototyping and use of models of computation for abstraction of heterogeneous or hybrid dataflow behaviors. This issue is critical to efficient prototyping of high performance signal processing applications, which are typically dataflow dominated, and include increasing levels of dynamic dataflow behavior (e.g., see [8]).

We address this issue using the CFDF model with underlying PSDF or PCSDF behavior and using it for system prototyping. We then show how platform-independent specifications based on this modeling technique can be used to efficiently develop platform-specific implementations.

6.3 Tunable Digital Downconverter

In DSP literature, the terms downsampling, decimation, and downconversion are often used interchangeably. In this chapter, a *decimator* refers to a block that simply decimates, downsamples, or downconverts the input signal without any other processing (e.g.,

see Fig. 2.2(a) and (b)). The ratio of the sampling rate at the input of a decimator to that at its output is referred to as its *decimation factor*. A decimator is generally preceded by an anti-aliasing filter [80]. In this chapter, we refer to such a combined structure, consisting of a filter and decimator, as a *decimation filter* (e.g., see Fig. 2.3(a) and (b)). In a polyphase implementation of a decimation filter, such as the one we use in our implementation, this structure is implemented as a single computing block [80]. We refer to the system or application that employs a decimator or decimation filter, possibly with other blocks such as mixers and filters, as a digital downconverter, and in particular, a FDD or TDD (e.g., see Fig. 6.1). The decimation factor of a decimation filter, TDD, or FDD refers to that of the decimator in it.

Fig. 6.1 shows a block diagram of a TDD application. It shows an 8-bit analog-to-digital converter (ADC) that receives a baseband input IF signal of bandwidth 800 MHz and samples it at the sampling rate of 1.6 giga-samples/second (GS/s). The internal design of the ADC block is such that 8 samples, where each sample is an 8-bit fixed point number, are output on the eight output lines at the same clock pulse. This results in 200 mega-samples/second (MS/s) on each of the outputs of the ADC block. Correspondingly, all the downstream blocks also have 8 input and output ports. We, thus, have 8 connections between any two blocks shown in Fig. 6.1 that are directly connected. We, however, have not shown this detail (all 8 connections) for the sake of clarity and simplicity.

The TDD subsystem, identified by the dotted box in Fig. 6.1, has to downsample this signal so that the resultant signal at the output of the TDD will have a tunable band of user-specified bandwidth (B_w) and center frequency (C_f). The output of the TDD is fed to the downstream DSP blocks over the 10x auxiliary user interface (XAUI) ports. The

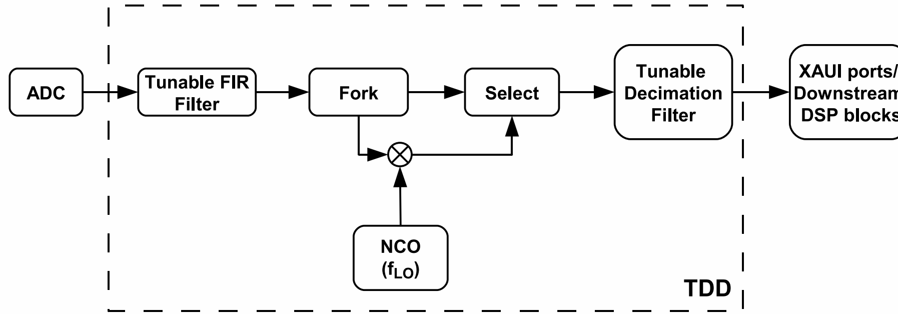


Figure 6.1: Block diagram of a tunable digital downconverter.

XAUI ports stream data over CX4 connectors of the IBOB, and have a maximum data transfer capability of 10 giga-bits per second (Gbps).

During narrow-band observation modes, the TDD allows narrow-band signals to be sampled at their corresponding Nyquist rates. When presented to the same number of channels in the downstream DSP system as that for the wide-band IF input, this allows fine-grain spectroscopy. Our TDD design supports integer decimation factors between 5 and 12. The valid values of C_f corresponding to the selected B_w can vary so as to span the entire 800 MHz IF input.

As shown in Fig. 6.1, the TDD consists of a tunable FIR filter. If the desired output is a baseband signal, then the FIR filter simply acts as a rectangular window with each of its taps set to 1. Also, in this case, the fork and select blocks are configured to route the output of the FIR filter directly to the tunable decimation filter (TDF), bypassing the mixer.

If the desired output is not a baseband signal, the FIR filter acts as a bandpass filter (BPF). The cut-off frequencies for this BPF are set using the specified parameter configuration (B_w and C_f). In this case, the output of the BPF is fed to a real mixer,

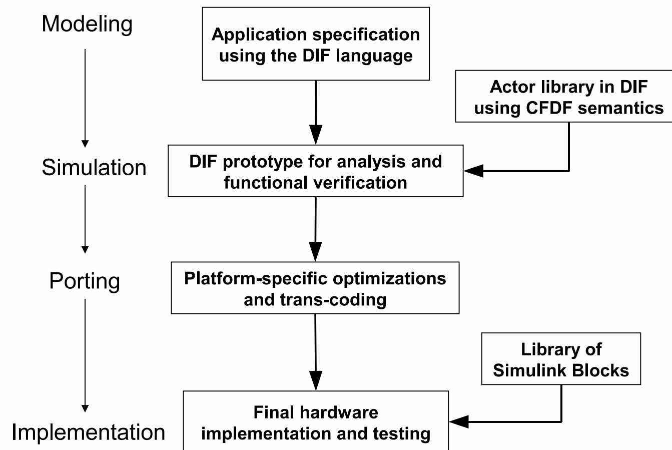


Figure 6.2: Proposed dataflow-based approach.

which translates it into a baseband signal. The local oscillator, with a frequency f_{LO}, is implemented as a numerically controlled oscillator (NCO). The frequency, f_{LO}, is dependent on the value of C_f and B_w. The output of the mixer is then fed to the TDF, which downsamples its input depending upon the specified B_w or decimation factor.

6.4 Dataflow-based Design and Implementation

We propose an approach for design and implementation of a TDD based on the dataflow formalisms discussed in Section 2.1 along with relevant capabilities of the DIF tool described in Section 2.2. Fig. 6.2 gives an overview of our dataflow based approach, which we now describe.

6.4.1 Modeling and Prototyping using DIF

We start with an application specification that describes the DSP algorithm under consideration (for example, here, TDD) along with proper input and output interfaces. The application is specified using the DIF language. This DIF specification consists of topological information about the dataflow graph — interconnections between the actors along with input and output interfaces. The DIF specification is a platform-independent, high-level application specification. The specification can be used, for example, to simulate the application, given the library of actors from which the specification is constructed.

Depending upon the application under consideration, the designer can select from among a variety of dataflow models of computation in DIF to effectively capture relevant aspects of the application dynamics. It should be noted that the designer does not always need to a specify the model in advance. The CFDF model can be used to describe individual modules (actors) in the application, and the DIF package can analyze the CFDF representation (CFDF modes, to be specific) of the actors, as specified by the designer through the actor code, and annotate the actors with additional dataflow information using various techniques for identifying specialized forms of dataflow behavior (e.g., see [67]). This step requires the functionality of individual actors to be specified in CFDF semantics. The designer can use the existing blocks from the Java actor library in DIF or develop his or her own library of CFDF actors.

In terms of tunability, the key components of the TDD as seen from Fig. 6.1 are the tunable FIR filter, and decimation filter blocks. The TDF block is of particular interest, considering that it is the only multirate block in the system. Its behavior resembles that of

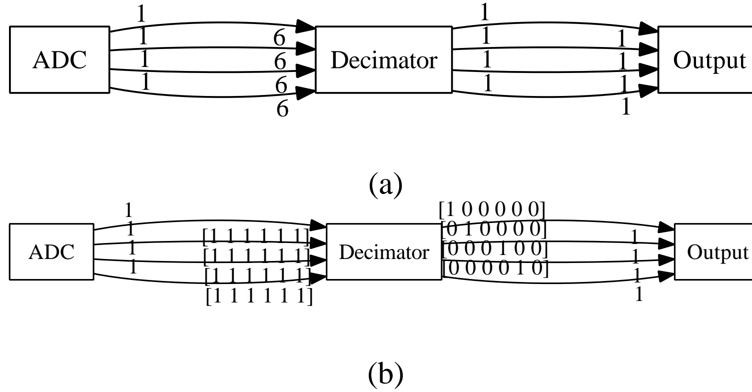


Figure 6.3: Dataflow behavior of a `Decimator` actor with 4 inputs and outputs for a *decimation factor* of 6 using (a) SDF, and (b) CSDF models.

the one described in Section 2.1.3. In view of this, we have identified PSDF and PCSDF as candidate dataflow models for efficient implementation of the targeted TDD system. For this system, we have to take into account the multiple inputs and outputs to actors, as mentioned in Section 6.3.

To illustrate details of the dataflow behavior of a decimator actor based on such specifications, we have shown one such `Decimator` actor with 4 inputs and outputs, and having a decimation factor of 6 in Fig. 6.3(a) and Fig. 6.3(b). For the sake of simplicity and clarity, we have excluded the other single rate blocks from the application graphs in this figure. In our implementation, we extend this behavior for an actor with 8 inputs and outputs. We have created a DIF prototype using PSDF and PCSDF as underlying models for equivalent CFDF representation of actor blocks. We have also developed a Java library of actors in DIF adhering to CFDF semantics for all of the blocks.

We then used DIF for software prototyping, analysis, and functional simulation. The DIF package uses the DIF specification to generate an intermediate graph represen-

tation, which can then be used as an input for further graph transformations including a *scheduling* transformation, which determines the schedule for an application. Here, by a *schedule*, we mean the assignment of actors to processing resources, and the execution ordering of actors that share the same resource. The functional simulation capabilities provided in DIF can be used to analyze and estimate buffer requirements in terms of the numbers of tokens accumulated on the buffers that correspond to dataflow graph edges. This provides an estimate of total memory requirements as well as specifications for individual buffers when porting the application to the targeted implementation platform.

Fig. 6.4 shows the TDD application graph generated using DIF. This is based on the TDD block diagram shown in Fig. 6.1 with addition of some actors that handle parameter configuration for the actors. We discard one of the two sets of outputs (more specifically, *sine* output) of the `localOsc` actor as we have employed a real mixer in our design. The complexity of the graph, which is increased due to multiple parallel edges between two actors, can easily be captured through a DIF specification that makes use of topological patterns. We have shown two possible specifications of the graph topology in DIF using topological patterns in Fig. 6.5 and Fig. 6.6.

Using the TDD specification and employing the notion of PLSs described in Section 5.3, we construct PLSs for the TDD application. Fig. 6.7(a) shows a PLS for a TDD application, where the `decimator` actor has the underlying SDF model, while Fig. 6.7(b) shows one in which the `decimator` actor employs the CSDF model. As annotated in these GSTs, loop counts `p0`, `p1`, and `p2` are parameterizable. The loop count `p0` is set to a user-specified number of iterations, while the loop counts `p1` and `p2` are tuned based upon the decimation factor as well as the underlying dataflow model for the

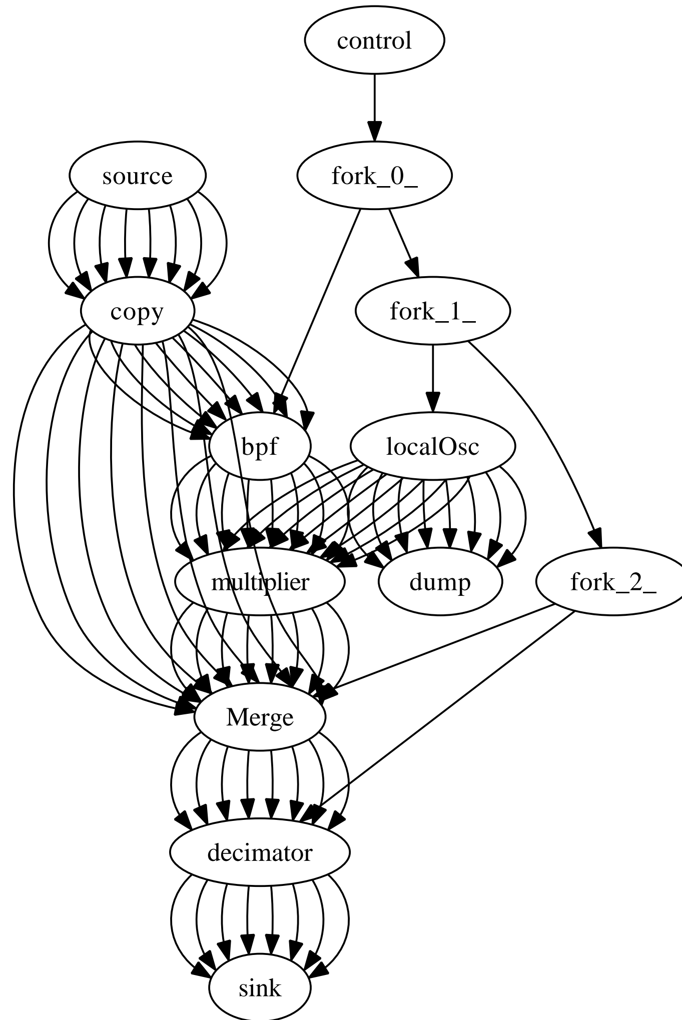


Figure 6.4: TDD application graph generated using DIF.

`decimator`. Fig. 6.7(a) and (b), in particular, show values of the parameterizable loop counts set for a `decimator` with a decimation factor of 11. This PLS can be viewed as providing CFDF-based execution for the given PDF-based actor specification model. The effect of various decimation factors on the total buffer requirements (in number of tokens) is shown in Table 6.1.

Table 6.1 shows the total buffer requirements using PLSs for various configurations of decimation factors. Note that for a given configuration (setting of graph parameters), a

```
topology {
    nodes = source, copy, bpf, Merge, decimator,
            sink, control, fork[3], multiplier,
            localOsc, dump;
    edges = soCp[8] -> multiedge(source, copy),
            cpMrg[8] -> multiedge(copy, Merge),
            cpBpf[8] -> multiedge(copy, bpf),
            bpfMul[8] -> multiedge(bpf, multiplier),
            mulMrg[8] -> multiedge(multiplier, Merge),
            mrgDec[8] -> multiedge(Merge, decimator),
            decSnk[8] -> multiedge(decimator, sink),
            loMul[8] -> multiedge(localOsc, multiplier),
            loDump[8] -> multiedge(localOsc, dump),
            conFrk, f0f1, f1f2 -> chain(control, fork[0:2]),
            f0Bpf(fork[0], bpf),
            f1Lo(fork[1], localOsc),
            f2Mrg(fork[2], Merge),
            f2Dec(fork[2], decimator);
}
```

Figure 6.5: Partial DIF specification — `topology` block — for the TDD application graph using topological patterns.

PSDF or PCSDF graph behaves like an SDF or CSDF graph, respectively. It can be seen that for the SDF model, the total buffer requirements vary with the decimation factor, and this is due to input buffers to the TDD block that need to accumulate varying numbers of tokens. Thus, employing the PSDF model will require tuning buffer sizes for different decimation factors if one wants to provide for optimized buffer sizes in terms of graph parameters.

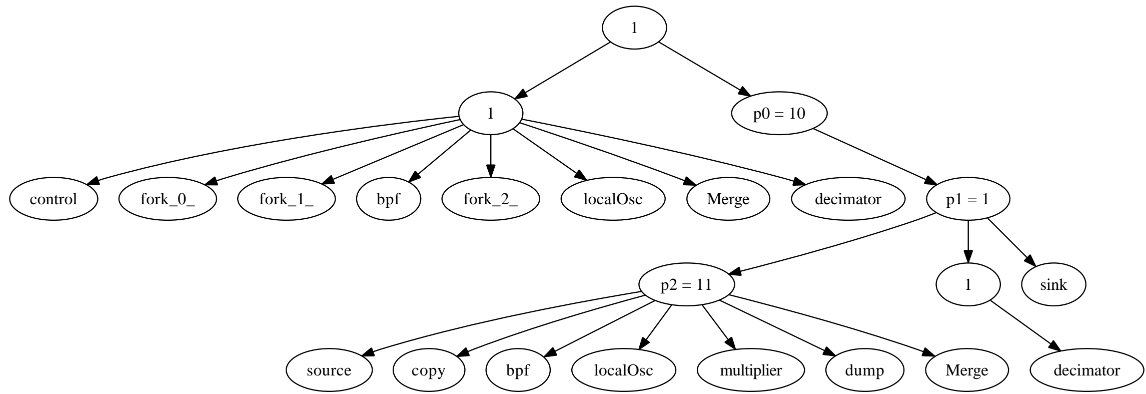
We have used the CASPER tool flow for developing our platform-specific implementation as explained later in Section 6.4.2. This implementation is targeted to an FPGA. Our objective here is to support tuning the decimation factor without regenerating hardware code. A dataflow buffer can be emulated using dual-port random access

```
topology {
    nodes = source, copy, bpf, Merge, decimator, sink,
            control, fork[3], multiplier, localOsc, dump;
    edges = c0[6] -> chain(source, copy, bpf,
            multiplier, Merge, decimator, sink),
            c1[6] -> chain(source, copy, bpf,
            multiplier, Merge, decimator, sink),
            c2[6] -> chain(source, copy, bpf,
            multiplier, Merge, decimator, sink),
            c3[6] -> chain(source, copy, bpf,
            multiplier, Merge, decimator, sink),
            c4[6] -> chain(source, copy, bpf,
            multiplier, Merge, decimator, sink),
            c5[6] -> chain(source, copy, bpf,
            multiplier, Merge, decimator, sink),
            c6[6] -> chain(source, copy, bpf,
            multiplier, Merge, decimator, sink),
            c7[6] -> chain(source, copy, bpf,
            multiplier, Merge, decimator, sink),
            cpMrg[8] -> multiedge(copy, Merge),
            loMul[8] -> multiedge(localOsc, multiplier),
            loDump[8] -> multiedge(localOsc, dump),
            conFrk, f0f1, f1f2 -> chain(control, fork[0:2]),
            f0Bpf, f1Lo, f2Mrg, f2Dec -> parallel(fork[0:2],
            fork[2], bpf, localOsc, Merge, decimator);
}
```

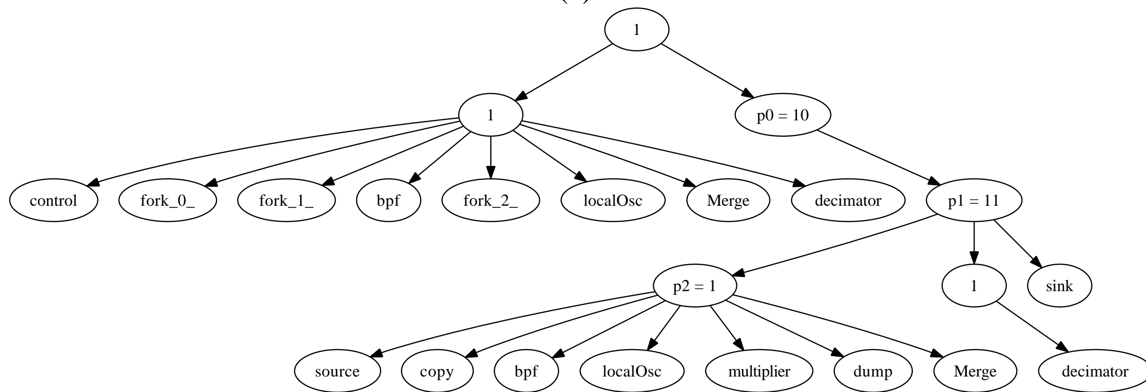
Figure 6.6: Partial DIF specification — `topology` block — for the TDD application graph using topological patterns.

memory (RAM) blocks in the targeted FPGA device, but tuning the sizes of such blocks is not possible during run-time. The ADC output is of a streaming nature (data is produced or consumed at every clock cycle without any synchronization signal), as is the DSP subsystem downstream of the TDD.

In order to achieve the throughput constraint imposed by the maximum data rate of the ADC output stream, SDF buffers need to be pipelined, which is not efficient using RAM blocks. Thus, we use the CSDF model, which does not require tuning of dataflow



(a)



(b)

Figure 6.7: PLSs for the TDD application configured for a decimation factor of 11, and `decimator` actor employing the (a) PSDF and (b) PCSDF models of computation.

Table 6.1: Total buffer requirements from a DIF prototype for different decimation factors using parameterized looped schedules.

Decimation Factor		5	6	7	8	9	10	11	12
Total buffer requirements	SDF	132	140	148	156	164	172	180	188
(Number of tokens)	CSDF	100	100	100	100	100	100	100	100

buffer sizes to achieve the maximum throughput constraint, as observed from our DIF-based prototype. A synchronization or enable signal derived from the TDD is used as a clock to drive the downstream DSP system. This signal is a decimated version of the clock signal.

We use our DIF prototype as a reference while integrating the design with the current CASPER tool flow for the target implementation on the IBOB. Section 6.4.2 further elaborates on this approach along with implementation results.

6.4.2 Integration with the CASPER Tool Flow

The CASPER tool flow is based on the BEE_XPS tool flow [56]. This tool flow requires that an application be specified as a Simulink model using XSG [56]. Since we do not have an automated tool for transforming a DIF representation into an equivalent Simulink model, porting the DIF specification to Simulink/XSG requires manual transcoding of the DIF specification. This also requires implementing parameterizable actor blocks that are currently not supported in the XSG, CASPER, or BEE_XPS libraries.

Each actor gets transformed into an equivalent functional XSG block. For each of the Simulink actor blocks, we provide a pre-synthesis parameterization that allows changing block parameters before hardware synthesis (see [57] for more details on Simulink

scripting). In order to implement our objective of tunability — post-synthesis parameterization — we use the *software register* mechanism in the BEE_XPS library to specify parameters that change during run-time (that is, after hardware code is generated, and depending upon user requirements.)

Software registers can be accessed and set during run-time from the TinyShell interface available for IBOB. This allows tuning TDD parameters without re-synthesizing the hardware each time the parameters change from the previous setting. Each block has an enable input signal. Through systematic transformations, an application graph in DIF can be converted into an equivalent Simulink/XSG model. We have developed an interface software package using C programs, and Bash and Python scripts to compute software register values for the required TDD configuration, and set these values on the IBOB over a telnet connection, which is used for remote access to the hardware platform at NRAO.

On the targeted FPGA device, we have employed dual-port RAM blocks that are loaded with pre-computed sinusoidal signal values of the required precision. Each of these dual-port RAM blocks is used to simultaneously read sine and cosine values from both of its ports. The oscillator frequency is set using a software register, and depends upon the desired output signal band.

The FIR filter associated with the TDF block can have up to 16 taps. Currently, the generic FIR filter without any decimation (used, for example, as a BPF in the design) can have up to 8 taps. These, again, are set using software registers. We have employed two filter banks in our design of a TDF that operate in tandem to allow maximum throughput. Hence, our TDF block has 32 multiplication operations. As mentioned earlier, our TDF design employs a polyphase implementation as described in [80].

Table 6.2: Implementation summary for TDD designs.

Parameter	Design 1	Design 2	Design 3	Design 4
Mixer	No	Yes	No	Yes
Input bandwidth (MHz)	800	800	800	800
Decimation factor D	$5 \leq D \leq 12$	$5 \leq D \leq 12$	$5 \leq D \leq 12$	$5 \leq D \leq 12$
Latency (ns)	65	150	85	190
FPGA slices (Out of 23616)	12234 (52%)	13315 (56%)	12322 (52%)	14232 (60%)
4 input LUTs (Out of 47232)	14139 (29%)	16123 (34%)	12123 (25%)	15035 (31%)
Block RAMs (Out of 232)	41 (17%)	48 (20%)	41 (17%)	48 (20%)
18×18 Multipliers (Out of 232)	—	—	32 (13%)	95 (40%)

Table 6.4.2 shows results for the TDD implementation on the IBOB using the Xilinx EDK 7.1.2. We have used this hardware platform and tool for all of the experiments reported on in the remainder of this chapter. Design 1 shows some of the device utilization parameters for a TDD that supports only baseband modes. This design does not include the tunable FIR filter, NCO, and mixer blocks shown in Fig. 6.1. Design 2 is based on the block diagram of a TDD shown in Fig. 6.1. As evaluation metrics for hardware cost, we have used the utilization of FPGA slices, 4-input look-up tables (LUTs), and block RAM units, and the number of embedded multipliers. Note that neither of these two designs use any of the available embedded multipliers for multiplication. Designs 3 and 4 are modified versions of designs 1 and 2, respectively, in that they employ embedded 18×18 multipliers. It can be seen that using embedded multipliers does not provide significant improvements in hardware cost. We observe that use of embedded multipliers, in fact, needs to be accompanied by addition of extra latency in the design to achieve timing

closure. We have been able to achieve maximal throughput using an implementation based on the PCSDF model as explained in Section 6.4.1.

6.4.3 Platform-specific Analysis using DIF

It is common to go back and forth between a high-level prototype and a corresponding platform-specific implementation while designing an embedded DSP system. Such alternation in design phases is common, for example, when one is developing a platform-specific library or tool flow. In support of such a design methodology, it is desirable for a high level design tool to support platform-specific analysis. This can be achieved by annotating the high-level application specification with platform-specific implementation parameters, which are derived through device data sheets, experimentation or some combination of both.

DIF supports specifying user-defined actor parameters. We use this feature in DIF to annotate actors with two relevant implementation parameters — the latency constraint, and number of embedded multipliers. This allows estimating results based on the DIF prototype itself instead of determining them from the constructed design, which is generally time consuming. We have verified the accuracy of metrics estimated by our DIF model compared with actual hardware synthesis results, as shown in Table 6.4.2.

Developers of tool flows and DSP libraries can profile their library blocks to determine a wide variety of platform-specific implementation parameters. DIF can use such information to estimate implementation parameters at a high level of abstraction, and earlier in the design cycle to help efficiently prune segments of the design space. Support

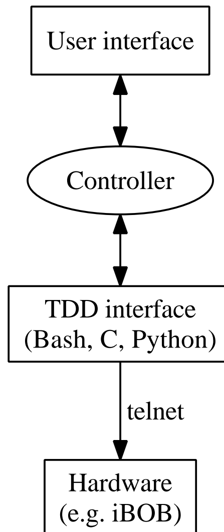


Figure 6.8: TDD System overview.

for estimation of various platform-specific resources for different platforms is beyond the scope of this thesis. It is, however, an important direction toward developing alternative model based design flows and open access tool flows for astronomical DSP solutions.

6.4.4 Software Interface for the Tunable Digital Downconveter

As mentioned earlier, parameterization associated with the TDD blocks is handled through the use of *software registers*. We have developed a TDD interface that computes various programmable TDD parameters, and sets the corresponding software registers by communicating with the IBOB board using the *telnet* utility depending upon the narrow-band mode chosen by the user.

Fig. 6.8 shows an overview of the system organization. A user can specify a valid narrow-band mode through a *user interface*. This information is then provided to the underlying *TDD interface* by a *controller*. The TDD interface uses this information (bandwidth, and center frequency of the narrow-band), along with the decimation factor (either

derived from the input bandwidth or set explicitly), and configures the corresponding TDD block. It then communicates this TDD configuration with the hardware. The TDD interface is implemented as a collection of utilities in the form of C programs, Bash scripts, and Python scripts.

One important aspect in our development of this TDD interface has been our emphasis on unit testing. We have used the unit testing features in the DICE framework (see Section 2.5) to develop an extensive unit test suite, which can be applied for rigorous validation of system functionality, and can be retargeted efficiently across different levels of abstraction (e.g., simulation versus implementation) and different design languages (e.g., C, Verilog and VHDL).

6.5 Exploring Implementation Trade-off with TDD and FDD Blocks

One of the motivations for the work presented in this chapter has been to develop library blocks needed for a TDD using Xilinx LogicCore and CASPER library blocks. The current CASPER DSP library provides a decimator that supports decimation factors that are powers of 2. The decimation factor as well as the filter coefficients of the FIR filter are not tunable after the hardware code is generated. Our design provides flexibility with not only the decimation factor but also the filter coefficients through the use of software registers, as explained earlier. The FDD designs, though not tunable, have lower hardware cost in terms of device utilization. Table 6.5 provides a summary of some of the hardware utilization parameters for the FDD designs. These designs have also been implemented on a CASPER IBOB. Note that the decimation factor of 10 has been achieved

Table 6.3: Implementation summary for FDD designs.

Parameter	Design 1	Design 2	Design 3	Design 4
Mixer	No	No	Yes	Yes
Input bandwidth (MHz)	800	800	800	800
Decimation factor	8	10	8	10
B_w (MHz)	100	80	100	80
C_f (MHz)	50	40	400	400
Latency (ns)	35	440	50	455
FPGA slices (Out of 23616)	4175 (17%)	6142 (26%)	5690 (24%)	6439 (27%)
4 input LUTs (Out of 47232)	5153 (10%)	5216 (11%)	5984 (12%)	6003 (12%)
Block RAMs (Out of 232)	41 (17%)	41 (17%)	49 (21%)	49 (21%)
18×18 Multipliers (Out of 232)	8 (3%)	8 (3%)	32 (13%)	32 (13%)

by first interpolating the input by a factor of 80, and then decimating it by a factor of 8. Comparison between the results in this table and those in Table 6.4.2 clearly highlights the trade-off between design flexibility and hardware cost. Using the model-based approach presented in Section 6.4, the designer can effectively explore this trade-off based on the given design requirements.

6.5.1 TDD and FDD for Multistage Downconversion

Though our TDD design supports limited decimation factors (integer factors between 5 and 12), its usage is not limited to these factors. It can be readily scaled and applied to achieve other decimation factors by cascading multiple TDF blocks. Fig. 6.9 shows some of the possible input/output sampling rate relations that can be achieved by such use of cascaded TDF blocks. Design 1 in Table 6.5.1 employs cascaded TDF

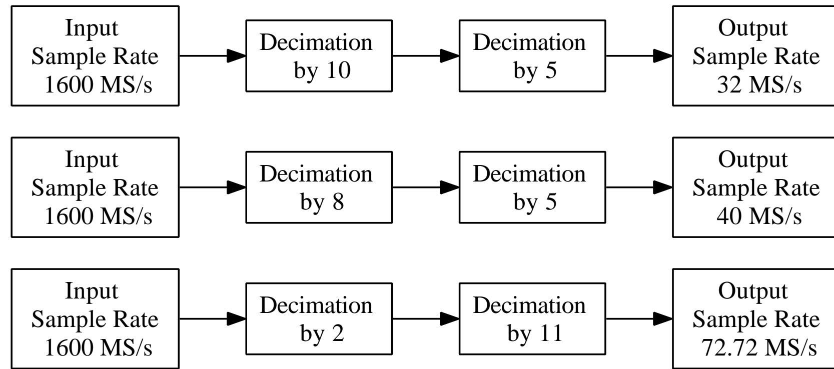


Figure 6.9: Two-stage digital downconversion.

blocks, while design 2 in Table 6.5.1 employs cascaded fixed-configuration decimation filter (FDF) blocks. Both of these designs have been developed to demonstrate multistage downconversion for a baseband signal and neither of them employs a mixer. It is possible to extend these designs to include a mixer to allow all possible narrow band outputs and not just the baseband output. For all of the designs in this table that use one or more TDF blocks, the TDF block employs dedicated embedded multipliers.

In this light, we further explore the trade-off between the low hardware cost of FDD designs and flexibility offered by TDD designs by examining a design consisting of an FDF block followed by a TDF block (designs 3 and 4 in Table 6.5.1). These designs provide limited tunable decimation factors compared to design 1, but also have lower hardware cost in terms of device utilization.

6.6 Summary

We have proposed a dataflow-based approach for prototyping radio astronomy DSP systems. We have used a dataflow-based high-level application model that provides a

Table 6.4: Implementation summary for designs employing two-stage downconversion using cascaded FDF or TDF blocks. B_w, if tunable, can be tuned to frequencies consistent with decimation factors supported by the TDD block.

Parameter	Design 1	Design 2	Design 3	Design 4
Mixer	No	No	No	No
Input bandwidth (MHz)	800	800	800	800
No. of FDD blocks	0	2	1	1
No. of TDD blocks	2	0	1	1
FDD Decimation factor(s)	—	8, 10	8	10
B_w (MHz)	Tunable (≤ 800)	10	Tunable (≤ 100)	Tunable (≤ 80)
Latency (ns)	170	475	120	505
FPGA slices (Out of 23616)	17141 (72%)	5765 (24%)	11073 (46%)	12641 (53%)
4 input LUTs (Out of 47232)	19718 (41%)	5506 (11%)	12245 (25%)	12310 (26%)
Block RAMs (Out of 232)	41 (17%)	41 (17%)	41 (17%)	41 (17%)
18×18 Multipliers (Out of 232)	64 (27%)	16 (6%)	40 (17%)	40 (17%)

platform-independent specification, and assistance in functional verification and important resource estimation tasks. This can prove effective in reducing the development cycle and faster deployment of DSP systems across various target platforms. We have employed this approach to methodically develop a TDD based DSP backend design. Our TDD implementation is targeted to the CASPER FPGA board, called IBOB, and supports tuning narrow band modes without the need for regenerating hardware code. We have also explored the trade-off between the low hardware cost for FDD designs and the flexibility offered by TDD designs. This trade-off has also been highlighted in the context of designs employing a two-stage downconversion scheme. A designer can explore this design space to best meet the application requirements.

Chapter 7

Summary and Conclusions

In this thesis, we have addressed various aspects of design flows employed by model based design tools for embedded systems in the context of rapid prototyping of high performance signal processing applications. We summarize these contributions along with our conclusions as follows:

1. We have introduced the concept of topological patterns, which can be used in dataflow modeling languages to identify and concisely iterate across arbitrary structures in a dataflow application graph. We have shown how the types of flowgraph substructures that are pervasive in the digital signal processing (DSP) application domain can be effectively represented in terms of topological patterns, and thereby used to generate compact, scalable application representations. We have also shown how an underlying design tool can exploit a high-level application specification consisting of topological patterns in various aspects of the design flow. In particular, we have demonstrated the efficacy of topological patterns in dataflow graph analysis, concise and scalable representation of homogeneous synchronous dataflow (HSDF) graphs, and exploring implementation-specific trade-offs. We have also demonstrated the use of topological patterns in graph analysis and extraction of implementation-specific features. We have applied the concept of topological patterns to represent schedules for application graphs. Such representations are use-

ful, for example, when porting schedules generated using one design tool to other platform-specific tools or design languages. We have demonstrated the utility of experimentation with pattern-specific scheduling transformations, and how topological patterns facilitate such experimentation.

2. We have formulated the core functional dataflow (CFDF) model of computation, which can be used to model a wide variety of deterministic dynamic dataflow behaviors, and used to capture various well known forms of dataflow in a single, unified formulation. We have also presented features of the CFDF model and tools based on it, such as support for 1) heterogeneous dataflow behaviors, 2) intuitive functional specification, 3) functional simulation that allows designers to model and verify interactions among various forms of dataflow, 4) portability from existing dataflow models, 5) minimally-restricted specification of actor functionality, and 6) efficient static, quasi-static, and dynamic scheduling techniques. With the CFDF modeling approach integrated into dataflow interchange format (DIF), we have demonstrated the use of CFDF concretely on various applications. Such an approach has allowed us to functionally simulate designs from early stages of design, and then focus on experimenting with schedules and dataflow transformations to improve performance.

3. We have presented a new scheduling technique for dynamic dataflow applications. This technique leverages the CFDF model, and operates by decomposing dynamic dataflow graphs into sets of dynamically interacting static dataflow graphs. We have demonstrated this scheduling technique on mixed-model applications with existing

schedulers, which has given a positive indication of the utility of the approach for software implementations of such dynamic dataflow applications. By identifying static groups of "modes" inside actors, we have exposed more of the static nature of applications, allowing traditional scheduling techniques to improve on memory requirements by up to 37%.

We have further used CFDF semantics to model a class of signal flow topologies that is important for modern communication systems. Our approach identifies the underlying static components in the application, systematically integrates well-established compile-time scheduling techniques for synchronous dataflow (SDF) graphs with more flexible CFDF semantics, and uses combined CFDF/SDF analysis to generate parameterized looped schedules (PLSs) that have significantly reduced run-time overhead, guaranteed memory bounds, and reduced memory requirements. Our approach therefore provides robust simulation of dynamic communication applications without major limitations on compile-time predictability and efficient scheduling.

4. We have demonstrated the use of a dataflow-based approach for prototyping radio astronomy DSP systems. We have used a dataflow-based high-level application specification format that provides a platform-independent specification, and assistance in functional verification and useful kinds of resource estimation. Such an approach is useful in improving designer productivity and facilitating faster deployment of DSP systems across various target platforms. We have employed this approach to methodically develop a tunable digital downconverter (TDD) based

DSP backend design. Our TDD implementation, which is targeted toward an FPGA board, called the interconnect break-out board (IBOB), from the collaboration for astronomical signal processing and electronics research (CASPER), supports tuning narrow band modes without the need for regenerating hardware code. We have also explored the trade-off between low hardware cost for fixed-configuration digital downconverter (FDD) designs and the flexibility offered by TDD designs. This trade-off has also been highlighted in the context of designs employing multistage downconversion schemes. Using our approach for trade-off exploration between hardware cost and flexibility, a designer can efficiently explore the associated design space to help optimize an implementation in terms of the given application requirements.

5. This thesis has contributed significantly to the development and release of the latest version of a graph package, called MoCGraph, that is oriented toward providing fundamental graphical data structures and implementations of graph algorithms to support analysis and manipulation of models of computation. Our contributions to this graph package include support for tree data structures, and generalized schedule trees (GSTs), in particular. Our extensions to the MoCGraph package have supported important features for the CFDF model, and for new functional simulation capabilities in the DIF package.

The work presented in this thesis, though demonstrated using specific design tools, is not restricted to those tools, and hence, can be applied to a wide variety of dataflow-based environments. Also, the applications presented, though instrumental in driving this

research, are demonstrative, and the prototyping methods can be extended readily to other relevant DSP applications.

Chapter 8

Future Work

We believe that pattern-specific scheduling techniques can provide improved scheduling capabilities for dataflow-based design environments that employ topological patterns. In our work on topological patterns in this thesis, we have emphasized the same by providing a facility that allows a user to experiment with various schedules in a systematic manner. The development of pattern-specific scheduling heuristics and support of those in design tools has been, however, beyond the scope of this thesis. Such exploration along with automating the application and integration of topological patterns are useful directions for further investigation.

For scheduling general *core functional dataflow* (*CFDF*) graphs, it would be useful to look into an approach that involves identifying static and dynamic components of heterogeneous dataflow graphs, generating two-actor schedules for clusters of two CFDF actors in the graphs, and merging those into nested schedules that are optimized for selected performance cost metrics based on probabilistic analysis of CFDF graphs. The motivation here is to build on the pairwise grouping methodology for dataflow graph clustering [12], which is a useful and flexible scheduling framework for synchronous dataflow graphs, and develop efficient simulation and synthesis capabilities for dynamic dataflow graphs. In this regard, it would be also useful to explore general CFDF topologies as targets for *parameterized looped schedule* (*PLS*) construction, and apply our methods to optimized

hardware and software synthesis.

Expanding on our work to integrate *tunable digital downconverter* (*TDD*) design with ongoing development of spectrometer designs at the National Radio Astronomy Observatory (NRAO) on the latest hardware from the Collaboration for Astronomical Signal Processing and Electronics Research (CASPER) group is a natural extension of the work presented in this thesis. There is growing interest in the radio astronomy community to have open-access and portable astronomical signal processing solutions. Currently, this is constrained by proprietary commercial tools targeted for specific platforms. We have also relied on these tools, mainly for hardware synthesis and code generation, in our work. In this context, it is of interest to have high-level application description languages with semantic foundations in models of computation and the corresponding design tools for efficient specification, simulation, functional verification, and synthesis. Developing model based platform-specific libraries, and devising techniques for automatic code generation from high-level representations, such as those in *dataflow interchange format* (*DIF*), specifically for the radio astronomy domain is an important direction for future research.

There have been some other directions that we have explored while working on the core aspects of this thesis. We have contributed to the development of a dataflow-based design tool, called *targeted DIF* (*TDIF*), which extends the capabilities of DIF with dynamic dataflow software synthesis, cross-platform actor design support, and dataflow-integrated features for instrumenting and tuning implementations [74]. The dataflow-based approach used in TDIF is unique in that it leverages the power of dynamic dataflow models, and provides integration of automatic code generation for programming inter-

faces and low level customizations for implementations targeted to heterogeneous platforms. Also, a new model based schedule representation called the *dataflow schedule graph* (*DSG*) representation has been recently introduced [84]. Integrating prototyping features and techniques presented in this thesis with these new tools and models is also an important direction for future work.

Bibliography

[1] *3GPP TS 36.211 V8.7.0 (2009-05): Physical channels and modulation*, 2009.

[2] S. Alliot and E. Deprettere. Architecture exploration of a large scale system. In *Proceedings of the IEEE International Workshop on Rapid System Prototyping*, pages 217–224, Geneva, Switzerland, June 2004.

[3] J. G. Andrews, A. Ghosh, and R. Muhamed. *Fundamentals of WiMAX: understanding broadband wireless networking*. Prentice Hall, 2007.

[4] J.W.M. Baars, L.R. D'Addario, and A.R. Thompson. Radio astronomy in the early twenty-first century. *Proceedings of the IEEE*, 97(8):1377 –1381, August 2009.

[5] G. Berry. Bottom-up computation of recursive programs. *ITA*, 10(1):47–82, 1976.

[6] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1948–1951, Istanbul, Turkey, June 2000.

[7] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.

[8] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, 2010.

[9] S. S. Bhattacharyya, S. Kedilaya, W. Plishker, N. Sane, C. Shen, and G. Zaki. The DSPCAD integrative command line environment: Introduction to DICE version 1. Technical Report UMIACS-TR-2009-13, Institute for Advanced Computer Studies, University of Maryland at College Park, August 2009.

[10] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, 47(9):849–875, September 2000.

[11] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[12] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, 2(1):33–60, January 1997.

[13] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[14] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26 –37, November 2009.

[15] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (Volume 3: Ptolemy II domains). Technical Report UCB/EECS-2008-37, EECS Department, University of California, Berkeley, April 2008.

[16] J. T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.

[17] C. Chang, J. Wawrzynek, and R. W. Brodersen. BEE2: a high-end reconfigurable computing system. *Design & Test of Computers, IEEE*, 22(2):114–125, March-April 2005.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.

[19] I. Corretjer, C. Hsu, and S. S. Bhattacharyya. Configuration and representation of large-scale dataflow graphs using the dataflow interchange format. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 10–15, Banff, Canada, October 2006.

[20] L. R. D'Addario and C. Timoc. Digital signal processing for the SKA: A strawman design. Technical Report SKA Memo No. 25, Square Kilometre Array, August 2002.

[21] J. Dalcolmo, R. Lauwereins, and M. Ade. Code generation of data dominated DSP applications for FPGA targets. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 162–167, June 1998.

[22] D.R. DeBoer, R.G. Gough, J.D. Bunton, T.J. Cornwell, R.J. Beresford, S. Johnston, I.J. Feain, A.E. Schinckel, C.A. Jackson, M.J. Kesteven, A. Chippendale, G.A. Hampson, J.D. O'Sullivan, S.G. Hay, C.E. Jacka, T.W. Sweetnam, M.C. Storey, L. Ball, and B.J. Boyle. Australian SKA pathfinder: A high-dynamic range wide-field of view survey telescope. *Proceedings of the IEEE*, 97(8):1507 –1521, 2009.

[23] P. E. Dewdney, P.J. Hall, R. T. Schilizzi, and T.J.L.W. Lazio. The square kilometre array. *Proceedings of the IEEE*, 97(8):1482 –1496, August 2009.

[24] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., second edition, 2000.

[25] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S.R. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):127–144, January 2003.

[26] J. Eker and J. W. Janneck. CAL language report, language version 1.0 — document edition 1. Technical Report UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley, December 2003.

[27] S.W. Ellingson, T.E. Clarke, A. Cohen, J. Craig, N.E. Kassim, Y. Pihlstrom, L.J. Rickard, and G.B. Taylor. The Long Wavelength Array. *Proceedings of the IEEE*, 97(8):1421 –1430, August 2009.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[29] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214 –230, March 1993.

[30] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[31] C. Harris, K. Haines, and L. Staveley-Smith. GPU accelerated radio astronomy signal convolution. *Experimental Astronomy*, 22:129–141, 2008. 10.1007/s10686-008-9114-9.

[32] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubhr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-based design methodology for digital signal processing systems. *EURASIP Journal on Embedded Systems*, 2007:Article ID 47580, 22 pages, 2007.

[33] S. Haykin. *Adaptive filter theory*. Prentice-Hall, Inc., 1996.

[34] Y. Hemaraj, M. Sen, R. Shekhar, and S. S. Bhattacharyya. Model-based mapping of image registration applications onto configurable hardware. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 1453–1457, Pacific Grove, California, October 2006. Invited paper.

[35] C Hsu, I. Corretjer, M. Ko., W. Plishker, and S. S. Bhattacharyya. Dataflow interchange format: Language reference for DIF language version 1.0, user's guide for DIF package version 1.0. Technical Report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2007. Also Computer Science Technical Report CS-TR-4871.

[36] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37–49, Dallas, Texas, September 2005.

[37] C. Hsu, S. Ramasubbu, M. Ko, J. L. Pino, and S. S. Bhattacharyya. Efficient simulation of critical synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*, pages 893–898, San Francisco, California, July 2006.

[38] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE*, pages 1098–1101, September 1952.

[39] G. Johnson. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*. McGraw-Hill School Education Group, 1997.

[40] J.L. Jonas. MeerKAT — the South African array with composite dishes and wide-band single pixel feeds. *Proceedings of the IEEE*, 97(8):1522 –1530, August 2009.

[41] B. Kienhuis and E. F. Deprettere. Modeling stream-based applications using the SBF model of computation. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 385–394, September 2001.

[42] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere. Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. *IEEE Transactions on Signal Processing*, 55(6):3126–3138, June 2007.

[43] S. Y. Kung. *VLSI Array processors*. Prentice Hall, 1988.

[44] S. Kwon, H. Jung, and S. Ha. H.264 decoder algorithm specification and simulation in Simulink and PeaCE. In *Proceedings of the International SoC Design Conference*, pages 9–12, October 2004.

[45] E. A. Lee. Recurrences, iteration, and conditionals in statically scheduled block diagram languages. In *Proceedings of the International Workshop on VLSI Signal Processing*, 1988.

[46] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan 1987.

[47] E. A. Lee and D. G. Messerschmitt. *Digital Communication*. Kluwer Academic Publishers, 1988.

[48] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773–799, May 1995.

[49] J. Lemaitre. *Model-based specification and design of large-scale embedded signal processing systems*. PhD thesis, Leiden University, The Netherlands, 2008.

[50] J. Lemaitre and E. Deprettere. FPGA implementation of a prototype hierarchical control network for Large-Scale signal processing applications. In *Proceedings of the International Euro-Par Conference*, Lecture Notes in Computer Science 4128, pages 1192–1203. Springer, Dresden, Germany, August 2006.

[51] The MathWorks Inc. *Using Simulink, Version 3*, January 1999.

[52] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice-Hall, Inc., 1989.

[53] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California, Berkeley, 1995.

[54] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cyclo-static dataflow. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, volume 1, pages 204–210 vol.1, Pacific Grove, California, October 1995.

[55] A. Parsons, D. Backer, C. Chang, D. Chapman, H. Chen, P. Droz, C. de Jesus, D. MacMahon, A. Siemion, D. Werthimer, and M. Wright. A new approach to radio astronomy signal processing. In *Proceedings of the General Assembly of the International Union of Radio Science*, October 2005.

[56] A. Parsons, D. Backer, Chen Chang, D. Chapman, H. Chen, P. Crescini, C. de Jesus, C. Dick, P. Droz, D. MacMahon, K. Meder, J. Mock, V. Nagpal, B. Nikolic, A. Parsa, B. Richards, A. Siemion, J. Wawrzynek, D. Werthimer, and M. Wright. PetaOp/Second FPGA signal processing for SETI and radio astronomy. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 2031 –2035, Pacific Grove, California, November 2006. Invited paper.

[57] A. Parsons, D. Chapman, and H. Chen. Xilinx system generator for DSP in the CASPER group. Technical Report CASPER Memo 11, Center for Astronomy Signal Processing and Electronic Research, University of California, Berkeley, January 2007.

[58] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling system for DSP applications. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 122–126 vol.1, Pacific Grove, California, November 1995.

[59] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for DSP using Ptolemy. *Journal of VLSI Signal Processing*, 9(1), January 1995.

[60] J. L. Pino and K. Kalbasi. Cosimulating synchronous DSP applications with analog RF circuits. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[61] W. Plishker. *Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors*. PhD thesis, University of California, Berkeley, January 2006.

[62] W. Plishker, N. Sane, and S. S. Bhattacharyya. A generalized scheduling approach for dynamic dataflow applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 111–116, Nice, France, April 2009.

[63] W. Plishker, N. Sane, and S. S. Bhattacharyya. Mode grouping for more effective generalized scheduling of dynamic dataflow applications. In *Proceedings of the Design Automation Conference*, pages 923–926, San Francisco, July 2009.

[64] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 17–23, Monterey, California, June 2008.

[65] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. *Transactions on High-Performance Embedded Architectures and Compilers*. Online version available from http://www.hipeac.net/node/3030, print version to appear.

[66] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, Samos, Greece, July 2008.

[67] W. Plishker, C. Shen, S. S. Bhattacharyya, G. Zaki, S. Kedilaya, N. Sane, K. Sudusinghe, T. Gregerson, J. Liu, and M. Schulte. Model-based DSP implementation on FPGAs. In *Proceedings of the International Symposium on Rapid System Prototyping*, Fairfax, Virginia, June 2010. Invited paper, DOI 10.1109/RSP_2010.SS4, 7 pages.

[68] R.M. Prestage, K.T. Constantikes, T.R. Hunter, L.J. King, R.J. Lacasse, F.J. Lockman, and R.D. Norrod. The Green Bank telescope. *Proceedings of the IEEE*, 97(8):1382 –1390, August 2009.

[69] S. Saha, S. Puthenpurayil, and S. S. Bhattacharyya. Dataflow transformations in high-level DSP system design. In *Proceedings of the International Symposium on System-on-Chip*, pages 131–136, Tampere, Finland, November 2006. Invited paper.

[70] N. Sane, C.-J. Hsu, J. L. Pino, and S. S. Bhattacharyya. Simulating dynamic communication systems using the core functional dataflow model. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1538–1541, Dallas, Texas, March 2010.

[71] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya. Scalable representation of dataflow graph structures using topological patterns. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 13–18, San Francisco Bay Area, USA, October 2010.

[72] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya. Topological patterns for scalable representation and analysis of dataflow graphs. *Journal of Signal Processing Systems - Special Issue on SIPS 2010*, December 2011. To appear.

[73] C. Shen, W. Plishker, S. S. Bhattacharyya, and N. Goldsman. An energy-driven design methodology for distributing DSP applications across wireless sensor networks. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 214–223, Tucson, Arizona, December 2007.

[74] C. Shen, H. Wu, N. Sane, W. Plishker, and S. S. Bhattacharyya. A design tool for efficient mapping of multimedia applications onto heterogeneous platforms. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, Barcelona, Spain, July 2011.

[75] J. J. Shynk. Frequency-domain and multirate adaptive filtering. *IEEE Signal Processing Magazine*, 9(1):14–37, January 1992.

[76] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009.

[77] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using Kahn process networks: the Compaan/Laura approach. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 340 – 345 Vol.1, February 2004.

[78] W. Sun, M. J. Wirthlin, and S. Neuendorffer. FPGA pipeline synthesis design exploration using module selection and resource sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):254–265, 2007.

[79] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, 2002.

[80] P. P. Vaidyanathan. Multirate digital filters, filter banks, polyphase networks, and applications: a tutorial. *Proceedings of the IEEE*, 78(1):56 –93, January 1990.

[81] R.V. van Nieuwpoort and J.W. Romein. Building correlators with many-core hardware. *IEEE Signal Processing Magazine*, 27(2):108 –117, March 2010.

[82] I. M. Verbauwhede, C. J. Scheers, and J. M. Rabaey. Specification and support for multidimensional DSP in the SILAGE language. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages II/473 –II/476, April 1994.

[83] G. K. Wallace. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii –xxxiv, February 1992.

[84] H. Wu, C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya. A model-based schedule representation for heterogeneous mapping of dataflow graphs. In *Proceedings of the International Heterogeneity in Computing Workshop*, pages 66–77, Anchorage, Alaska, May 2011.

[85] Xilinx Inc. *System Generator for DSP - User Guide, Release 10.1.1*, April 2008.