Model-based design of systems running software defined radios

Waqas, U.

Award date:
2012

Model-based design of systems running software defined radios

Master's Thesis

August 2012

Author:

Umar Waqas (0758262)

Supervisors:

dr. ir. Sander Stuijk (Eindhoven University of Technology)
ir. Peter Kourzanov (NXP Semiconductors)

EINDHOVEN UNIVERSITY OF TECHNOLOGY
NXP SEMICONDUCTORS

Abstract

Wireless communication has become an integral part of our everyday life. In order to meet the current requirements of wireless operators and technology providers, Software Defined Radios (SDRs) are used. SDR is a form of radio that performs the signal processing in software. The requirements include flexible design, upgrade and reuse of radios. In this thesis, we propose a model based design approach to develop SDRs. Specifically, we present constructs to model the digital baseband processing in an SDR. As a case study, we model a Digital Video Broadcasting Terrestrial (DVBT) decoder over a heterogeneous Multi Processor System-on-Chip (MPSoC) which is MARS.

A typical SDR may have data and reception dependent operations referred to as scenarios during its operation. These operations have varying resource requirements. We present constructs to model scenarios. Moreover, we identify the scenarios present in a DVBT decoder. SDRs usually have a high data rate that requires efficient implementation and dynamic memory management. In this thesis, we describe how to model one such implementation aspect, namely, packet pools. In a MPSoC, several masters accessing a shared slave may interfere with each other. In order to take this interference into account, we model the AXI based interconnect present in the MARS platform.

In this thesis, we propose a technique to reduce the complexity of an FSMSADF graph. This technique reduces the number of initial tokens and/or the number of actors present in an FSMSADF graph while preserving the timing behavior of the graph and decreasing the analysis time. For experimental evaluation of the constructs presented in this thesis, we developed a tracing framework and algorithms to compare traces generated from the models with the traces generated from the actual system. Collectively, these constructs contribute to the model based design of SDRs which in turn allows us to meet the requirements of wireless communication and technology providers.

Acknowledgments

This thesis describes the work performed in a graduation project conducted at NXP Semiconductors in approximately seven months. The graduation would have been very difficult without inspiration, guidance and support from several people. I would like to express my gratitude to them.

The foremost, I would like to mention Sander Stuijk, my supervisor at TU/e, who introduced me to this project at NXP Semiconductors. During the tenure of the project, he has always been a source of motivation and guidance. His keen supervision increased the quality of the work performed and urged me to find better solutions. He helped me to realistically plan the project activities and make practical decisions. It has been a privilege to work with him.

I would like to thank Peter Kourzanov, my supervisor at NXP Semiconductors for several lengthy but fruitful discussions. I appreciate his valuable feedback on the tools and techniques developed in the project. He motivated me to go beyond the horizon and explore things further.

I appreciate the guidance from Artur Burchard on understanding the architecture of the MARS platform. His explanations clarified several architectural concepts. I would like to thank Hong Li for several discussions on the operation of a DVBT decoder. He always answered my questions on an urgent basis providing detailed explanations. I appreciate the support from David Riemens for explaining concepts related to the MARS SDK. I appreciate his efforts for arranging a MARS board and for setting it up.

I appreciate the coffee break discussions with Luuk Loeffen. He always provided food for thoughts. I appreciate suggestions from Sunil John for the tracing framework designed in the project. I would like to thank him for providing valuable feedback on my presentations. I would like to thank Ruxandra Bobiti and Salman Shafqat for proof reading this thesis and for cooking for me in the busy times during the project. Finally, I would like to appreciate the support from my family and friends. Without these contributions, it would have been more difficult to achieve the current form of the project.

Contents

Chapter 1

Introduction

Over the last few decades, technology has influenced how we disseminate information. The invention of telegraph, telephone, radio and television has laid the foundations for modern day communications. With these advances in technology, communication got better, faster and more reliable. Modern day communication allows seamless connectivity, high transfer rates, reliable and secure transmission for both short range and distant communication.

Wired and wireless communication are two broad types of communication. In general, wired communication is faster and more reliable than wireless communication but facilitates fixed point communications only. Wireless communication is suitable for nomadic, mobile and distant communication. In the sequel, we refer to the devices that wirelessly communicate as *radios*. With the evolution of wireless communication, it gradually became an integral part of our everyday life. In the following section, significant inventions during the evolution of wireless communication are described. Section 1.2 presents the motivation behind this thesis. In Section 1.3 software defined radios are introduced. Section 1.4 introduces bus-based systems-on-chips. The contributions of this thesis are described in Section 1.6. In Section 1.7, the organization of this thesis is described.

1.1 Radio: the past, present and future

The *photophone* is considered as the first operational wireless communication device. Articulated sounds were transmitted using photophones between two points that were 200 meters apart. Increasing the range, improving the reliability and standardization were important objectives of the inventions succeeding the photophone. Figure 1.1 presents significant inventions during the evolution of wireless communication, focusing on scientific, commercial and public successes.

In 1927, it became possible to wirelessly communicate between the US and Britain. This marked a significant increase in the range of radio as well as it indicated the ease of access to the technology. In 1962 the placement of *Telstar* into orbit facilitated transatlantic reception of a television feed. From commercial to personal use, the radio was emerging and spreading across the globe as a key enabler of the kind of wireless communication

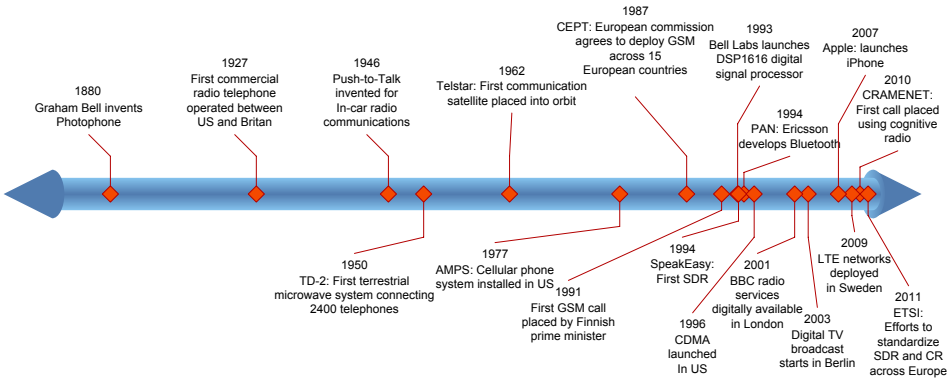

Figure 1.1: The evolution of radio over the last century.

man had ever dreamed of. The quality and coverage of wireless communication had a significant impact on the (commercial) success of radios. Advanced Mobile Phone System (APMS) was designed and deployed in US to provide better (compared to 0G systems) coverage and quality of wireless communication. GSM introduced digital circuit switching techniques in wireless communication that led to transmission of digital data along with articulated sounds. Eventually, in 1991, GSM became operational across Europe.

Wireless communication incorporates several signal processing techniques which make the connectivity possible. Different wireless communication standards perform the required signal processing differently. As the radio evolved; new standards emerged. It became infeasible to design a new integrated circuit to perform signal processing for every emerging standard. The *SpeakEasy* was the first radio in which the signal processing was performed in software (described in Section 1.3). The British Broadcast Company (BBC) started digital radio transmission in London in 2001. Similarly, digital TV transmission started in Berlin in 2003. Digital processing techniques were now an integral part of many wireless communication standards. *Cognitive radio*, a type of radio that is intelligent, is an active research topic in the field of wireless communication. A cognitive radio considers the user behavior and its environment during its operation. For example, it is able to sense the interference, the available standards and the carrier state in order to communicate. Software defined radios (SDRs) are key enablers of cognitive radios. With the standardization of cognitive and software defined radios, the wireless communication will enter an era where the radios are intelligent and efficient.

1.2 Motivation

Wireless operators and the technology providers need to cope with an increasing demands for high data rates and enhanced quality of communication while reducing the cost of consumer products. Innovations and improvements in the technology increase the quality of wireless communication resulting in new wireless communication standards [AAG+11]. SDR brings in the flexibility to upgrade a device that implements an existing standard and the addition of new standards to a device, by downloading improved software to the device. Thus SDR enables an existing device to upgrade to a new standard (by reconfiguration), where in case of a complete hardware implementation, it would have been required to redesign the hardware.

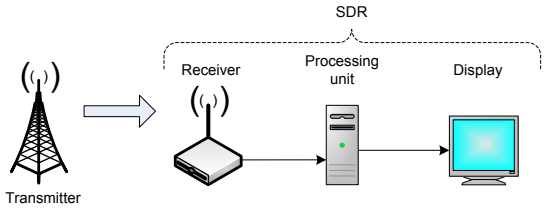

Figure 1.2: An example of an SDR.

Model based design allows to evaluate the design choices without implementing the system (completely). Model based design of SDRs will facilitate designers to analyze implementations of SDRs during the early design phases. Moreover, for existing systems, model based design allows to identify potential improvements by early evaluation of design decisions. For example, it allows to assess the feasibility of adding more applications to the system, analyze the resource utilization and identification of bottlenecks present in the system. These improvements either reduce the system cost or allow effective use of the existing resources contributing to fulfillment of the requirements of the wireless operators and the technology providers. In this thesis, we provide constructs to model SDRs over bus based system on chips thus contributing to model based design of SDRs.

1.3 Software defined radios

Software defined radio (SDR) is a type of wireless communication that implements all or part of the signal processing techniques in software. Figure 1.2 presents an example of an SDR based digital television (TV). The *transmitter* broadcasts the TV signal which is received by the *receiver*. The receiver digitizes the signal and passes it to the *processing unit*. The *processing unit* performs *digital baseband processing* which generally consists of 1) *filtration* 2) *(de)modulation* and *(de)coding*. As a result of the digital baseband processing, the TV feed is extracted and displayed.

Figure 1.2 is a primitive example of an SDR. An SDR based system usually has 3 components: 1) an analog-digital front end, 2) a processing unit and 3) an input-output subsystem. The front end serves as an interface facilitating wireless communication. The processing unit performs the signal processing. The type of input-output device depends on the application for which the SDR is designed. For instance, in case of a visual feed, the output device is a display terminal. In case of an audio feed the output device is typically a speaker. However, the input-output subsystem can be any other application specific system e.g a fax machine. Moreover, according to the application requirements, the number of front ends, processing units and the input-output subsystems may vary. In the sequel we refer to this component based model of SDRs as the *conceptual model of SDRs*.

Figure 1.3 illustrates the layered processing in a typical SDR [BHM+05]. The first layer, *radio frequency - intermediate frequency* (RF-IF) is a front end. The incoming signal is converted to a stream of digitized data by the *analog-to-digital converter* (ADC). The digital stream is passed to the digital baseband processing layers. The *filters* usually remove unwanted frequencies and suppress noise present in the signal. The filtered data is then demodulated by the *modem*. Subsequently, the data stream is decoded by the *codec* and handed over to the application for further processing. In case of a transmission, the sequence starts in the reverse order from the application layer towards

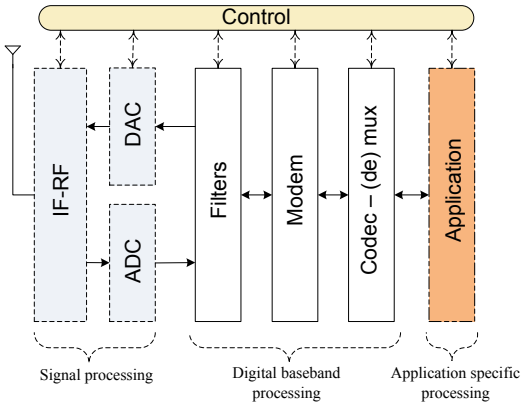

Figure 1.3: The architecture of a crude SDR [BHM+05].

the transceiver as is also illustrated in Figure 1.3.

1.3.1 DVBT decoder

Digital Video Broadcasting Terrestrial (DVBT) [DVBa] is an international standard for broadcasting of digital television feeds. Since its inception, DVBT has become one of the most widely accepted digital video broadcasting standards [DVBb]. In this section we introduce the operation of a typical DVBT decoder. However, for theoretical details about the DVBT standard, we refer the reader to [DVBa].

Figure 1.4 presents the block diagram of a typical DVBT decoder. The incoming signal, that is received by the signal processing component (dashed blocks), is processed by the *Automatic Gain Control* (AGC) that adjusts the signal (e.g amplitude) according to the signal strength. This adjusted signal is then digitized by the *Analog-to-Digital Converter* (ADC). The digitized signal is input to the digital baseband processing (non-dashed blocks). *FFT synchronization* is the first operation performed in the digital baseband processing layer of a typical DVBT decoder. Based on the input, this operation computes the transmitter characteristics, for example, size of the FFT window, fractional part of the Carrier Frequency Offset (CFO) (i.e. the crystal clock difference between the transmitter and receiver) and the OFDM symbol timing. The input data along with the estimated characteristics is used to perform a *Fast Fourier Transform* (FFT) that converts the input signal from the time domain to the frequency domain. Using the converted signal, the *CFO* operation computes the integer part of the CFO. A DVBT transmitter can operate in many transmission modes [DVBa]. Moreover, due to the possible existence of multiple paths of the signal traverses between the transmitter and receiver, and the mobility of the transmitter or receiver, the so called *Doppler compensation* and *channel equalization* needs to be performed. Collectively, these operations are referred to as *channel estimation*.

In order to extract the modulation and channel coding configuration, Transmission Parameter Signaling (TPS) bits are encoded by the transmitter in the OFDM symbols. The *TPS decoding* operation decodes these TPS bits. In an OFDM symbol, the data carriers contain the data bits that are mapped using Quadrature Amplitude Modulation (QAM) or Quadrature Phase-Shift Keying (QPSK). The QAM/QPSK demapping operation demodulates these data bits which are interleaved by the transmitter in order to support *long burst error correction*. The *Inner deinterleaver* operation deinterleaves

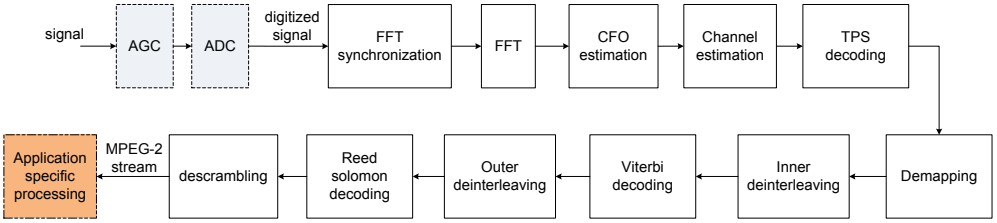

Figure 1.4: Block diagram of a typical DVBT decoder [YWC, DVBa].

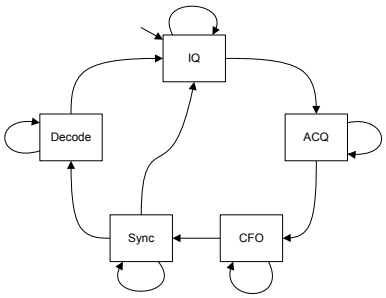

Figure 1.5: FSM representing scenarios present in a typical DVBT decoder.

the data bits that are convolutionaly decoded using *Viterbi decoding*. The bit stream obtained after Viterbi decoding is rearranged as bytes by the *Outer deinterleaver*. This byte stream is further processed by the *Reed Solomon decoding* and subsequently *Descrambled*. Finally, the transport stream (multiplexed MPEG-2 stream) is handed over to the application for application specific processing.

The DVBT decoder implemented on the MARS platform operates in five scenarios during its execution[1]. Figure 1.5 presents these scenarios along with their transitions. Initially, the decoder is in the *IQ* scenario. In this scenario the decoder estimates the so called In-phase and Quadrature-phase (IQ) imbalance to obtain the channel response. This estimation is performed by the FFT synchronization block. Note that in each scenario, the DVBT decoder may require many OFDM symbols (depending on the carrier state) to perform its computation and switch to the next scenario. Once the IQ balance is estimated, the DVBT decoder switches to the *ACQ* scenario. In this scenario, the Acquisition (ACQ) is performed to find the *OFDM symbol boundary*, the *FFT window size* used by the transmitter, and the *cyclic prefix length*. ACQ is performed by the FFT synchronization block. Once all parameters are found, the DVBT decoder switches to the next scenario i.e. CFO. In this scenario, the Carrier Frequency Offset (CFO) is computed by the CFO block. Once CFO is estimated, the DVBT decoder switches to the *SYNC* scenario. In this scenario, the DVBT decoder estimates the so called time tracking parameter e.g. common phase estimation, frequency tracking etc. This estimation is performed by the channel estimation and TPS decoding blocks. Once the parameter estimation is complete, the DVBT decoder switches to the decode scenario in which the demapping and subsequent blocks start decoding the incoming MPEG stream.

[1]The terms 'scenarios' and 'operating modes' are sometimes used interchangeably. However, we reserve the term 'operating mode' exclusively for the set of transmission modes used in the DVBT transceiver. The term 'scenario' is used to refer to the distinct execution behaviors when executing in a particular operating mode.

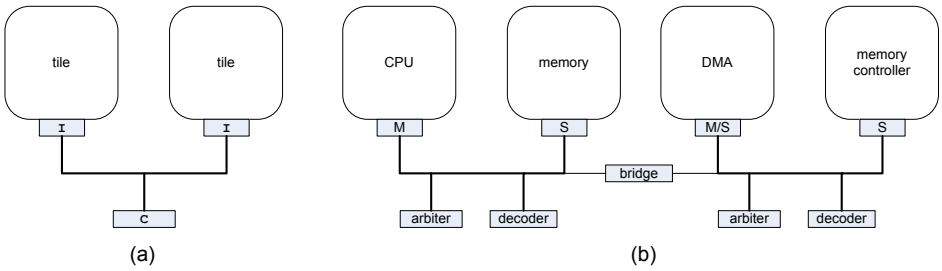

Figure 1.6: An example of a bus based SoC.

1.4 Bus based systems-on-chip

A *bus* is a group of wires used to transfer data between several components present in a system. Due to its simplicity and low cost, bus based communication is one of the most widely used communication architecture in embedded systems.

Figure 1.6 (a) presents a bus based Multi Processor System-on-Chip (MPSoC) template. This template consists of 3 components, a *tile*, an *interface* (denoted as I) and a *control component* (denoted by C). The interface and the control components facilitate connectivity between tiles. Figure 1.6 (b) provides an example of a SoC based on the template. It consists of 4 tiles. The CPU tile consists of a central processing unit. The CPU tile is connected to the bus through the master interface (M). A *master* is an interface that initiates the transfers over the bus. On the other hand, a *slave* interface (S) only responds to the incoming requests and cannot initiate transfers. For example, the *memory* tile is connected to the bus through the slave interface. A hybrid interface is a component that can act both as a master or as a slave. For example, the *DMA* tile is connected through a hybrid interface (M/S). Moreover, a *bridge* interface component is used to connect two buses. It acts as a hybrid interface connected on each bus. An *arbiter* is a control component that decides who gets access to the bus to initiate transfers. A *decoder* is a control component responsible to redirect the transfers to its intended destination.

1.4.1 Architecture of the MARS platform

Multi Application Radio System (MARS) [MAR] is a bus based multiprocessor system-on-chip (MPSoC). Figure 1.7 presents the hardware architecture of the MARS platform. It consists of a digital front-end (DFE) that receives the data provided by an external source (transmitter). In the MARS platform, there is an uplink that is capable to transmit the data to an external receiver. Both the DFE and the uplink facilitate the front end of the conceptual model of the SDRs (see Figure 1.3).

There are two vector digital signal processors (VDSPs) present in the MARS platform. These processors facilitate the *filtration* and the *modem* layers of the digital baseband processing (presented in Section 1.3). Moreover, these processors have two local memories (D0 and D1). The MARS platform contains a hardware accelerator i.e. FLORA to accelerate the *decoding* layer of the digital baseband processing. The ARM processor present in the MARS platform is a general purpose processor that configures the VDSPs and the FLORA accelerator. The ARM processor, the VDSPs and the FLORA accelerator constitute the processing unit of the conceptual model of SDRs.

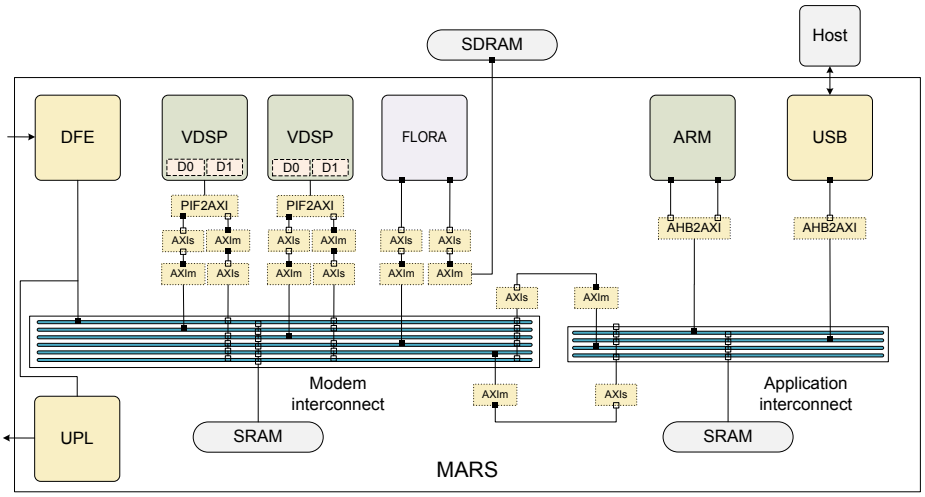

Figure 1.7: The architecture of the MARS platform.

There are two static random access memories (SRAMs) present in the MARS platform that are used to store program instructions and data. Moreover, the MARS platform also connects to an external synchronous dynamic random access memory (SDRAM). The output of the MARS platform, i.e. a processed signal, is accessed by the host system through the USB interface. In terms of the conceptual model of SDRs, the USB interface acts as an interface to the input-output subsystem.

The components described so far were tiles. These tiles are connected using several protocols from the Advanced Microcontroller Bus Architecture (AMBA) protocol suite and the Processor InterFace (PIF) bus. A bus based interconnect groups several buses together and provides standard interfaces to them. The MARS platform contains 2 64-bit Advanced Extensible Interface (AXI) bus based interconnects from the AMBA 3.0 protocol suite. One interconnect is responsible to transfer data across the modem subsystem that comprises of the DFE, uplink, VDSPs, SRAM and the FLORA tiles. There are 6 buses present in the modem interconnect. The VDSPs have PIF based interfaces that connect to the modem interconnect using AXI2PIF and AXI master-slave converters. The application subsystem consists of ARM, SRAM and USB tiles. The application interconnect consists of 4 buses. Both ARM and USB tiles are connected through the AHB2AXI converters. The application and the modem interconnect are connected with each other using an AXI bridge that comprises of 2 pairs of AXI master-slave converters.

There is only a single master present on a bus present in the application or the modem interconnect. The master ports connected to the bus are depicted as black squares where the slave ports are depicted as white squares. Possibly, a slave could be connected to several buses. For example, the SRAM slave present in the modem subsystem is connected to 5 buses which in turn requires arbitration. An AXI interconnect contains address decoder and arbiters (one for every shared slave).

The components present in the platform influence the behavior of an application running on the platform. For example, the communication between VDSP1 and SRAM (in the modem subsystem) will interfere with the communication of VDSP2 with the same SRAM. In order to consider this interference in the timing analysis, the communication over the interconnect needs to be modeled explicitly. We model the communication over

the AXI interconnect in this thesis as described in Section 4.2.

1.5 Model based design and challenges in designing SDRs

Early evaluation of the choices made during the design of SDRs is a non-trivial task. We present several such choices and motivate how model based design facilitates early evaluation of these choices. Furthermore, we exemplify the challenges in implementing a DVBT decoder over the MARS platform.

System complexity and abstraction. SDRs usually consist of several functional blocks or components interacting with each other. SDRs are complex due to this interaction. During early design phases, these interactions are usually not completely known and require abstraction. Moreover, as the design proceeds, the abstractions are elaborated allowing to incrementally decide on these design choices. Model based design facilitates such an incremental approach by modeling relevant aspects of the system and abstracting from irrelevant details. We exemplify such aspects present in the DVBT decoder in the case study presented in Chapter 7.

Splitting computation. The functional blocks present in an SDR are implemented as tasks. There exist several choices when distributing (splitting) the computation present in the blocks over these tasks. For example, for the DVBT decoder, one possible split is to have a task implementing the computation performed in a functional block or another choice is to split the computation present in a functional block across multiple tasks. Implementing each design choice is a time consuming and error prone process. Due to the abstraction, model based design reduces the time consumed in evaluation of these splits.

Communication choices. The split of computation influences the communication overhead which in turn affects the timing behavior. For example, tasks mapped on different processing units communicate through the interconnect and interfere with each other. This interference must be considered while taking these design choices. We present, in Chapter 4, constructs to model the AXI interconnect.

Optimizations. There are several optimizations possible while implementing SDRs. If known at an early design phase, these optimizations must be taken into consideration. One such optimization for the DVBT decoder is as follows.

The input to a DVBT decoder is an OFDM symbol. The size of each OFDM symbol depends on the mode of transmission. For example, for 8K, 64QAM, 1/4 guard rate mode the OFDM symbol size is 10240 bytes. For the same mode, approximately 872 symbols are received per second. Moreover, the input OFDM symbol is accessed by many blocks. For example, all blocks till the demapping block in the decoder chain (see Figure 1.4) access the input OFDM symbol as well as the outputs of the predecessor blocks in order to generate their outputs. As soon as the demapping has been performed, the OFDM symbol is no longer required. The large size of the OFDM symbol and high data rate make it challenging to implement a DVBT decoder as an embedded system and appeals for efficient (space and time) memory management.

The incoming OFDM symbols are analogous to packets of data. A *packet pool* is a collection of preallocated packets that are used to fulfill the memory requirements of an application dynamically. Many embedded kernels like OpenComRTOS [OPE], Nexos [BR09] and SoD [BB04] offer packet pools. Packet pools fall into the category of *region based memory management* [TT97]. In order to reduce the overhead of dynamic memory

management, region based memory management requires that the packets are released (freed) in the same order as they are acquired. This invariant prohibits the possibility of producing so called *holes* in the memory which in turn requires *defragmentation* for efficient memory management. This assumption typically holds in a DVBT decoder implementation as the OFDM symbols are processed in the digital baseband processing layer in FIFO order. Moreover, the overhead of copying data from block to block is usually reduced by incorporating *copy by reference* i.e. sending the address of the packet instead of copying data. Hence the memory requirements of a DVBT decoder can be fulfilled using packet pools over low cost of dynamic memory management.

Model validation. Model validation assesses how near the model is to the actual system. Furthermore, it identifies improvements in a model. In order to validate a model with its implementation, execution of the system and the model needs to be compared. Implementing such a comparison is non-trivial as it requires to record the execution of the system. Due to scarcity of resources in embedded platforms, implementing such a recording is difficult (see Section 6.2). In Chapter 6 we present a tracing framework that facilitates model validation.

In this thesis, we demonstrate model based design focusing to model the aspects presented above. Moreover, Chapter 7 presents a case study that evaluates the model based design approach presented in this thesis on a DVBT decoder which is implemented on the MARS platform.

1.6 Contributions

We contribute the following to the model based design of SDRs:

- Constructs to model digital baseband processing in software defined radios over bus based MPSoCs.
- Constructs to model AXI interconnect in the presence of shared slaves.
- Constructs to model packet pools.
- Implementation of a tracing framework to trace an SDR application running on the MARS platform.
- An evaluation on the accuracy of an SDR modeled with the FSMSADF MoC compared to an actual implementation of the application on the MARS platform.
- A technique to reduce the number of initial tokens and actors present in an FSMSADF graph.

1.7 Report overview

An introduction to the evolution of the wireless communication, SDRs and bus based MPSoCs is presented in this chapter. In Chapter 2 the problem addressed in this thesis is described. Chapter 3 presents the preliminaries to the model of computations used in this thesis. In Chapter 4, constructs to model digital baseband processing in software defined radios is presented. Chapter 5 presents a technique to reduce FSMSADF graphs. Chapter 6 presents the tracing framework. In Chapter 7 the constructs presented in this thesis are evaluated by modeling a DVBT decoder implementation over the MARS platform. Finally, Chapter 8 concludes this thesis and proposes future work.

Chapter 2

Overview of the Design Approach

Modern day wireless communication aims to provide high data rates to realize enhanced quality while reducing the cost of consumer products. As described in Section 1.2, it is desired to increase the quality of communication while reducing the cost and time to market of wireless consumer products. Model based design allows to analyze these products. Section 1.5 describes challenges faced during design of SDRs. In this chapter, the problem addressed in this thesis is described in more detail. Section 2.1 lists the goals of this thesis that contribute to a model based design approach for SDRs. Section 2.2 identifies the challenges involved in achieving these goals. Finally, Section 2.3 concludes this chapter.

2.1 Goals

Model based design steers the design process. It starts with *modeling the application* i.e. its functionality, *modeling the underlying* platform on which the application runs and *modeling the mapping* of the application to the resources offered by the platform. The results obtained from the timing analysis performed on the model facilitate making design choices. In this thesis we contribute to model based design of SDRs as follows:

- Provide constructs to model and analyze bus based SoCs. These constructs must model the components present in a SoC which influence the behavior of an application running on the SoC. These constructs must model the processing as well as the communication in the SoC.
- Model the mapping of an SDR on the resources offered by a bus-based SoC. The analysis must capture the associated trade offs which occur in different mappings.
- Provide constructs to model and analyze SDRs using measures like throughput, latency and operation in bounded memory. The modeling constructs must allow to capture the (relevant) behavior and implementation details of SDRs.

- Verify the outcomes of the modeling and analysis by comparing the system traces to the model traces.

The models, analysis, mapping and trace comparison should be performed automatically. This will require using an analysis toolkit, possibly extending it and implementing trace extraction on the platform. Collectively, these goals will contribute to the model based design of SDRs.

2.2 Challenges

Modeling SDRs and platforms which facilitate implementation of SDRs have several challenges. Specifically, the following challenges are identified:

- The digital baseband processing layer present in an SDR typically has data and carrier state dependent behavior leading to several scenarios. For example initialization, carrier parameter extraction and decoding. Moreover, the computation and hence the time required to perform the computation in each scenario differs. This difference must be considered during the analysis of the temporal properties e.g throughput. One may suggest to only consider the worst case execution scenario during the analysis which turns out to be pessimistic. Explicitly considering the scenarios will reduce this pessimism.
- Due to the high data rate and large packet sizes, the implementation of the digital baseband processing layer is usually optimized. One such optimization is the use of packet pools (see section 1.3.1). There are 3 properties of packet pools that influence the behavior of the implementation. These are 1) the size of the packet pool i.e. the number of packets present in it 2) the location where a packet pool resides i.e. local or remote memory in a SoC and 3) the amount of bytes accessed from each packet. Modeling these properties is a challenging task as the packet pool might be shared among many tasks and may be located on a shared memory. Moreover, mapping of packet pools to one of the available memories may have several trade offs. For example, mapping a packet pool to a remote memory (accessed over the bridge) will incur more latency than accessing the memory directly (without using the bridge).
- In a bus based SoC, simultaneous communication between pairs of tiles may cause interference. Modeling this interference requires modeling of the components present in the interconnect which is a challenging task. In order to model this interference in the MARS platform, the AXI based interconnect and the converters need to be modeled. For example, the packet pool might reside on a shared memory and two tiles accessing (same or different) packets from a packet pool will cause interference. In addition, the interconnect model must capture the trade offs present in accessing remote versus local memories.
- Validation of the designed model is a must. The model must be compared to the implementation in order to access the precision of the model[1]. This might require extensions to the analysis toolkit and to the SDR itself. For example, for

[1]The mentioned approach requires implementation of the system to be available (which is usually not available in early design phases). Model based design does not require the implementation to be available. However, in presence of such a implementation, it is desired to evaluate the model with the implementation

the DVBT decoder and the MARS platform it is required to record the events occurring in the system in order to compare them with the events present in the model. This recording and comparison will allow validation and be performed by development of a tracing framework. Efficient (space and time) design of such a tracing framework for an embedded system is a challenging task.

2.3 Conclusion

The problem addressed in this thesis was described in this chapter. The first problem addressed is to provide constructs to model SDRs. There are several challenges present in providing such constructs like modeling the scenarios and the memory managers used in the platform. Moreover, the communication architecture needs to be modeled in order to analyze the communication overhead. Finally, once the models are developed; their accuracy must be verified.

Chapter 3

Dataflow Preliminaries

The dataflow model of computation is often used to model digital baseband processing and other types of streaming applications [Gei09, SGB06, Stu07, SGM+11]. A comparison of dataflow MoCs is described in [PRE] and is not repeated in this thesis. In this chapter, we formally introduce two dataflow model of computations, namely, *Synchronous DataFlow* (SDF) [LM87], which is introduced in Section 3.1 and *Finite State Machine based Scenario Aware DataFlow* (FSM-SADF) [SGTB11], which is introduced in Section 3.2.

3.1 Synchronous DataFlow Graphs

SDF is a dataflow model of computation [LM87]. It consists of actors connected through dependency edges. The actors models atomic execution of computation. The edges model data or control dependencies between actors. Let the set A denote the set of actors present in the SDF graph. Let the set D denote the set of dependencies between the actors in the SDF graph. Let $\mathbb{N}$ denote the set of natural numbers and $\mathbb{N}_0$ denote the set of natural numbers including 0. For illustration, we use the example SDF graph shown in Figure 3.1.

There are three actors present in Figure 3.1(a), namely, a, b and c. An actor is represented as a *circle* with its name inside the circle. For the example SDF graph the set of actors $A = \{a, b, c\}$.

Definition 3.1: (DEPENDENCY EDGE) A dependency edge $d \in D$ is a tuple ($srcActor$, $destActor$, $srcRate$, $dstRate$, $initialTokens$) where $srcActor \in A$ is the source actor of d denoted by $srcActor(d)$, $dstActor \in A$ is the destination actor of d denoted by $dstActor(d)$, $srcRate \in \mathbb{N}$ is the number of data items (tokens) produced by the $srcActor(d)$ after an execution, $dstRate \in \mathbb{N}$ is the number of data items (tokens) required by the $dstActor(d)$ in order to execute and $initialTokens \in \mathbb{N}_0$ is the number of tokens denoted by $initialTokens(d)$ initially present on d. ◇

There are four dependencies present in the SDF graph specified in Figure 3.1 (a). A dependency is pictorially represented as a *directed edge* from a source actor to a des-

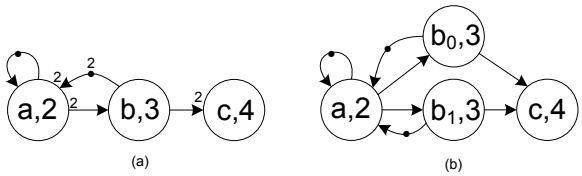

Figure 3.1: An example of an SDF graph (a) and its HSDF graph (b).

tination actor. For example, actor c is dependent on actor b which is denoted by a directed edge from b to c. Actor a produces two tokens, on the dependency edge from a to b upon completion of its execution (source rate) which is denoted by the number on the start of the edge. Actor c requires one token in order to fire (destination rate). Initial tokens are presented as a *dot* on a directed edge. For example, there are two initial tokens present on the directed edge from b to a. Notice that rates and initial token counts equal to one are omitted for clarity and we follow this convention in the sequel. The set of dependencies D for the example SDF graph is $D = \{(a, a, 1, 1, 1), (a, b, 2, 1, 0), (b, a, 1, 2, 2), (b, c, 6, 1, 0)\}$. Each actor in an SDF graph has an execution time written next to its name (after a comma inside the circle). For example, in Figure 3.1 (a) , actor a has execution time of two time units, b has three time units and c has 4 time units.

Definition 3.2: (SDF) A synchronous dataflow (SDF) graph G is tuple (A,D,Υ) where A is the set of actors present in the SDF, D is the set of dependencies between actors present in the SDF graph and $\Upsilon : A \rightarrow \mathbb{N}_0$ is a function that determines the execution time $e \in \mathbb{N}_0$ for a given actor $a \in A$. ◇

Definition 3.3: (HSDF) A homogeneous synchronous dataflow (HSDF) graph is an SDF graph (A,D,Υ) in which all dependencies $d \in D$ have unit source and destination rates i.e. $srcRate(d) = dstRate(d) = 1$. ◇

Figure 3.1(b) shows an equivalent HSDF graph [LM87] for the SDF graph shown in Figure 3.1(a). This graph has been obtained using the conversion algorithm presented in [LM87]. It consists of four actors, namely, a, b_0, b_1 and c. There are 7 dependencies present in the HSDF graph. As specified in Definition 3.3, the source and destination rates of the dependencies are one. Algorithms to convert an SDF graph to an equivalent HSDF graph are presented in [SB09] and [Gei09]. We discuss the pros and cons of each algorithm in detail in the following chapter. As compared to the SDF shown in Figure 3.1(a), the number of actors and dependencies has increased in the HSDF. This increase is exponential in worst-case.

Definition 3.4: (REPETITION VECTOR AND CONSISTENCY) A repetition vector γ of an SDF graph $G(A, D)$ is a function $\gamma : A \rightarrow \mathbb{N}_0$ such that for every dependency $d = (s, d, p, q, n) \in D$ it holds that $\gamma(s) \times p = \gamma(q) \times q$. An SDF graph G is consistent if it holds that $\forall a \in A\ \gamma(a) > 0$. ◇

The repetition vector for the actors [a,b,c] present in the SDF graph shown in Figure 3.1(a) is $[1, 2, 1]$. A non-consistent SDF graph is not useful in practice as it either deadlocks (due to insufficient tokens in a cyclic dependency) or it needs an infinite amount of memory. In this thesis, we verify that every proposed SDF graph is consistent.

Definition 3.5: (ITERATION) An iteration I of an SDF graph $G(A, D)$ is a collection of actor executions of the SDF graph in which an actor $a \in A$ fires exactly $\gamma(a)$ times.◇

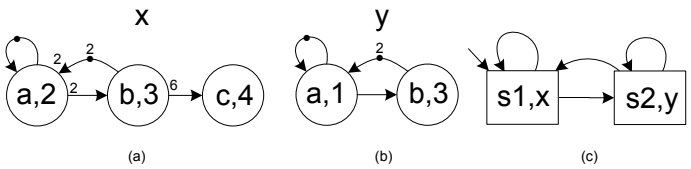

Figure 3.2: An example FSMSADF graph with SDF graphs (a) and (b) corresponding to each scenario and an FSM (c).

Intuitively, the repetition vector specifies when an *iteration* of the graph is completed. An iteration for the example SDF graph completes when the actor a fires one time, actor b fires two times and actor c fires one time.

3.2 Finite State Machine based Scenario Aware DataFlow

The Finite State Machine based Scenario Aware Dataflow (FSMSADF) [SGTB11] is a dataflow model of commutation. Figure 3.2 presents an example FSMSADF. It consists of two scenarios x and y. Each scenario in an FSMSADF graph contains a corresponding SDF graph. The SDF graphs for the scenarios x and y are shown in Figure 3.2 (a) and (b) respectively. Moreover, the transitions between the scenarios are specified in the FSM Figure 3.2(c). In many cases the name of a scenario and the name of a corresponding state is same. Due to this similarity, the scenario name may not be shown in the FSM. In the sequel this convention is followed.

Definition 3.6: (SCENARIO FSM) A scenario FSM F on a set of scenarios S is a tuple (Q,q_0,δ,Σ) where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times Q$ is a transition relation between the states present in the FSM and $\Sigma : Q \rightarrow S$ is a function that maps a state to its corresponding scenario. $\diamond$

The FSM specified in Figure 3.2 consists of a set of states $Q = \{s1, s2\}$ with initial state $q_0 = s1$. There are four transitions present in the FSM. The transition relation $\delta = \{(s1, s1), (s1, s2), (s2, s1), (s2, s2)\}$. The function Σ maps $s1 \rightarrow x$ and $s2 \rightarrow y$.

Definition 3.7: (FSMSADF) An FSMSADF graph G_F is a tuple (S, F) where S is set of SDF graphs corresponding to each scenario and F is a scenario FSM. $\diamond$

Figure 3.2 presents an FSMSADF with 2 scenarios, namely, x and y and a scenario FSM with two states $s1$ and $s2$. The SDF graphs corresponding to scenario x and y are specified in Figure 3.2 (a) and (b) respectively. Note that the number of initial tokens present in the scenario graphs must be equal. In the sequel we only present the tokens required to model an application. However, the number of initial tokens in the SDF graphs is made same as follows. First the maximum number of initial tokens present in the SDF graphs in an FSMSADF is computed. Secondly, to each SDF graph the difference between the number of initial tokens in the SDF graph and the maximum computed in the first step is computed. Finally, initial tokens, equal to the difference computed in the second step, are added to a self edge on a dummy actor. This dummy actor has an execution time equal to 0 and is connected to an arbitrary actor.

3.3 Conclusion

An introduction to SDF and FSMSADF MoCs is presented in this chapter. The actors and dependencies between them are formally defined in this chapter. In Chapter 4 the streaming tasks present in the digital baseband processing layer are modeled as actors and data passing across different functional blocks is modeled as dependencies. The FSMSADF MoC facilitates modeling of dynamism present in an SDR. In Chapter 7, the dynamism present in the DVBT decoder is modeled using the FSMSADF MoC.

Chapter 4

Modeling Software Defined Radios

Software defined radios (SDRs) were introduced in Section 1.3. This chapter describes constructs to model SDRs. Section 4.1 describes generic constructs to model the computation present in the digital baseband processing layer of an SDR. Once the computation is modeled, usually in the next design phase, the communication is taken into account. Section 4.2 presents constructs to model an AXI-based interconnect. The FSMSADF MoC facilitates *scenarios* which model the dynamism present in an SDR. The statespace of an FSMSADF graph [SGTB11] can be large leading to long run times of the analysis algorithms. In Section 4.3 we present a technique to limit the statespace of an FSMSADF graph (which may potentially introduce inaccuracies in the analysis result). Section 4.4 discusses the work related to modeling SDRs. Finally, Section 4.5 concludes this chapter.

4.1 Modeling digital baseband processing

The digital baseband processing layer is implemented as a set of streaming tasks that process data in a chained fashion. We model each task present in the system as an SDF actor. The streamed operation of these tasks give rise to data dependencies. The tasks and dependencies in the implementation of the DVBT decoder are presented in Chapter 7. In this section we present generic constructs to model operations usually performed in SDRs.

Modeling a periodic source and sink. The incoming signal is periodically broadcasted by a transmitter in the form of a symbol. This symbol is received by the front end. In Figure 4.1, a construct to model a periodic arrival of symbols is shown. The actor *Source* models the periodic arrival of incoming data. Each firing of this actor produces a symbol (i.e. a token in SDF terminology). The execution time of this actor is equal to the period of the incoming symbol. Every input symbol is processed by the digital baseband processing layer of an SDR (shown as a block in Figure 4.1). The signal processing layer processes the input data and the output is handed over to the application for application specific processing (see Section 1.3 for details of layered processing).

Figure 4.1: Modeling periodic source and sink typically present in SDRs.

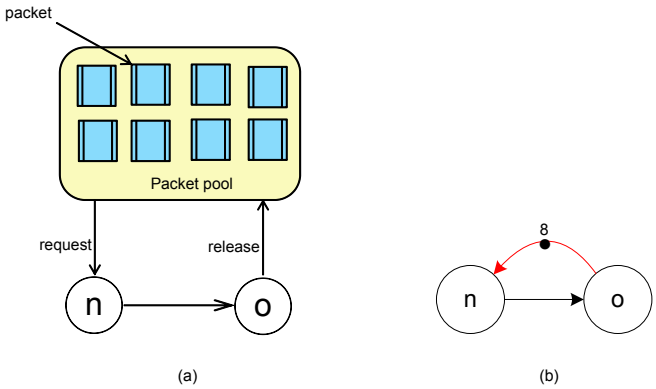

Figure 4.2: Operations supported on a packet pool (a) and a construct to model the size of the packet pool (b).

We model this hand over with a *Sink* actor. Note that the size of the output symbol is usually different from the size of the input symbol. For example, in the DVBT decoder, the size of the input symbol is equal to the size of the OFDM symbol specified by the DVBT specification [DVBa] and the size of the output packet is 16 KB[1]. Moreover, the processed data is typically handed over to the application through an USB interface and in that case the execution time of the *Sink* actor is equal to the time required by the USB interface to transmit a single packet of data.

Modeling packet pools. Packet pools facilitate copy by reference which optimizes the sharing of incoming symbols across tasks in an SDR. In this section we present constructs to model packet pools. There are 3 properties of a packet pool which influence the execution of task that manipulates packets from a packet pool. These properties are 1) number of packets in the packet pool, 2) the size of each packet and 3) the memory where the packet pool resides. In this section we present SDF constructs to model the first property. The constructs to model second and third property are presented in Section 4.2.

Figure 4.2(a) illustrates possible operations on a packet pool. The two operations allowed on a packet pool are 1) *request* a packet and 2) *release* a packet. Figure 4.2(b) presents a construct to model the size of the packet pool. Note that we assume that the requester and releaser tasks are known at design time and are fixed in a scenario of an application. Moreover, we assume that there is only one requester and releaser task for a packet pool. Modeling multiple requester and releaser tasks for a packet pool is proposed as future work.

Modeling the packet resizer. During processing, a symbol is processed by several

[1]The output of the DVBT decoder is a MPEG stream transmitted in the form of packets. On the MARS platform, the MPEG stream is transmitted over USB where the size of a USB packet is 16 KB. However, this size may vary from one platform to another platform.

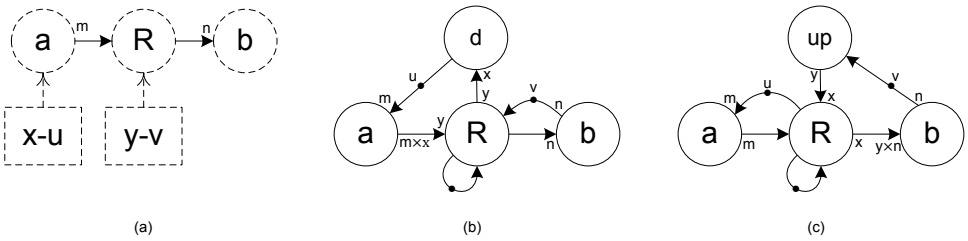

Figure 4.3: Modeling packet resizers present in the system.

tasks present in the digital baseband processing layer. The input/output sizes of these intermediate symbols typically vary. For example, in case of a DVBT decoder, the size of the output of *Viterbi* decoding equals to 4536 bytes (for 64QAM, 8K FFT window size and 1/4 guard interval [DVBa]). This output is processed by the *Reed Solomon* decoder that needs the size of the input packet to be equal to 208 bytes. This size conversion requires packet resizers.

Figure 4.3 presents constructs to model packet resizers when region based memory management is used. Figure 4.3 (a) presents a task graph consisting of 3 tasks (shown in dashed circles), namely, *a,b* and R and 2 packet pools (shown in dotted rectangles). Task a requests m packet(s), where $m \geq 1$, from the packet pool x-u where x is the size of each packet and u is the number of packets present in the packet pool. Once task a gets m packet(s), it fills the data into them and passes them to the resizer R. The resizer is responsible to resize the packets from size of x to y. For that purpose, task R requests n packet(s), where $n \geq 1$, from the packet pool y-v. After resizing the packet(s), task R passes the packets to task b.

Modeling the packet resizers becomes non-trivial when the resizers use packet pools which have less packets than required. It either blocks the sender or the receiver task. For example, assume that the resizer R in Figure 4.3(a) resizes from x to y where x equals 1, u equals 6, y equals 2 and v equals 2. In this case task a blocks after sending the first packet. The resizer needs 3 packets to resize and send the data to b, however, there are only 2 packets present in packet pool y-v. The resizer holds the incoming packet from a (and a remains blocked) until task b returns a packet packet back to the pool. The constructs which model packet resizers must take this blocking due to unavailability of packets into account.

Packet resizers can resize for both cases i.e. when $x > y$ and when $y > x$. Figure 4.3(b) presents an SDF model for the resizer when $x > y$. It models the packet resizing between actor a and b. The actors R and d collectively model the packet resizing and requests to the packet pool. The actor a fires $m \times x$ tokens (analogous to m packets having size x). The resizer reads in y tokens and passes them to actor b. The task R in (a) buffers the additional data. This buffering is modeled in (b) by the dependency from actor a to R. The actor d models the requests from a packet pool x-u. The dependency from d to a models the packets present in the packet pool. Initially, there are u packets in the packet pool. Afterwards, for each x bytes received from R, one token is added to the dependency which is analogous to one packet. Similarly, the dependency from actor b to actor R models the packets in the packet pool y-v.

Figure 4.3(c) presents an SDF model for the resizer when $x < y$. The dependency from actor R to a models the packet requests to the packet pool x-u. The dependency from actor b to *up* models the packet requests from the packet pool y-v. While resizing, the task R buffers the additional data. This buffering is modeled by the dependency from

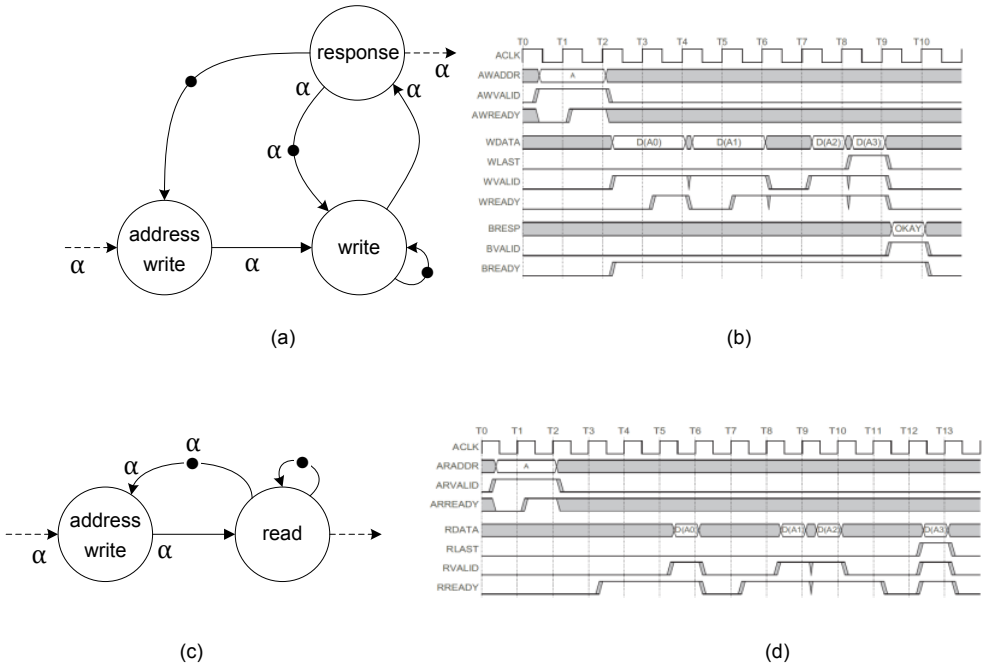

Figure 4.4: AXI write burst (a) with its signal waveform (b) and read burst (c) with its signal waveform (d).

actor R to b. Note that these SDF models will deadlock in case when $m > u$ or $n > v$. This means that the packet pools have less packets than the actors request. Clearly, the system is not operational in that case.

4.2 Modeling the AXI interconnect

The SDF and the FSMSADF MoC assume that there is no communication overhead. In a real system there is however an overhead when communicating data between processors and/or memories. This overhead is due to the time taken by the interconnect to transfer data. We explicitly model this communication overhead. In an MPSoC, multiple masters accessing a shared slave interfere with each other. This interference influences the execution of a task. In this section, we present constructs to model the communication overhead and the interference over the AXI interconnect. In addition, these constructs will model the two properties of the packet pool (see Section 4.1) that are 1) size of a packet present and 2) the location where the packets reside. We model the communication between two tasks (read and write) and insert the model between them according to the interconnects that are accessed and the memory mapping. In particular, we model an AMBA3 AXI interconnect [AXI].

AXI interconnect facilitates *read* and *write* bursts for communication. Figure 4.4 illustrates the operation of read/write bursts (for a detailed discussion of AXI interconnect the reader is referred to [AXI]). An AXI write operation (Figure 4.4 (b)) is initiated by the master by writing the address on the AWADDR register followed by transmission of data in the form of burst (burst length of size 4). The slave confirms the reception of the burst by sending an acknowledge response. This is done by putting the BRESP signal high. Note that the address, data and acknowledgment transmission are governed

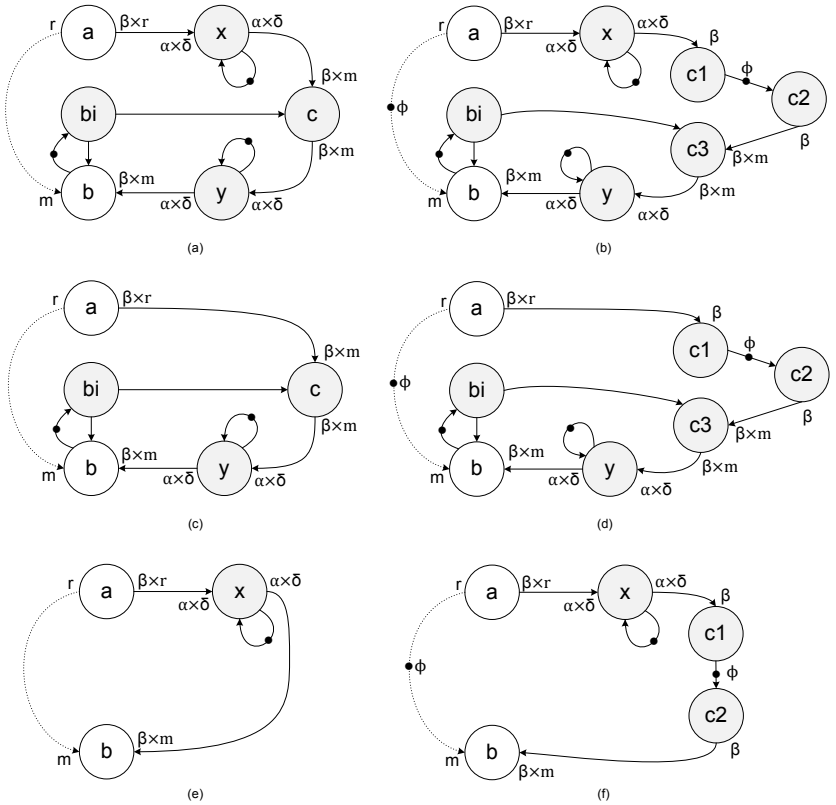

Figure 4.5: AXI read-write model (a), read-write model with initial tokens (b), write only model (c), write only model with initial tokens (d), read only model (e) and read only model with initial tokens (f).

by VALID and READY handshakes. Figure 4.4(a) presents an SDF construct to model an AXI write operation (dotted arrows indicate dependencies to source and destination actors). It consists of 3 actors, namely, *address write*, *write* and *response*. The parameter α denotes the burst length with which the AXI interconnect is configured (4 in our examples). Moreover, for the MARS platform, the execution times for *address write*, *write* and *response* actors is, in worst case, 8 ns (as each operation requires 2 clock cycles in worst case[2]). Similarly, the AXI read burst is presented in Figure 4.4 (d). The read burst does not have a response when the burst ends. The SDF graph to model an AXI read is presented in Figure 4.4 (c). The AXI read model has one actor less compared to the AXI write model as there is no acknowledgment in the read burst but it has VALID/READY handshakes (see Figure 4.4(b) and Figure 4.4(d)).

Assume an SDF graph in which an actor a is connected to an actor b through a dependency edge. Actor a is the source of this edge and has a production rate of r tokens. Actor b, the destination of the edge, has a consumption rate of m tokens. Assume further that the memory used to implement the edge between a and b is located in separate memory which the processors on which a and b are running need to access using the AXI interconnect. This situation models the case in which a source task writes to a memory and a destination task reads it. Figure 4.5 presents parametrized

[2]In the MARS platform, the interconnect is clocked at 300MHz with each tick 3.333ns apart. For worst case analysis, we round the ticks to 4ns. However, the models are generic and the time taken for an operation can be specified according to the clock in an arbitrary system.

SDF constructs to model such a transmission of data. The actors colored gray are which model the communication between the source and the destination actors. These constructs are obtained by combining the AXI read/write models shown in Figure 4.4. The number of initial tokens in the combined model was reduced using the reduction technique described in Chapter 5. For instance, the combined (but not reduced model) contains $2 \times \alpha + 3$ initial tokens. The reduced model (Figure 4.5 (a)) contains only 3 initial tokens.

The parameters used in the model are listed in Table 4.1. The parameter r denotes the rate of the source actor. Similarly, m denotes the read rate of the destination actor. ϕ denotes the number of initial tokens present on a dependency edge between a source and a destination actor. The burst length of an AXI interconnect is denoted by α. The sharing degree determines, in worst case, the number of masters accessing a shared slave simultaneously and is denoted by κ. This sharing degree is computed for each dependency by analyzing the memory mappings. This degree is used to compute the execution times of actors. β denotes the size of a packet used to communicate. The word size of the interconnect is denoted by δ. The parameter t_c determines the time taken by 2 clock cycles.

Parameter	Description
r	source write rate
m	destination read rate
α	AXI burst length
β	packet size
δ	word size of the interconnect
ϕ	number of initial tokens
t_c	time for 2 clock cycles
κ	sharing degree

Table 4.1: Parameters in the AXI read-write models.

Figure 4.5(a) presents an AXI read-write model. It assumes that the source writes the data (modeled by actor x) to the memory and the destination reads the data from this memory (modeled with actor y). This assumption generally holds for SDF actors. Actor c models the completion of transmission of data. Actor bi models the read request from the destination actor to the memory controller to initiate the memory transfer. It ensures that the task which is modeled by actor b is ready to receive incoming data. The execution times of actors bi and c are 0. The execution time of x is computed as $\Upsilon(x) = 2 \times t_c \times \kappa + t_c \times \alpha \times \kappa$. The $2 \times t_c \times \kappa$ represents the time required for *address write* and the acknowledgments. This formula assumes that in worst case all masters interfere in the transmission. The $t_c \times \alpha \times \kappa$ represents the time required by the interconnect to perform the *write burst*. Note that if the write operation is using the bridge then the execution time for the actor x is $\Upsilon(x) = 2 \times t_c \times \kappa + 2 \times t_c \times \alpha \times \kappa$. In a write burst over the bridge the data is written twice during the transmission between the master and the memory slave. Thus the time required to perform a write over the bridge doubles.

Similarly, the execution time for the read actor y is computed as $\Upsilon(y) = t_c \times \kappa + t_c \times \alpha \times \kappa$. In case the read is over the bridge then the execution time is $\Upsilon(y) = t_c \times \kappa + 2 \times t_c \times \alpha \times \kappa$. Figure 4.5(b) presents a variant of the model presented in 4.5(a) which models the initial tokens present on the elaborated buffer (shown in dotted arrows). The actors *c1*, *c2* and *c3* model the initiation of a read request from the task modeled by actor b. These actors have an execution time equal to 0. Note that it is possible to model the initiation

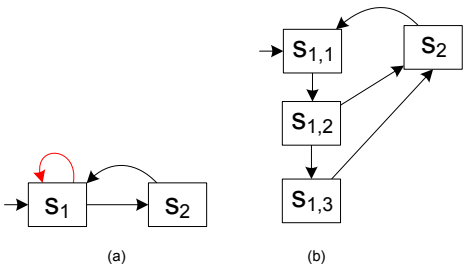

Figure 4.6: Bounding scenario transitions in an FSMSADF graph.

with a single actor (instead of three) but this will make the number of initial tokens present in the construct dependent on other construct parameters. We will show during the case study that reducing the initial tokens reduces the time required by the analysis algorithms. Therefore we prefer to have a model with fewer tokens instead of a model with fewer actors.

Figure 4.5 (c) and (e) presents models for situations when the transmission is write or read only respectively with their variants modeling initial tokens presented in 4.5(d) and (f). The execution times of the tasks present are computed in the similar way as described for 4.5(a) and (b). Note that the read only and write only constructs model the incoming stream from the DFE (write only) and the outgoing stream over the USB interface (read only).

4.3 Bounding FSMSADF statespace

The FSM present in an FSMSADF models the scenario transitions in an application. In the FSMSADF MoC, it is allowed to infinitely stay in a state. Figure 4.6(a) presents an FSM in which the graph can stay in state s_1 infinitely. It is often desired to limit the stay of a state in itself. For example, to incorporate designer feedback to the model. Moreover, the FSMSADF statespace may be too large due to store the complete statespace in the memory (which is required by the analysis algorithms). We bound the transitions by expanding the FSM and removing the self loops on the bounded state. For example, Figure 4.6(b) presents an FSM in which the number of times the FSM stays in state s_1, denoted by x, is bounded by $2 \leq x \leq 3$. This bounding is performed by replacing s_1 with 3 states, $s_{1,1}$, $s_{1,2}$ and $s_{1,3}$ and allowing transitions from $s_{1,i}$ to s_2 only when $a \leq i \leq b$ where a and b are the lower and upper bounds respectively. The bounding of transitions for each state increases the number of states in the FSM to $b - a + 1$. We define *depth* as the number of states expanded when removing a self edge. For example, the depth for s_1 in Figure 4.6(b) is 2. The effect of bounding the statespace for the DVBT model is described in Section 7.4.

4.4 Related work

Modeling SDRs using dataflow MoCs is studied in [Yan09, BBL08, SGM+11]. A Mode Controlled DataFlow (MCDF) graph is used to model a DVBT decoder in [Yan09]. The modeling in [Yan09] aims to find a feasible schedule for an MCDF graph satisfying the real time requirements of a DVBT decoder. An FSMSADF based model of the Long Term Evolution (LTE) SDR is presented in [SGM+11]. It models the dynamism

present in an LTE baseband processing tasks. However, [Yan09, BBL08, SGM+11] do not provide constructs to model typical operations like periodic sources and sinks, packet pools and the communication over an interconnect. These constructs are essential to model the system level operation of an SDR. Analysis of suitable MoCs to model SDRs is described in [BBL08]. It concludes that statically scheduled dataflow graphs pessimistically model SDRs. This conclusion coincides with the results of our experiments performed in Chapter 7. Moreover, it discusses applicability of other MoCs to model SDRs. A detailed comparison of dataflow MoCs is described in [PRE], based on which, the FSMSADF MoC is proposed in this thesis to model the dynamism present in an SDR.

The Fractional Rate Dataflow (FRDF) extension to SDF is described in [OH04]. This extension allows to have fractional read and write rates on a dependency. FRDF closely relates to the packet resizers described in Section 4.1. However, FRDF requires an extension to model packet pools. As shown in an example in Section 4.1, the packet resizers may block actors in the case when the number of packets in a packet pool is less than the required number of packets. This blocking influences the timing analysis, thus is essential to be modeled.

The AXI models described in Section 4.2 are inspired from the interconnect models discussed in [JSS+11, ASSG08, Stu07]. [JSS+11] describes a parametrized SDF graph to model communication between two tiles present in an MPSoC. It models buffering, serialization and deserialization of packets in the interconnect present in the MAMPS platform [Kum09]. However, [JSS+11] assumes homogeneous interconnect i.e. the communication overhead is the same for all tiles[3]. The AXI models described in Section 4.2 facilitate different models for different types of communications in an MPSoC i.e. when the communication is heterogeneous. An SDF graph to model bi-rate communication service provided by an interconnect is presented in [ASSG08]. Using bi-rate communication model provides tighter estimates for timing requirements. However, [ASSG08] does not specify how to combine multiple bi-rate SDF graphs to model multiple masters present in an MPSoC.

4.5 Conclusion

In this chapter, constructs to model SDRs were presented. The tasks present in the digital baseband processing layer are modeled as actors and the data passing between the tasks are modeled as dependencies between the actors. Usually, SDRs consist of periodic source/sinks, packet resizers and packet pools. Constructs to model such operations in an SDR were presented. Moreover, constructs to model the AXI interconnect were presented in this chapter. These constructs are evaluated in Chapter 7.

[3]The communication model can be used with different parameters for different types of communication. In order to estimate the interference caused by multiple masters (to compute parameters) the models require extensions.

Chapter 5

Reduction of FSMSADF Graphs

The Finite State Machine based Scenario Aware DataFlow (FSMSADF) model of computation (MoC) was introduced in Chapter 3. A dependency edge between actors is *redundant* if removing the edge does not change the start times of the actors present in an application. In this chapter we focus on reducing an FSMSADF graph by removing redundant dependencies and actors present in an FSMSADF. Removal of redundant dependencies may lead to removal of initial tokens which in turn reduces time taken by the analysis algorithms (see Section 7.4 for details).

In the following section, we motivate the reduction of FSMSADF graphs. Section 5.2 describes the reduction approach. Section 5.3 introduces the Max-Plus algebra [BCOQ92] which is used in the reduction technique. In Section 5.4 the reduction approach itself is described. Section 5.5 presents an algorithm to convert the simplified Max-Plus matrix to an HSDF graph. In Section 5.6 the reduction approach is extended to reduce FSMSADF graphs. Section 5.7 describes work related to the reduction technique. Finally, Section 5.8 concludes this chapter.

5.1 Motivation

It is desirable to have intuitive and simple dataflow models. Models near to the concept are intuitive. Simpler models capture the same behavior but require less analysis time and effort. The technique proposed in this chapter reduces the number of initial tokens present in an FSMSADF graph while preserving the timing behavior of an application.

Figure 5.1 presents examples of a model with the same throughput but varying number of initial tokens and actors. In Figure 5.1(a) an SDF model for an AXI *write* burst is described (this model is discussed in detail in Section 4.2). The actor a models the address write, c models a write and b models the write response sent back by the slave. α models the burst length of the AXI interconnect.

Figure 5.1(b) presents an equivalent model, in terms of throughput, to Figure 5.1(a) but it has two actors a and v compared to three actors in Figure 5.1(a). The number of dependencies and initial tokens has also reduced from five to two and from three to one

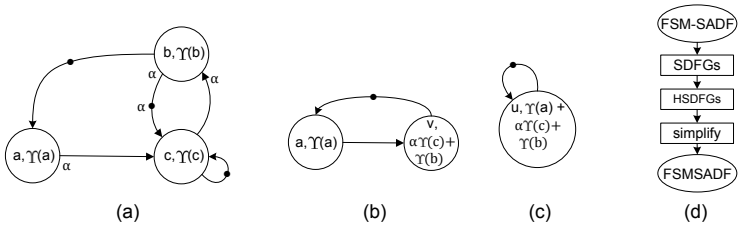

Figure 5.1: The reduction approach (d) and illustrative SDF graphs (a, b, c).

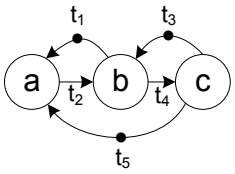

Figure 5.2: An illustrative HSDF graph.

respectively. The SDG graph shown in Figure 5.1(b) abstracts from the AXI write and just account for the time required to perform writes (hence the throughput is equal) or in other words it clusters α writes and groups the time needed with the time required to write the response. To ensure that the timing behavior of the actors in the resulting graph is identical to the original graph, v has an execution time equal to $\alpha \times \Upsilon(c) + \Upsilon(b)$ where $\Upsilon(b)$ and $\Upsilon(c)$ denotes the execution times of actors b and c in Figure 5.1(a). The SDF graph presented in Figure 5.1(c) has equal throughput as Figure 5.1(a) and Figure 5.1(b). It has only one actor with a dependency to itself. The execution time of this actor is equal to $\Upsilon(a) + \alpha \times \Upsilon(c) + \Upsilon(b)$ where $\Upsilon(a)$, $\Upsilon(b)$ and $\Upsilon(c)$ denote the execution times of actors a, b and c in Figure 5.1(a).

Clearly, we preserve the throughput of the SDF graphs shown in Figure 5.1(a), Figure 5.1(b) and Figure 5.1(c) but we lose the buffer requirements and the applicable analysis techniques (buffer throughput trade off [Stu07]). The technique proposed in this chapter preserves the throughput as well as the timing behavior of an application. The technique aims to reduce the number of initial tokens present in an FSMSADF graph preserving the start times of actors present in the FSMSADF graph. In the sequel, we refer to this technique as *reduction*.

5.2 Reduction approach

The overall approach of the reduction technique is presented in Figure 5.1(d). An FSMSADF is composed of an FSM which models the transitions of the system between *scenarios* and an SDF graph for each *scenario*. We simplify the SDF graphs present in the FSMSADF by converting the SDF graphs to equivalent HSDF graphs and then simplifying the HSDF graphs while preserving the schedule of the SDF graphs (In case an actor is removed, the remaining actors have the same schedule which may have different execution times). We focus to reduce the number of initial tokens present in the model in order to reduce the analysis time. The reduced HSDF graphs are used to create a reduced FSMSADF graph.

The reduction technique described in this chapter requires the SDF graphs present in a FSMSADF to be converted to an equivalent HSDF. Any arbitrary SDF can be

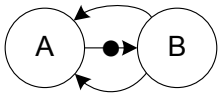

Figure 5.3: An illustrative HSDF graph showing redundant dependencies.

converted to an equivalent HSDF as specified in [LM87, SB09]. However, the conversion may lead to an exponential increase in the number of actors present in the HSDF graph compared to the SDF graph. Moreover, [Gei09] presents a Max-Plus based SDF to HSDF conversion technique, which, in many cases shows a smaller increase in the number of actors present in the HSDF. For our reduction technique, any SDF to HSDF conversion technique suffices. In this chapter, we assume that the SDF graph to HSDF is converted using any of the well known methods [LM87, Gei09]. The HSDF graph is converted to its Max-Plus representation as described in the following section.

5.3 Max-Plus representation of HSDF graphs

In this section an algorithm to generate a Max-Plus representation for an arbitrary HSDF graph is described. Let A be the set of actors present in an HSDF graph. Moreover, we define a Max-Plus term to be of the form $max(\alpha_1, .., \alpha_n) + \theta$ where α_i is an arbitrary Max-Plus term and θ is a constant. The reduction method is illustrated on the HSDF graph shown in Figure 5.2[1]. For reference, the tokens produced on the channels are labeled as $t_1, ..., t_5$. It is possible to convert an arbitrary actor present in an HSDF graph to such a form. An actor has a set of input channels and a certain amount of time delay incurred due to the atomic execution of the actor. The input channels become parameters to the max operator and the delay is added as a constant in the corresponding equation of the form $max(\alpha_1, .., \alpha_n) + \theta$. We use this property while generating the Max-Plus terms using Algorithm 1.

Each term characterizes the production of a token by an actor as described in [Gei11]. Algorithm 1 describes a procedure to construct Max-plus terms for an arbitrary HSDF graph. The algorithm iterates over all actors present in the graph and generates a Max-Plus term for each actor (which corresponds to each token produced by the actor). The term consists of the max operator applied to the tokens over the incoming channels to the actor and added by the execution time of the actor[2]. Using Algorithm 1, the following Max-Plus terms are generated for the HSDF graph presented in Figure 5.2.

$$
\begin{aligned}
t_2 &= max(\bar{t_1}, \bar{t_5}) + \Upsilon(a) \\
t_1 = t_4 &= max(\bar{t_3}, t_2) + \Upsilon(b) \\
t_3 = t_5 &= max(t_4) + \Upsilon(c)
\end{aligned}
$$

We denote the initial tokens present in an HSDF graph by a bar over the token name. For example, initial token $\bar{t_1}$. We need the non-initial tokens to be present to analyze redundant dependencies with non zero initial tokens. Figure 5.3 shows an HSDF graph. In the graph, one of the dependencies from B to A is redundant as removing one of them will not change the timing behavior of the HSDF graph. Note that this redundancy

[1] We assume any arbitrary execution times for the actors and do not specify them in the figure.

[2] The inverted commas indicate that the term is an output string rather than a statement in the algorithm itself.

cannot be identified by only having Max-Plus terms with initial tokens. In the following section we present the simplification steps performed to reduce the HSDF.

Algorithm 1 Generate Max-Plus terms for an HSDF graph

Input: Set of actors A and the set of dependencies D present in the HSDF graph
Output: Set of Max-Plus terms *terms*

```
1: terms = φ
2: for all a ∈ A do
3:    inDep = getIncomingDependencies(a,D)
4:    tokens = getTokens(inDep)
5:    t ='max(tokens) + Υ(a)'
6:    terms = terms ∪ t
7: end for
8: return terms
```

5.4 Reduction

In this section we present the definitions that are used in reducing the Max-Plus representation of an HSDF graph. Applying the reduction steps may remove terms from the Max-plus representation. The usage of these steps is illustrated on the terms generated in Section 5.3.

Definition 5.1: (SEMIFIELD) Let $\kappa = \{x | x \in \mathbb{R} \cup \{-\infty\}\}$ be a semifield endowed with the following two operations:

- $max : \kappa \times \kappa \to \kappa$ is a binary operator such that $max(\alpha, \beta)$ determines the maximum of $\alpha, \beta \in \kappa$. The max operator is associative, commutative and has a zero element $-\infty$ such that $max(-\infty, \alpha) = \alpha$.
- The operator + forms a group on $\kappa_* = \kappa \backslash \{-\infty\}$. It is a commutative, distributive with respect to the max operator and has an identity element 0 such that $\alpha + 0 = 0 + \alpha = \alpha$.

For the proof of associativity, commutativity of the *max* operator and the distributivity of the + operators we refer the reader to [BCOQ92].

Definition 5.2: (UNARY) The *max* operator has no effect over a single operand i.e. *max(a)* = *a*, $\forall a \in \kappa$. ◇

By definition, the *max* operator is a binary operator. In case of SDFGs its is possible to obtain a representation where *max* is applied to a single operand. In such a case, the max operator can be omitted. Intuitively, the maximum element out of a list containing a single element is the only element present in the list.

Definition 5.3: (IDEMPOTENT) The *max* operator is idempotent i.e. *max(a,a)* = *a*, $\forall a \in \kappa$. ◇

Multiple applications of the max operator to the same operand are idempotent i.e. the multiple applications have no effect. For a formal proof of idempotency of the *max* operator we refer the reader to [BCOQ92].

Definition 5.4: For any arbitrary Max-plus terms α, β, ϵ, if it holds that $\alpha \leq \beta + \theta$ then $max(\alpha, max(\beta, \epsilon) + \theta) = max(\beta, \epsilon) + \theta$. ◇

The intuition behind Definition 5.4 is if $\alpha \leq \beta$ then the expression depends on the result of the inner max thus the outer max operator could be omitted. Formally, the proof for the definition is as follows:

$$
\begin{aligned}
L.H.S &= max(\alpha, max(\beta, \epsilon) + \theta) \\
&= max(\alpha, max(\beta + \theta, \epsilon + \theta)) && \textit{by distributivity of } + \textit{ over max} \\
&= max(max(\alpha, \beta + \theta), \epsilon + \theta) && \textit{by associativity of max} \\
&= max(\beta + \theta, \epsilon + \theta) && \textit{as } \alpha \leq \beta + \theta \textit{ holds} \\
&= max(\beta, \epsilon) + \theta && \textit{by distributivity of plus} \\
&= R.H.S
\end{aligned}
$$

In order to simplify Max-Plus terms using Definition 5.4, it is required to identify and verify the condition whether $\alpha \leq \beta + \theta$ holds. The following definition allows to do so.

Definition 5.5: For an arbitrary Max-Plus term α and constants θ, λ, $max(\alpha)+\theta+\lambda \geq max(\alpha) + \theta$ where $\theta \geq 0$ and $\lambda \geq 0$. ◇

In case of SDFGs, θ and λ are always non-negative as, by definition, the actor execution times in SDFGs are non-negative.

Definition 5.6: For any arbitrary Max-Plus terms α, β, ϵ if it holds that $\alpha \leq \beta$ then $max(\alpha, \beta) = max(\beta)$ and $max(\alpha, \beta, ..., \epsilon) = max(\beta, ..., \epsilon)$. ◇

The *max* operator applied to the terms for which the inequality could be determined from the terms itself, for example, using Definition 5.5 then the max operator can be reduced. Consider the following terms generated in the previous section for the HSDF presented in Figure 5.2.

$$
\begin{aligned}
t_2 &= max(\bar{t_1}, \bar{t_5}) + \Upsilon(a) \\
t_1 = t_4 &= max(\bar{t_3}, t_2) + \Upsilon(b) \\
t_3 = t_5 &= max(t_4) + \Upsilon(c)
\end{aligned}
$$

Reducing the above equations yields the following equations:

$$
\begin{aligned}
t_2 &= max(\bar{t_1}, \bar{t_5}) + \Upsilon(a) \\
t_1 = t_4 &= max(\bar{t_3}, max(\bar{t_1}, \bar{t_5}) + \Upsilon(a)) + \Upsilon(b) && \textit{substitute } t_2 \\
t_3 = t_5 &= t_4 + \Upsilon(c) && \textit{by definition } 5.2 \\
&= max(\bar{t_3}, max(\bar{t_1}, \bar{t_5}) + \Upsilon(a)) + \Upsilon(b) + \Upsilon(c) && \textit{substitute } t_4
\end{aligned}
$$

Using Definition 5.5, $t_5 \geq t_1$, the reduction results in the following set of equations:

$$
\begin{aligned}
t_2 &= max(\bar{t_5}) + \Upsilon(a) && \textit{by definition } 5.6 \\
&= \bar{t_5} + \Upsilon(a) && \textit{by definition } 5.2 \\
t_1 = t_4 &= max(\bar{t_3}, max(\bar{t_5}) + \Upsilon(a)) + \Upsilon(b) \\
t_3 = t_5 &= max(\bar{t_3}, max(\bar{t_5}) + \Upsilon(a)) + \Upsilon(b) + \Upsilon(c)
\end{aligned}
$$

The reduction technique can be automated by using an existing Term Rewrite System (TRS) implementation [GTSKF03, CDE+02]. Definitions 5.1-5.6 can be formulated as rewrite rules and an HSDF can be formulated as terms. The formulated terms and rewrite rules can then be used to compute the normal forms of the rules. These normal forms are the simplified terms, from which, it is possible to construct the simplified HSDF graphs as specified in Section 5.5. The automation of the reduction technique is proposed as a future work.

5.5 Conversion to an HSDF graph

In this section we construct an HSDF graph from the simplified Max-Plus expressions described in the previous section. The simplified terms from the previous section are as follows:

$$
\begin{aligned}
t_2 &= \bar{t_5} + \Upsilon(a) \\
t_1 = t_4 &= max(\bar{t_3}, t_2) + \Upsilon(b) \\
t_3 = t_5 &= t_4 + \Upsilon(c)
\end{aligned}
$$

The simplified terms could be of the form $max(max(...), ..., max(...)) + \theta_1 + ... + \theta_n$. In order to convert the simplified terms to back to an HSDF graph, we first convert the terms to the form $max(t_1, ..., t_n) + \theta$. Algorithm 2 performs such a conversion. The notation used in Algorithm 2 and Algorithm 3 are as follows. Let Φ denote the set of tokens present in an HSDF graph. Let the set of terms be T where a term denoted by τ is of the form $max(t_1, ..., t_n) + \theta$. Let $constant(\tau)$ denote the θ of the term τ and let $tokens(\tau)$ return the set of tokens $\{t_1, ..., t_n\} \subseteq \Phi$ present in the max operator of the term τ. Similarly, let $operands(\tau)$ provide the set of operands of the max operator present in the term τ. Let $\lambda : T \rightarrow A$ be function that provides an actor $a \in A$ for a term $\tau \in T$. Let $\delta : \Phi \rightarrow T$ be a function that provides the term $\tau \in T$ which defines the token $t \in \Phi$ where every token present in a simplified HSDF is defined by a term. Let the function $\kappa : T \rightarrow \Phi$ be function that provides a token $t \in \Phi$ defined by the term $\tau \in T$. Let the function $isToken(t)$ determine whether t is a token or not.

Algorithm 2 converts the simplified terms to the form $max(\alpha_1, .., \alpha_n) + \theta$. The input to the algorithm is the set of Max-Plus terms. The for loop between lines 2-6 conditionally introduces the max operator to a term if it is not present. The for loop between lines 8-18 ensures that each max operator present in the simplified terms must only contain tokens. If a max operator has another max operator as an operand then the operand max is replaced by the token it defines. Finally, the last for loop between lines 20-24 sums up the constants present in a Max Plus term. As a result, the simplified terms, of the form $max(\alpha_1, .., \alpha_n) + \theta$, are output of the algorithm. For example, the output of the Algorithm 2 for the simplified equations is as follows:

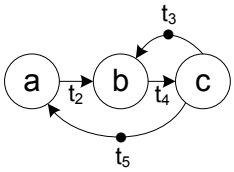

Figure 5.4: The reduced HSDF.

$$t_2 = max(\bar{t_5}) + \Upsilon(a)$$
$$t_1 = t_4 = max(\bar{t_3}, t_2) + \Upsilon(b)$$
$$t_3 = t_5 = max(t_4) + \Upsilon(c)$$

Algorithm 2 Simplify to get terms of the form $max(t_1, ..., t_n) + \theta$

Input: Set of terms T_0
Output: Set of terms T such that all terms are of the form $max(t_1, ..., t_n) + \theta$

1: $T_1 = \phi$
2: **for all** $\tau \in T_0$ **do**
3: **if** τ is of the form $t_\alpha + \theta$ **then**
4: $T_1 = T_1 \cup \{max(t_\alpha) + \theta\}$
5: **end if**
6: **end for**
7: $T_2 = \phi$
8: **for all** $\tau \in T_1$ **do**
9: $O_n = \phi$
10: **for all** $O \in operands(\tau)$ **do**
11: **if** $isToken(O)$ **then**
12: $O_n = O_n \cup \{O\}$
13: **else**
14: $O_n = O_n \cup \{\kappa(O)\}$
15: **end if**
16: **end for**
17: $T_2 = T_2 \cup \{max(O_n) + constants(\tau)\}$
18: **end for**
19: $T = \phi$
20: **for all** $\tau \in T_2$ **do**
21: **if** τ is of the form $max(...) + \theta_1 + ... + \theta_n$ **then**
22: $T = T \cup \{max(...) + \theta\}$ such that $\theta = \theta_1 + ... + \theta_n$
23: **end if**
24: **end for**
25: **return** T

Algorithm 3 constructs an HSDF graph from the simplified terms. It converts the input terms to the form $max(\alpha_1, .., \alpha_n)+\theta$ by using Algorithm 2. The for loop between lines 4-9 creates a set of actors and a set of dependencies from the terms and returns the HSDF graph as a result. For each equation an HSDF actor is created. For example, there are three actors in Figure 5.4, one for each simplified equation. The dependencies between actors are computed by analyzing the parameters to a max operator. For example, a dependency exists between actor a and c as t_5 and is defined by corresponding equation

of c used in the max operator of equation defining t_2. The HSDF graph corresponding to the simplified terms is presented in Figure 5.4. The simplification resulted in removal of a redundant dependency in the SDFG which had an initial token. Both the original and the reduced SDFGs have same throughput (i.e 1/3) but the reduced SDFG has two initial tokens instead of three. Moreover the timing behavior of both SDF graphs remains the same. This can be verified by extending the trace comparison algorithm presented in Section 6.5.

Algorithm 3 Construct simplified HSDF graph

Input: Set of terms T_0
Output: Simplified HSDF graph G_{HSDF}
1: $T = SimplifyTerms(T_0)$
2: $A_G = \phi$
3: $D_G = \phi$
4: **for all** $\tau \in T$ **do**
5: $\quad A_G = A_G \cup (\tau, \theta_\tau)$
6: $\quad$ **for all** $t \in tokens(\tau)$ **do**
7: $\quad\quad D_G = D_G \cup (\lambda(\delta(t)), \lambda(\tau), 1, 1, 0)$
8: $\quad$ **end for**
9: $\quad$ **for all** $\bar{t} \in tokens(\tau)$ **do**
10: $\quad\quad D_G = D_G \cup (\lambda(\delta(t)), \lambda(\tau), 1, 1, 1)$
11: $\quad$ **end for**
12: **end for**
13: **return** $G_{HSDF}(A_G, D_G)$

5.6 Reduction of FSMSADF

The FSMSADF model of computation is introduced in Chapter 3. It consists of an FSM, which models scenario transitions and a set SDF graphs. Previous sections describe a technique to reduce HSDF graphs. In this section we extend the reduction technique to FSMSADF graphs.

Algorithm 4 describes a technique to reduce FSMSADF graphs. This reduction is performed in two steps. The first step is compute the set of redundant initial tokens. A token in an FSMSADF graph is only redundant if it is redundant in all SDF graphs in the FSMSADF graph. In the second step, the tokens redundant in all HSDF graphs are removed. The algorithm accepts an FSMSADF graph G_F as the input and produces a reduced FSMSADF graph G_{Fr} as the output. The for loop between lines 3-7 computes reduced HSDF graphs for the SDF graphs present in the FSMSADF. At line 8, tokens removed from all HSDF graphs are computed. The for loop between lines 11-16 computes the reduced HSDF by only removing the tokens which are redundant in all scenarios. Finally, at line 17, the reduced FSMSADF graph is returned.

5.7 Related work

Reduction techniques for SDF graphs are specified in [Gei09]. These techniques aim to reduce large SDF graphs into SDF graphs which can be analyzed in less time. The reduced graph is a conservative estimate of the original graph as the reduction technique specified in [Gei09] is an approximation. The technique presented in this chapter

Algorithm 4 Reduce an FSMSADF graph

Input: An FSMSADF graph $G_F(S, F)$
Output: A reduced FSMSADF graph G_{Fr}

1: $states = Q(F)$
2: $\Sigma = map(F)$
3: $diff = []$
4: **for all** $s \in states$ **do**
5: $\quad h = ConvertToHSDF(SDF(\Sigma(s)))$
6: $\quad h_{reduced} = ReduceHSDF(h)$
7: $\quad diff[s] = removedTokens(h, h_{reduced})$
8: **end for**
9: $S = \bigcap_{s \in states} diff[s]$
10: $newHSDFs = \phi$
11: $\Sigma_{new} = []$
12: **for all** $s \in states$ **do**
13: $\quad h = ConvertToHSDF(SDF(\Sigma(s)))$
14: $\quad h_{reduced} = ReduceHSDFBounded(h, S)$
15: $\quad newHSDFs = newHSDFs \cup h_{reduced}$
16: $\quad \Sigma_{new}[\Sigma(s)] = h_{reduced}$
17: **end for**
18: **return** $G_{Fr}(F_{new}(Q(F), q_0(F), \delta(F), \Sigma_{new}), newHSDFs)$

reduces FSMSADF graphs. Moreover, it preserves the schedule of an FSMSADF graph. However our aim coincides with the aim of the reduction technique described in [Gei09]. i.e. to reduce the analysis time.

5.8 Conclusion

In this chapter a method of reducing an FSMSADF graph is proposed. It is shown by an example that the method can reduce the number of channels and initial tokens present in an HSDF graph. The reduction technique removes non-critical dependencies while preserving the schedule of an HSDF graph. Furthermore, the technique was extended to reduce an FSMSADF graph. Automation of the reduction technique can be performed by using an existing TRS system and is proposed as a future work.

Chapter 6

Trace Extraction

Tracing is a form of software logging useful for validation. A trace records system events in the order of their occurrence. This chapter describes the design and implementation of a tracing framework developed to trace applications running over the MARS platform. In the next section, the motivation behind the development of such a framework is discussed. Section 6.2 presents the challenges present in implementing such a framework over the MARS platform. Section 6.3 describes the architecture of the tracing framework. Section 6.4 describes the API provided by the tracing framework. Section 6.5 describes an algorithm to compare two traces to validate a model. Finally, Section 6.6 concludes this chapter.

6.1 Motivation

Digital baseband processing is typically implemented as a set of streaming tasks that share data in a pipelined fashion in which a task present in the pipeline may depend on a number of previous tasks. Tracing provides insight into the behavior of these tasks and can be used to validate a model. The tracing framework is used to validate the DVBT model described in Chapter 7. Validation of such an execution sequence requires logging of relevant events that occur during the execution of these tasks. In most cases, the order of occurrence of these events is of interest to validate an execution. For example, Figure 6.1(a) shows the task graph of an application[1]. It consists of three tasks that have dependencies as specified in the task graph. The execution trace of the task graph is shown in Figure 6.1(b). Figure 6.1(c) presents an FSMSADF graph that models the application shown in Figure 6.1(a). It has three scenarios, namely, *s1*, *s2* and *s3*. The execution times of all actors is *25 ms* except for actor A, in scenario *s2*, has an execution time of *26 ms*. The additional 1 ms accounts for initialization delay over the ARM processor. The trace generated from this FSMSADF graph is shown in Figure 6.1(d) (shaded executions of task A indicate that there are simultaneous executions from task C). All buffers in the application, at maximum, accommodate a single token. The dependencies with initial tokens model the size of buffers. Trace comparison allows to assess *tightness* (see Section 6.5) of a model. Section 6.5 describes an algorithm to

[1] A dashed circle represents a task present in the application.

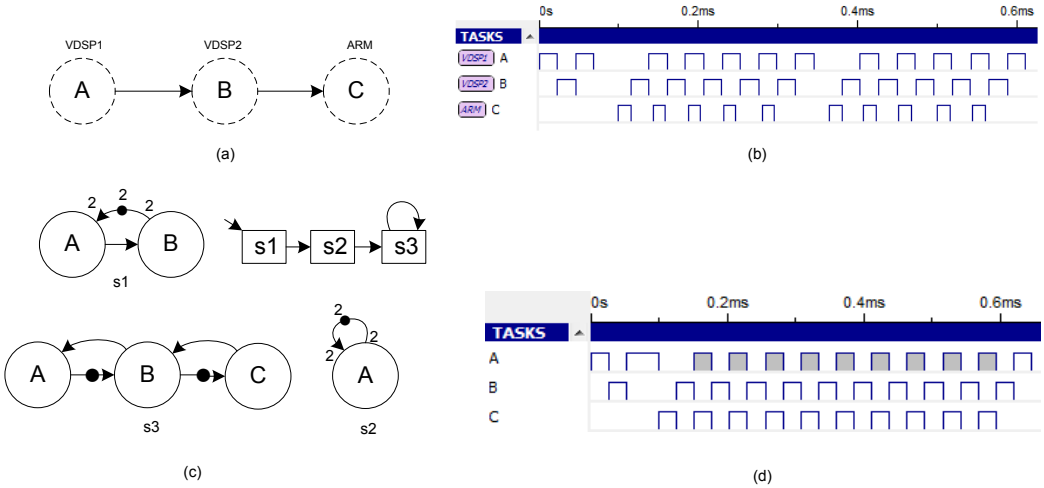

Figure 6.1: Sample application (a), its execution trace (b), an FSMSADF graph modeling the sample application (c) and the model trace of the FSM-SADF graph (d).

compare two traces. Using the algorithm, the model trace presented in Figure 6.1(d) was found to be tight with respect to the trace presented in Figure 6.1(b).

6.2 Challenges

Implementation of the tracing framework is a challenging task. Tracing large number of events in a short period of time requires large bandwidth. Bandwidth is usually scarce on embedded platforms. In case of the DVBT decoder, there are approximately 100,000 relevant events per second. Assuming each event requires two words (one word is 4 bytes) a bandwidth of 800 KB/s is needed. In case of MPSoCs, events occur in parallel which further increases the bandwidth requirement. Similarly, the number of events influences the memory requirements. For example, for the DVBT decoder, 1 MB of RAM facilitates storage of a maximum trace of 1.5 seconds. Moreover, the selection of events to trace is non-trivial (especially when a large number of events exist) and mostly data dependent. The tracing application should facilitate application designers to specify events of interest.

The tracing framework should use minimal resources to avoid interference with the application being traced. Moreover, the tracing framework must operate in residual resources which are typically scarce in embedded systems. Furthermore, a trace is analogous to a series of events happening in time. Browsing the events in textual form is tedious for the designer and therefore a trace visualizer is needed.

6.3 Architecture

The architecture of the tracing framework is presented in Figure 6.2. It was designed using the Sea-of-DSP (SoD) and SoD+ API [BB04] on the MARS platform. The rounded rectangles represent processing cores. The squares inside each core represent set of tasks (which are traced) executing on each core (tasks are shown as circles inside the squares).

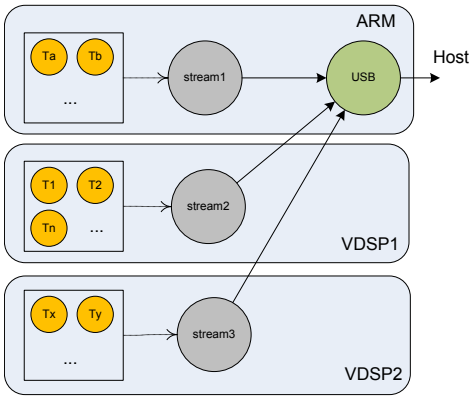

Figure 6.2: The architecture of the tracing framework.

```
#include<TDExecProbe.h>

/* simple task */
void Task()
{
    log_start(MY_ID);

    do_something();

    if(condition == true)
        commit_log();

    log_end(MY_ID);
}
```

Figure 6.3: A sample task logging execution trace using the tracing framework.

The circles, outside the squares, denote virtual tasks accessed through the SoD *External API*. The external API allows any task to write to the input of a virtual task using the handle to the task (instead of having a buffer for communication). Using the external API reduces the number of buffers needed and facilitates a generic architecture (independent of the traced tasks). The dotted arrows represent the transmission of data using the SoD external API. The arrows represent SoD buffers. The circles outside the squares are SoD tasks that multiplex the streams of traces from the processing cores over the USB interface to the host. The tracing framework is designed to stream the traces out of the MARS using a separate USB pipe platform thus facilitating generation of theoretically infinite stream of traces.

6.4 Tracing API

The tasks to be traced are specified by the designer by using the tracing API which consists of three macros 1) logStart(ID) 2) logEnd(ID) and 3) logEvent(ID). Each task is assigned a unique ID which is provided in the macros. Moreover, each log-event must be *committed* (before logging the end-event) by the designer in order to ensure that the event is logged. This allows designers to control logging of events. Once the events are logged, the tracing framework transmits the stream out of the MARS platform. Figure

6.1(b) shows the traces of application shown in Figure 6.1(a) (while executing over the MARS platform). Figure 6.3 presents an example task that logs its execution events using the API provided by the tracing framework.

6.5 Trace comparison

Large traces are difficult to compare. The comparison described in this section aims to validate a system trace generated from a system by comparing it with a model trace generated from a model. This comparison allows the designers to verify that a model is conservative. In the sequel an FSMSADF graph is considered *tight* if its model trace is conservative to a system trace.

The intuition behind trace comparison is as follows. An execution of a task is defined as a set of instructions executed atomically. A task present in a digital baseband processing layer typically has many executions. The start time and the end time of an execution of a task must not be larger than its corresponding execution in the model trace. If this holds for all executions present in a system trace then the model is considered to be *tight*. Algorithm 5 describes a procedure to verify tightness of a model trace with respect to a system trace that it models. Let $getTasks(T)$ return all tasks present in a trace T. At line 1, the tasks common to the system trace and the model trace are computed. Algorithm 5 assumes that only the tasks common to the model trace and to the system trace are of interest to verify tightness. The function $getExecutionSequence(t, T)$ returns the sorted sequence of executions of task t in trace T. The function $count(s)$ counts the number of executions in a sequence s. The $getExecution(s, i)$ gets the i^{th} execution of a task in a sequence s. For each task, the algorithm checks whether the number of executions of a task are the same as its number of executions in the model trace. This check is performed between lines 5-7. For each execution of a task the algorithm checks whether the start time and the end time of an execution in the system trace is greater than the corresponding execution in the model trace. If so, the algorithm returns *false*. If not, *true* is returned indicating that the model is tight.

6.6 Conclusion

Tracing provides insight into the operation of an application. It is useful in validation. In this chapter the design details for the tracing framework were presented. This tracing framework allows tracing of SoD applications on the MARS platform. Moreover, the tracing framework allows to compare two traces in order to assess the tightness of a model. We use the tracing framework to assess the tightness of the DVBT model presented in Chapter 7.

Algorithm 5 Compare a system trace with a model trace to assess tightness.

Input: A system trace T_s and a model trace T_m
Output: Boolean indicating whether the model trace is tight with respect to the system trace.

```
1:  tasks = getTasks(T_s) ∪ getTasks(T_m)
2:  for all t ∈ tasks do
3:      seq_s = getExecutionSequence(t, T_s)
4:      seq_m = getExecutionSequence(t, T_m)
5:      if count(seq_s) ≠ count(seq_m) then
6:          return false
7:      end if
8:      for i = 1 → count(seq_s) do
9:          e_s = getExecution(seq_s, i)
10:         e_m = getExecution(seq_m, i)
11:         if start(e_s) > start(e_m) or end(e_s) > end(e_m) then
12:             return false
13:         end if
14:     end for
15: end for
16: return true
```

Chapter 7

Case Study

Constructs to model digital baseband processing are discussed in Chapter 4. Using them, we model the digital baseband processing in a DVBT decoder which is implemented on the MARS platform. The decoder operates with the following operational parameters, 8K, 1/4 guard rate and 64QAM[1]. The required throughput for this mode is 893 *symbols/s*. In this case study, the SDF^3 dataflow analysis toolkit [SGB06] is used to analyze the models presented in this chapter.

An SDF model of the DVBT decoder is described in Section 7.1. Section 7.2 presents an FSMSADF model of the DVBT decoder and compares it with the SDF model. Section 7.3 uses the FSMSADF model to evaluate whether DVBT Diversity can be implemented over a single VDSP. In the same section, bottlenecks present in the DVBT decoder are identified. Section 7.4 describes the bottlenecks in the approach and the models presented in this chapter. Section 7.5 describes the upper and lower bounds used to bound the FSMSADF statespace of the DVBT decoder. Using these bounds the model trace for an FSMSADF graph presented in this chapter is generated. Section 7.6 describes the results of the comparison of a system trace with a model trace of an FSMSADF graph. Finally, we conclude this chapter in Section 7.7.

7.1 SDF model of the DVBT decoder

Streaming tasks present in a DVBT decoder execute atomically. In SDF, an actor models an atomic computation. We model the tasks present in the DVBT decoder as SDF actors. The passing of data between these tasks (see Section 1.3.1) is modeled as dependencies between actors. Figure 7.1 presents an SDF graph that models the DVBT decoder implementation over the MARS platform. The *src* actor models the periodic arrival of OFDM symbols. The *sink* actor models the periodic transmission of the MPEG stream for application specific processing over the USB interface. The *dfe isr* actor models the interrupt service routine that receives the data from the *src* and passes it to the *acquisition* actor. The dependency edges with initial tokens on them model the size of the buffers used to pass pointers to packets except the edges with

[1]DVBT feed is broadcasted with these parameters in Eindhoven, The Netherlands.

name	**time** (ns)	**name**	**time** (ns)
src	1120000	dfe isr	12017
acquisition	42471	agc	188280
fft	801934	tps sync	238631
equalization	453411	upl	168
deqam	140782	flora t1	24920
packet resizer 1	6720	desc sync	11000
flora t2	18240	ts filter	20480
packet resizer 2	6680	usb out	22120
sink	267000		

Table 7.1: Execution times of actors in the DVBT SDF graph.

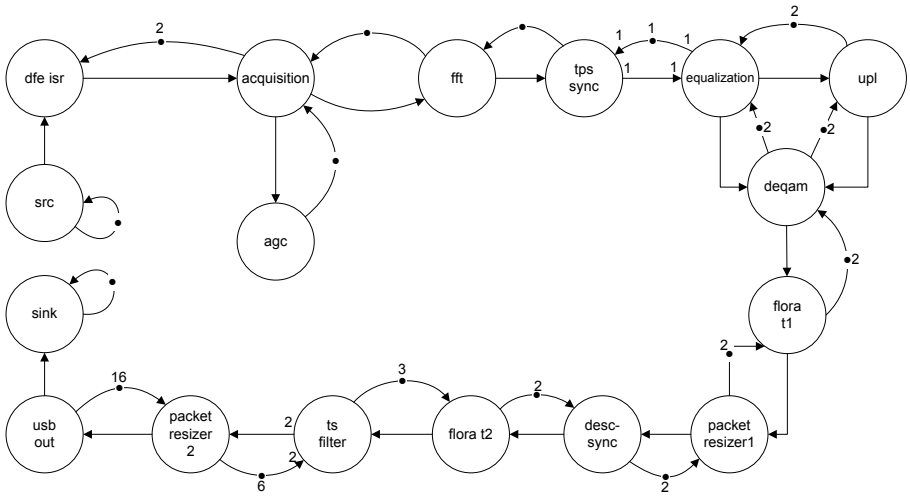

Figure 7.1: An SDF graph modeling the digital baseband processing in the MARS DVBT decoder.

initial tokens over the periodic source and sink. This passing continues in a similar way in the remainder of the DVBT pipeline. Finally, the data reaches the *usb out* actor which models the transmission of USB packets. The execution times of the actors in the SDF graph are presented in Table 7.1. The execution times of the actors are the worst case observed times during the operation of the DVBT decoder[2]. The execution time of *src* is exactly the period between the arrival of two consecutive OFDM symbols [DVBa]. The execution time of *sink* is computed as follows:

$$Bandwidth\ available = 480Mb/s = 60MB/s \approx 60 \times 10^6 B/s$$

$$Max\ packets\ transmitted = \frac{bandwidth}{packet\ size} = \frac{60 \times 10^6}{16 \times 10^3} = 3.75 \times 10^3$$

$$Time\ for\ one\ packet = \frac{1}{3.75 \times 10^3} = 0.266 \times 10^{-3} s/packet \approx 267000ns$$

[2]The execution times are the worst case observed times over the VDSPs and ARM processors. Note that these times are specific to the MARS platform.

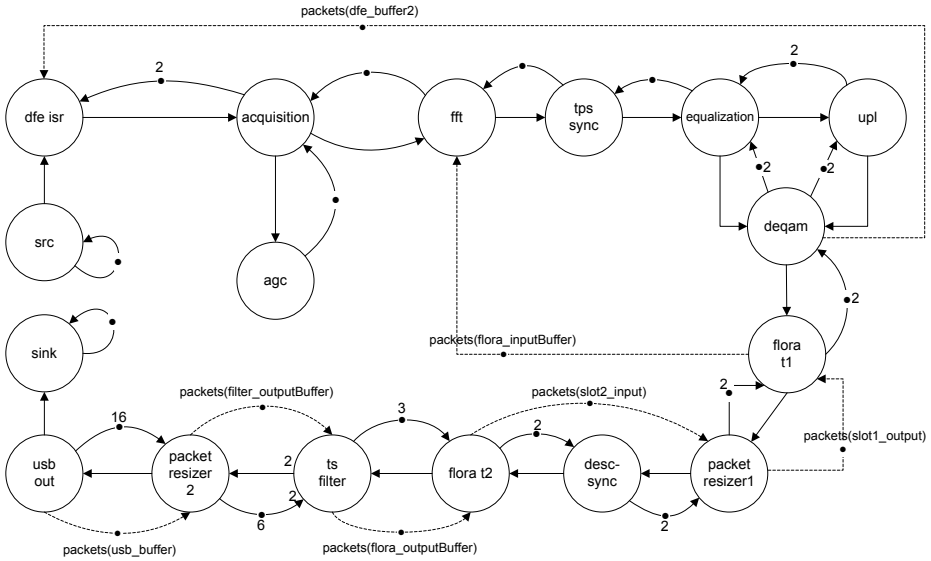

Figure 7.2: An SDF graph to model the packet pools and the digital baseband processing in the MARS DVBT decoder implementation.

name	size	name	size
dfe_buffer2	2	flora_inputBuffer	2
slot1_output	2	slot2_input	2
flora_outputBuffer	3	filter_outputBuffer	6
dfe_buffer2	2	flora_inputBuffer	2
usb_buffer	32		

Table 7.2: Number of packets present in the packet pools.

The maximum throughput[3] of the SDF model shown in Figure 7.1 is 961 symbols/s meeting the throughput constraint of 893 symbols/s. The SDF graph shown in Figure 7.1 does not model the packet pools.

The SDF graph shown in Figure 7.2 models the packet pools used in the DVBT decoder. The dashed arrows model the size of the packet pools. The number of initial tokens present on the dashed dependencies is equal to the number of packets in the packet pool ($packets(n)$ denotes the number of packets present in the packet pool n). Table 7.2 lists the number of packets present in a packet pool. The throughput of the SDFG shown in Figure 7.2 is 961 symbols/s. Note that the throughput remains the same as the SDF graph shown in Figure 7.1 even if the packet pools are modeled. It is due to the fact that the number of packets present in the packet pool is sufficient not to reduce the throughput of the application. However, the SDF graph shown in Figure 7.2 does not model the communication overhead over the AXI interconnect.

The constructs to model communication over the AXI interconnect are described in Section 4.2. In order to model the communication overhead between two actors, an AXI model is inserted between the actors. The type of the AXI model depends on the mapping of the communicating actors and the mapping of the packet pool. Figure 7.3 (a) describes a graph to compute the type of AXI interconnect when two tasks read/write

[3]The maximum throughput is reported assuming that the period of the *src* can be reduced.

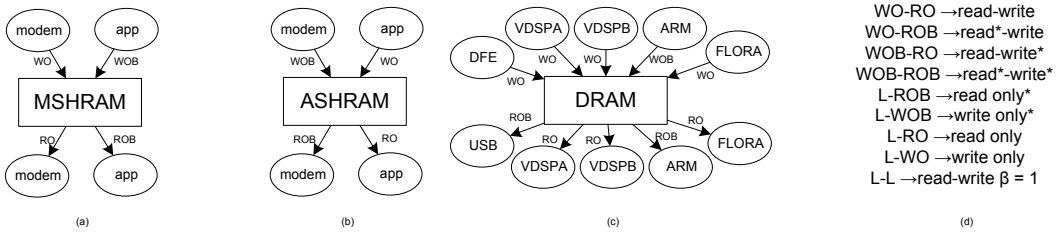

Figure 7.3: Graphs used to determine appropriate constructs to model communication over the AXI interconnect.

to MSHRAM. The VDSPA, VDSPB and FLORA are present in the *modem* section of the MARS platform. The *app* (abbreviation of application) section contains ARM and USB DMA. A source task mapped onto a processor in *modem* section writes data to MSHRAM that is read by a task in the *app* section then the communication type is *WO-ROB*. Note that, for the DRAMs either in VDSPA or VDSPB, the communication type is determined based on the processor and hence the processor names are explicitly specified in Figure 7.3 (c). Figure 7.3 (d) shows the type of the interconnect model for each communication type. For *WO-ROB*, *read*-write* models the communication[4]. The parameters for the AXI models (see Section 4.2) are as follows. The source write rate r and the destination read rate m are determined from the source and destination actors respectively. The AXI burst length α for the MARS platform is 4 bytes. The word size γ is 8 bytes for the MARS platform. The time needed for an AXI operation t_c is 8 ns. The number of initial tokens ϕ is determined from the dependency between the source and the destination actor. The sharing degree for each slave β is computed by counting the number of actors writing to the slave. Each dependency, except the dependencies from *src* and to the *sink* actor, is replaced by an AXI model in the SDF graph based on the type computed using the graphs shown in Figure 7.3. In the sequel we refer to this replacement as *elaboration*. The throughput after elaboration of the SDF graph presented in Figure 7.2 is 958 symbols/s.

The tasks present in the DVBT decoder share the processors in a round-robin fashion. The VDSPA is shared between *dfe isr*, *acquisition*, *fft*, *tps sync*, *equalization*, *upl* and *deqam* tasks. Figure 7.4 shows an SDF graph that models the packet pools and the processor sharing among the tasks present in the DVBT decoder. The dependency (shown as a bold edge) from *deqam* to *dfe isr* models the sharing of VDSPA. Adding this dependency enforces a round-robin scheduling order over the tasks that share the processor. Similarly, the ARM processor is shared between *flora t1*, *packet resizer1*, *desc-sync*, *flora t2*, *ts filter*, *packet resizer2* and *usb out*. A dependency is added from *usb out* to *flora t1*. This dependency models the sharing of the ARM processor. The ARM processor also executes several interrupt service routines. These routines increase the execution times of the tasks they interrupt. Thus the overhead of the routines is included in the task execution times and in not modeled explicitly. In case more than one applications share the processors, the approach used to model processor sharing in this thesis requires an extension. After elaboration, the throughput of the SDF graph shown in Figure 7.4 is 558 symbols/s. The cyclic dependency between *dfe isr*, *acquisition*, *fft*, *tps sync*, *equalization*, *upl* and *deqam* has MCM and is the bottleneck in the system. It prohibits to meet the throughput constraint of the DVBT decoder.

The throughputs of the SDF models presented in this section are summarized in Table

[4]A * indicates doubling the time of read and/or the write actor. For example, *read** indicates to double the execution time of the actor modeling the read in the AXI model.

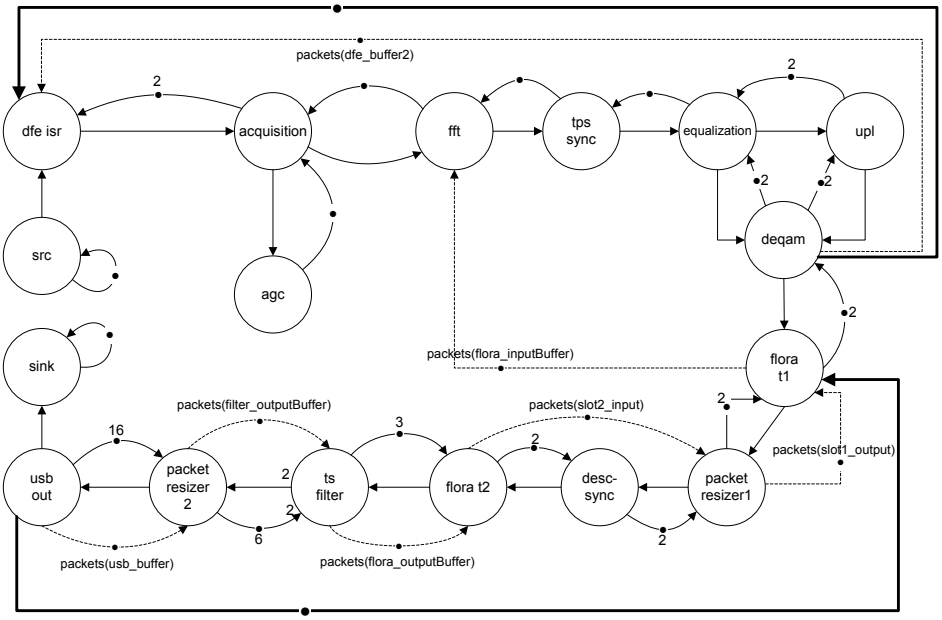

Figure 7.4: An SDF graph to model the packet pools, processor sharing and the digital baseband processing in the MARS DVBT decoder implementation.

model	SDF	FSMSADF	Improvement(%)
DVBT	961	1216	26
Packet pool	961	1216	26
AXI	958	1211	26
Processors	558	939	68

Table 7.3: Throughputs (symbols/s) of different models used in the case study.

7.3. The cycle between *dfe isr*, *acquisition*, *fft*, *tps sync*, *equalization*, *upl* and *deqam* is the bottleneck. Note that packet resizers are not modeled in the case study described in this chapter. See Appendix B for details. The SDF graphs described in this chapter do not model the varying resource requirements of the DVBT decoder. They are modeled as specified in the following section.

7.2 FSMSADF model of the DVBT decoder

A DVBT decoder has varying resource requirements. In this thesis these requirements are modeled using the FSMSADF MoC. The FSMSADF MoC facilitates *scenarios*.

Actor / Scenario	Demod	Sync	CFO	ACQ	IQ
acquisition	20474	20474	20474	20474	42471
fft	96374	96374	801934	-	-
tps sync	238631	17742	-	-	-

Table 7.4: Varying actor execution times (in *ns*) in the DVBT decoder.

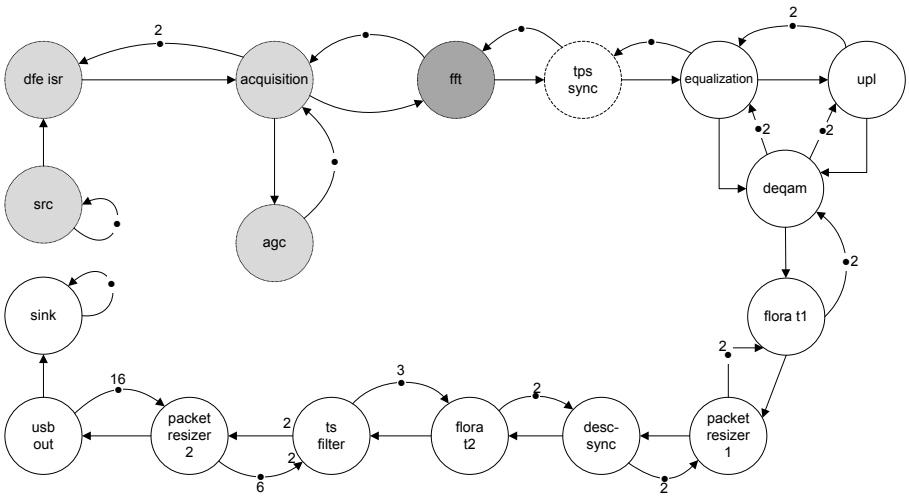

Figure 7.5: An FSMSADF graph modeling the digital baseband processing in the MARS DVBT decoder.

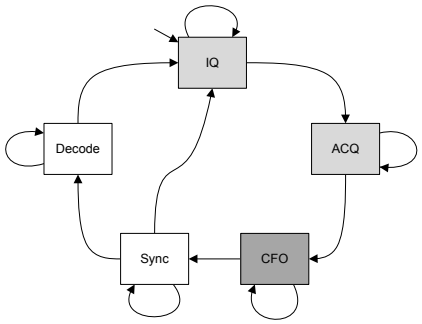

Figure 7.6: The FSM corresponding to the FSMSADF graph presented in Figure 7.5.

The varying resource requirements are split into a set of scenarios (this split is evaluated in Section 7.4). Each scenario has a corresponding SDF graph. Figure 7.5 presents an FSMSADF graph that models the DVBT decoder implementation over the MARS platform, with its FSM shown in Figure 7.6. The FSM consists of five states, namely, *IQ*, *ACQ*, *CFO*, *Sync* and *Decode*. Initially, the decoder is in the *IQ* state. The actors active in a state have the same shading scheme as their associated FSM state[5]. When the decoder switches to a next state, functional blocks become active incrementally. Similarly, an SDF graph corresponding to a scenario consists of additional actors compared to its previous scenario. For example, all actors after *equalization* are only present in the SDF graph corresponding to the *Demod* scenario. The actors that have varying execution times are presented in Table 7.4 (remaining actors have the same execution times as specified in Table 7.1).

The maximum achievable throughput of the FSMSADF graph presented in Figure 7.5 is 1216 symbols/s. It meets the throughput constraint of 893 symbols/s. However, it does not model the packet pools in the DVBT decoder. The FSMSADF graph shown

[5]We present the SDF graphs corresponding to each scenario with shades. Each SDF graph consists of actors with the same shade and all previous actors in the DVBT pipeline. Thus when scenario switch occurs, the actors become active incrementally.

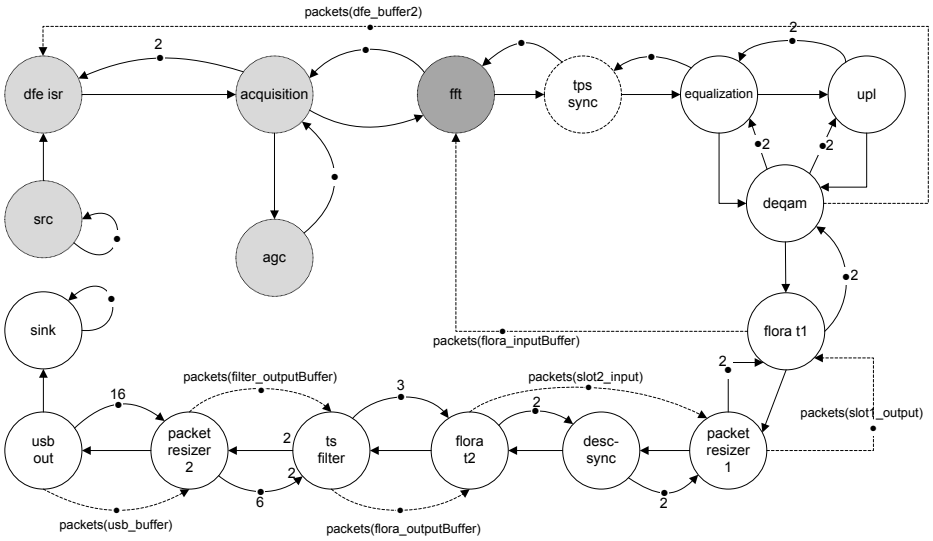

Figure 7.7: An FSMSADF graph to model the packet pools and the digital baseband processing in the MARS DVBT decoder implementation.

in Figure 7.7 models packet pools (details of modeling packet pools are the same as in Section 7.1). The FSM of this FSMSADF graph is the same as shown in Figure 7.6. The throughput of the FSMSADF graph that models the packet pool is 1216 symbols/s.

The throughput of the FSMSADF graph shown in Figure 7.7 reduces to 1211 symbols when it is elaborated. The method of elaboration is same as specified in Section 7.1. Figure 7.8 shows an FSMSADF graph that models the packet pools, processor sharing and the digital baseband processing in the MARS DVBT decoder. The throughput of this graph after AXI elaboration is 939 symbols/s. It meets the throughput constraint of the DVBT decoder. Table 7.3 lists the throughputs of the SDF and the FSMSADF graphs discussed in this section and in Section 7.1. The improvement column shows the percent improvement in throughput when using an FSMSADF graph instead of an SDF graph. Clearly, the FSMSADF graphs have a higher throughput than their corresponding SDF models. This concludes that the FSMSADF models presented in this chapter facilitate tighter estimations of the resource requirements of the decoder. An advantage of having tighter estimates is the fact that it avoids over allocation of resources.

7.3 Early evaluation and improvements

Early evaluation and bottleneck detection are promises of model based design. An FSMSADF based DVBT model is used to evaluate whether DVBT Diversity, a variant of DVBT, can be implemented using a single VDSP. Secondly, we identify bottlenecks present in the MARS DVBT decoder. The SDF^3[SGB06] dataflow analysis toolkit is used to perform analysis on our models. The details of each experiment are as follows.

Diversity on a single VDSP. DVBT Diversity is a type of DVBT in which two input streams (from two antennas) are processed by a decoder to increase the quality of the output MPEG stream. The FSMSADF model presented in Figure 7.7 that models the digital baseband processing and the packet pools is used to investigate

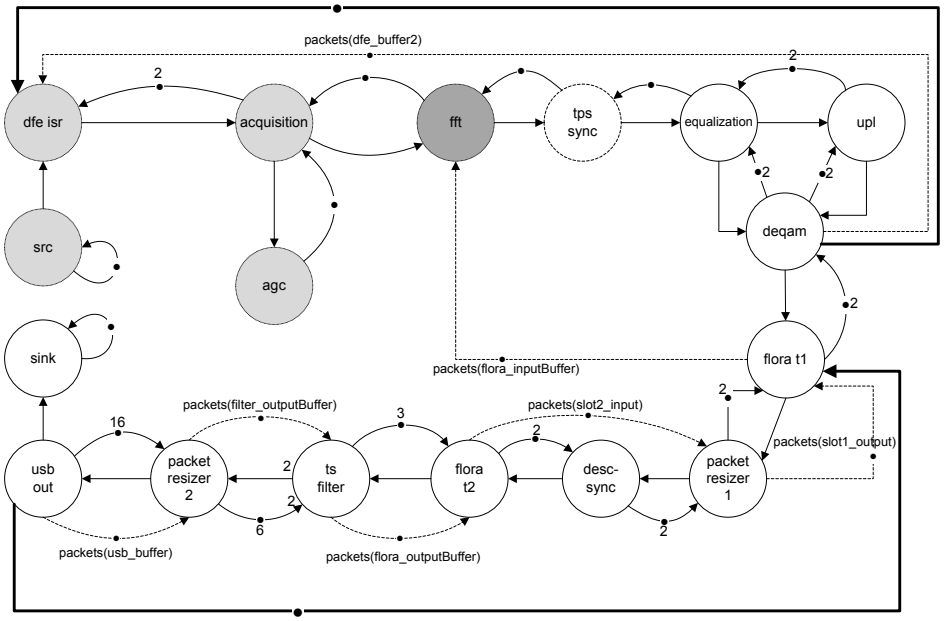

Figure 7.8: An FSMSADF graph to model the packet pools, the processor sharing and the digital baseband processing in the MARS DVBT decoder implementation.

whether it is possible to implement DVBT Diversity decoder over a single VDSP. In a DVBT Diversity decoder, all tasks running on the VDSPs process information from two OFDM symbols. In order to verify the implementation of DVBT Diversity, the execution times of the *acquisition*, *fft*, *tps*, *equalization* and *deqam* actors is doubled. The FSMSADF graph (that models DVBT and packet pools), shown in Figure 7.7, with doubled execution has a maximum achievable throughput of 722 symbols/s. It does not meet the throughput constraint of the DVBT decoder i.e. 893 symbols/s. It is concluded that it is not possible to implement a DVBT Diversity decoder on a single VDSP using the current implementation of the task running on the VDSPs.

Bottlenecks and improvements. The maximum achievable throughput of the DVBT decoder is limited by the critical scenario sequence and the critical actor firings in the sequence. Finding the critical scenario sequences and the critical actors firings will identify the bottlenecks present in the system. These bottlenecks are candidates for improvement. The critical scenario sequence in the DVBT decoder is when the decoder stays in the *Decode* scenario. The tasks executing over the VDSP, namely, *dfe isr*, *acquisition*, *fft*, *tps sync*, *equalization*, *upl* and *deqam* are bottlenecks in the implementation of the DVBT decoder. Thus, any task running over the VDSP is a candidate for improvement. Moreover, *equalization* has the maximum execution time among the tasks causing the bottleneck. Note that the improvement is needed only when the tasks are decoding the input OFDM symbols. Thus only the functionality executed in the *Demod* scenario requires optimization.

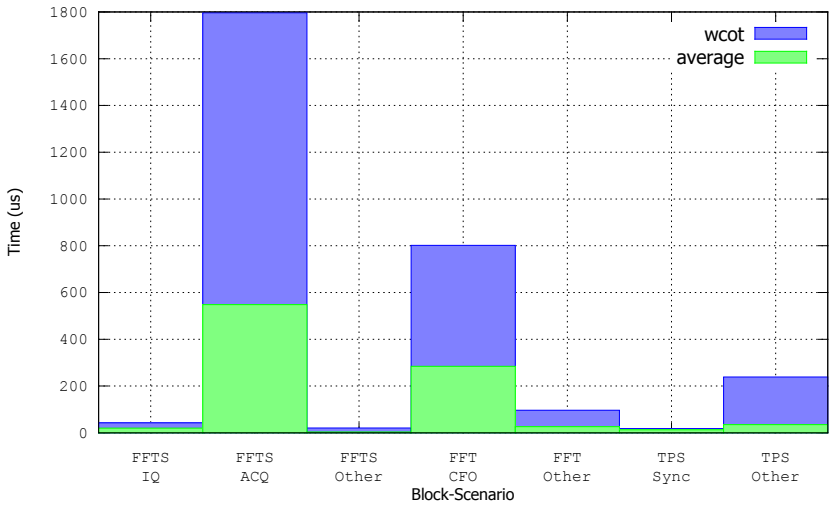

Figure 7.9: Comparison of execution times of several blocks in different scenarios.

7.4 Bottlenecks in the approach and the model

The approach followed in this thesis is evaluated in this section. It is evaluated whether the identified scenarios are sufficient to capture varying resource requirements. Moreover, we identify which factors affect the time complexity of the analysis algorithms.

Identified scenarios. The switching between scenarios is data and carrier state dependent leading to different execution times of blocks across scenarios. Figure 7.9 illustrates this by a comparison. It presents the Worst Case Observed Time (WCOT) and the average case execution times of the functional blocks present in the DVBT decoder. Only the execution times of FFT-synchronization (FFTS), FFT and TPS decoding (TPS) vary during the operation of the decoder. In Figure 7.9, the blocks are arranged on the x-axis in *block-scenario* format, where *block* is the name of a block and *scenario* is the name of the scenario it is in. Notice the fluctuation of the execution times of the FFTS block across the *IQ, ACQ* and *Other* [6] scenarios. Considering the maximum WCOT of a block during the analysis will be too pessimistic and it appeals to explicitly model the data and the state dependent behavior. For the DVBT decoder, this behavior (scenarios) is observable during its operation. The set of possible transitions between these behaviors can be statically determined. Moreover, there exists a significant difference between the average and the WCOT of FFTS block in the ACQ scenario and the FFT block in the CFO scenario due to the internal dynamism present in these blocks. This difference appeals to model this dynamism as scenarios and is proposed as a future work.

Factors contributing to the time complexity. The time complexity of the analysis algorithms is influenced by the FSMSADF graph. In this section, the effect of the number of initial tokens and depth (see Section 4.3) of an FSMSADF graph on the runtime of some analysis algorithms is discussed. The depth of an FSM present in an FSMSADF graph influences the number of states present in the FSMSADF statespace. Figure 7.10 shows the result of an experiment which analyzes the influence of the depth of an FSM over the FSMSADF statespace. In this experiment, the depth of the DVBT

[6] *Other* indicates, for a functional block, the rest of the scenarios which are not shown in the figure.

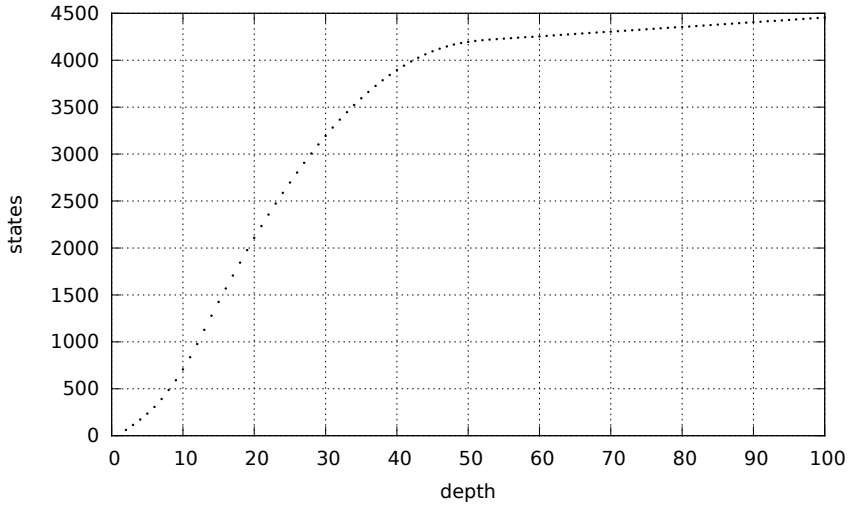

Figure 7.10: The influence of the depth of an FSM over the FSMSADF statespace.

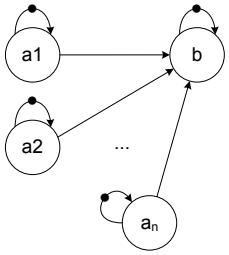

Figure 7.11: An FSMSADF graph used to analyze the influence of the number of initial tokens on the analysis algorithms.

FSM was varied between 1-100 for the *Demod* scenario. Initially, the states in the FSMSADF statespace increase exponentially with respect to the depth of the FSM until they reach the saturation point. This saturation occurs due to the fact the new time stamp vectors explored by the statespace generation algorithm are dominated by the previously explored time stamps. However, in a case when not all self edges are removed from the FSM, the saturation point may never be reached and the exponential growth in the number of states may continue.

Secondly, the number of initial tokens present in an FSMSADF graph influences the time complexity of the analysis algorithms. In order to analyze this influence we increase the number of initial tokens in the FSMSADF graph specified in Figure 7.11. The number of initial tokens is varied between 0-16000. Figure 7.12 shows the results of the experiment. While selecting an FSMSADF graph for this experiment, the aim is to be as general as possible i.e. the experiment should only consider the increase in the number of initial tokens into account. Clearly, the analysis time increases quadratically with respect to the number of initial tokens. It is quadratic because the size of a Max-Plus matrix increases quadratically with the number of initial tokens in an FSMSADF graph.

Type	IQ	ACQ	CFO	Sync
Lower bound	11	21	5	75
Upper bound	11	21	5	130

Table 7.5: Upper and lower bounds for expanding the DVBT FSM.

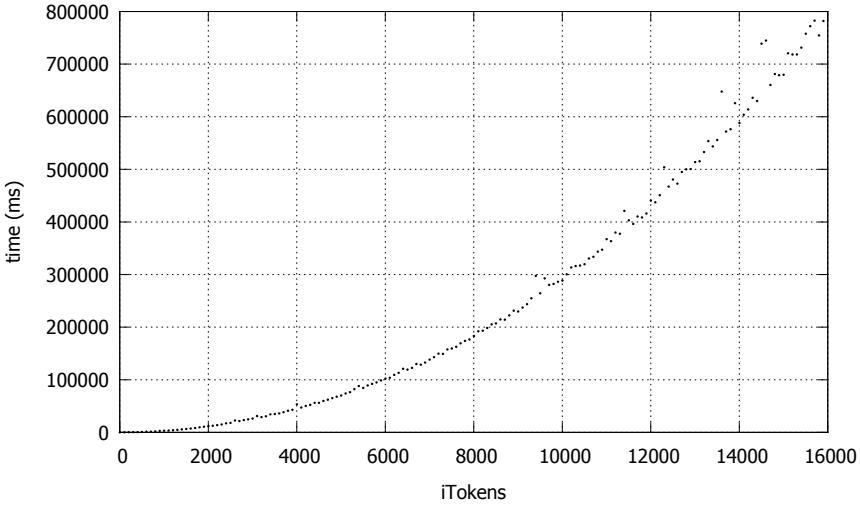

Figure 7.12: The influence of the number of initial tokens over the analysis time.

7.5 Upper and lower bounds for the DVBT decoder

An approach to expand an FSM in an FSMSADF graph is described in Section 4.3. The expansion is required in order to limit the statespace of an FSMSADF graph [SGTB11]. Expansion allows to compute the statespace in limited memory[7]. If not expanded, it is not possible to load the FSMSADF statespace even in 150GB of memory. Computing the FSMSADF statespace is required to perform the worst case performance analysis [SGTB11] and to generate a model trace [PRE].

The upper and lower bounds used for the expansion of the FSM of the DVBT decoder are specified in Table 7.5. These bounds are computed by performing a series of experiments. In these experiments, we vary the intensity of the signal transmitted to the DVBT decoder. The variation of the signal was between the minimum intensity on which the decoder remains operational and the maximum intensity. It is evident from the bounds that varying the signal intensity only affects the number of iterations the decoder stays in *sync* scenario. The expansion limits the FSMSADF statespace but, on the other hand, increases the number of states (and scenarios) present in an FSMSADF graph. Moreover, the expansion modifies the application behavior and should not be considered as a reduction technique for the FSMSADF statespace.

[7]The expansion performed to the FSM for the DVBT decoder fits in 4 GB of memory.

7.6 Comparison of the DVBT model trace with the system trace

The bounds presented in the previous section were used to limit the statespace (see Section 4.3) of the DVBT FSMSADF graph presented in Figure 7.7. It models the packet pools and the digital baseband processing in the DVBT decoder. The FSM in the FSMSADF was expanded to limit the FSMSADF statespace. Limiting the FSMSADF statespace is required to generate the model trace (without limiting the statespace it does not fit into the memory). The model trace of the DVBT decoder was generated by the algorithms specified in [PRE]. The system trace for the DVBT decoder implementation on the MARS platform was generated using the tracing framework described in Chapter 6. The traces were compared using Algorithm 5 specified in Chapter 6. The model trace is not tight with respect to the system trace, as the algorithm returned *false*. This is because the *acquisition* task in *ACQ* scenario, in the system trace, has a worst case execution time of 1720000 ns whereas in the model the worst case execution time is 20474 ns. This leads to the failure of the tightness condition verified by Algorithm 5. Using 1720000 ns as the worst case execution time for the *acquisition* task in *ACQ* scenario would be too pessimistic. The *acquisition* task takes 1720000 ns to align the boundary of the incoming OFDM symbol. The *acquisition* task (in worst case) performs busy wait[8] for 1120000 ns). The additional 600000 ns account for the processing after the symbol boundary is aligned, thus, leading to an execution time of 1120000 + 600000 = 1720000 ns. Using 1720000 ns as a worst case, the execution time of the *acquisition* task is greater than the OFDM symbol period. Moreover, this happens once during the *ACQ* scenario. Modeling this behavior will be reported (by the FSMSADF timing analysis algorithms) as a violation of the throughput constraint. Therefore, in this thesis, we ignore modeling the alignment. An approximation to model this behavior is to add a separate scenario with a relaxed throughput constraint, in which the DVBT decoder aligns the OFDM symbol boundary.

7.7 Conclusion

This chapter presents several SDF and FSMSADF models for the MARS DVBT decoder. These models differ on the types of behavior they model. The SDF model of the DVBT decoder failed to fulfill the throughput constraint when it models the processors present in the MARS platform, because the SDF model is not able to model the varying resource requirements of the DVBT decoder and it is pessimistic. The FSMSADF model of the DVBT decoder (which models processors, AXI interconnect, packet pools and the digital baseband processing) fulfills the throughput constraint. In this chapter, we identified bottlenecks and improvements in the current implementation. Improving any task running on the VDSP will increase the throughput of the system (or the resources can be shared with other applications). Furthermore, we analyzed the bottlenecks in the modeling approach used in this thesis. Our experiments indicate that an increase in the number of initial tokens present in an FSMSADF graph may increase the runtime of the analysis algorithms quadratically.

[8] A while(1) loop with a condition to check whether the OFDM symbol boundary is aligned.

Chapter 8

Conclusion and Future Work

Model based design of SDRs is proposed as a solution to the requirements of wireless operators and technology providers. In this thesis, an approach to model SDRs is proposed. Section 8.1 concludes this thesis. However, several extensions to the approach are possible. These extensions are discussed in Section 8.2.

8.1 Conclusion

Model based design of SDRs is proposed as a solution to the requirements of wireless operators and technology providers. Radios incorporate several signal processing operations. These operations, according to the conceptual model of SDRs, are categorized as *signal processing*, *digital baseband processing* and *application specific processing*. This thesis describes constructs to model digital baseband processing.

An application consists of computation and communication. In this thesis constructs to model both the computation and communication are described. Namely, constructs to model atomic execution of tasks, periodic sources and sinks, packet pools, and packet resizers are described. Moreover, constructs to model the AXI interconnect are described in this thesis. Using these constructs, as a case study, the DVBT decoder implementation over the MARS platform is modeled.

In the case study, the FSMSADF MoC is used to model the varying resource requirements of a DVBT decoder. The results indicate that SDF MoC pessimistically models the varying resource requirements, thus, limiting the maximum achievable throughput. In the case study, using the designed model, it was concluded that the DVBT Diversity cannot be implemented on a single VDSP without optimizing the current implementation. The tasks to optimize are identified by finding the bottleneck in the implementation. The approach described in this is assessed in the case study. This assessment revealed that the dynamism present inside the scenarios is a candidate to be modeled as sub-scenarios. Futhermore, increasing the number of initial tokens present in an FSMSADF graph may increase the analysis time quadratically. Thus, it is desired to reduce the number of initial tokens present in an FSMSADF graph. In this thesis, a technique to reduce the number of initial tokens in an FSMSADF graph is proposed.

This technique removes the redundant dependencies from an FSMSADF graph. In order to validate the constructs described in this thesis, a tracing framework is designed. The constructs, the reduction technique and the tracing framework are contributions of this thesis to the model based design of SDRs.

8.2 Future work

Several extensions are possible to the approach described in this thesis. We propose these extensions as future work and they are as follows.

Modeling dynamism. The tasks present in a DVBT decoder have dynamic execution times. Figure 7.9 presents the average and worst case observed execution times of the blocks present in the DVBT decoder. The FFTS block in the ACQ scenario and the FFT block in the CFO scenario still have dynamism as the average and worst case observed time show a significant difference. An approach to model this dynamism is to add scenarios to the existing scenarios. Adding scenarios will lead to tighter timing analysis.

Modeling Out-of-Order command execution over the AXI interconnect. The AXI protocol offers advanced features like Out-of-Order command execution. The AXI models presented in Section 4.2 assume that a master does not issue a new command when an existing command is pending. For the DVBT decoder implementation over the MARS platform, this assumption is valid. However, it is of interest to model Out-of-Order command execution, for other applications executing over the MARS platform, because it influences the timing behavior of an application. For example, Out-of-Order execution may delay a command, compared to a later command, issued by the same master influencing the time required to communicate over the interconnect.

Model extraction from a trace. The trace extraction framework designed in this thesis traces an application. Given a trace, it is of interest to come up with a model which has a similar or detailed trace. In the presence of a system implementation, the extracted model may help in improving the existing model. It is a challenging task to extract a model which models the system completely as not all possible behaviors of a system might be observable during an execution.

Model multiple applications on the MARS platform. A DVBT decoder implementation is modeled in this thesis. The experiments conducted in Section 7.3 indicate that the throughput of the DVBT decoder exceeds the throughput constraint by $1445 - 825 = 620$ symbols/s. The excess resources can be used to execute an additional application on the MARS platform. An additional application can be modeled by extending the approach presented in this thesis. As a first step, the resources present in the MARS platform must be specified in the model. The SDF^3 toolkit allows such a specification. The second step is to model the applications that run on the MARS platform. Once modeled, multiple applications can be analyzed using the FSMSADF analysis techniques.

Reduction of FSMSADF graphs. It is of interest to automate the FSMSADF reduction technique. The automation can be performed using an existing Term Rewrite System (TRS) and by proving confluence. The definitions presented in Chapter 5 are analogous to term-rewrite rules. The terms corresponding to an HSDF graph can be simplified using a TRS, for example, by using Maude [CDE$^+$02]. The rewrite rules can be proved confluent using APPROVE [GTSKF03]. Confluent rewrite rules always lead to same normal form (one and only one HSDF graph).

Identification of transient and periodic execution. The tracing framework facilitates comparison of a model trace with a system trace. In case of SDRs, estimation of parameters (transient regime) may have completely different behavior compared to its normal operation (periodic regime) which is periodic. It is of interest to identify the start of periodic regime of an SDR by analyzing the system trace.

Bibliography

[AAG+11] O. Anjum, T. Ahonen, F. Garzia, J. Nurmi, C. Brunelli, and H. Berg. State of the art baseband DSP platforms for Software Defined Radio: A survey. *EURASIP Journal on Wireless Communications and Networking*, 2011(1):5, 2011.

[ASSG08] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 3–14. IEEE, 2008.

[AXI] AMBA3 AXI Protocol Specification v1.0 [Online accessed 15-07-12]. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022d/index.html`.

[BB04] R.V.D. Berg and H.S. Bhullar. Next generation Phillips digital car radios, based on a sea-of-dsp concept. *IEEE ISPC GSPx*, 2004.

[BBL08] H. Berg, C. Brunelli, and U. Lucking. Analyzing models of computation for software defined radio applications. In *System-on-Chip, 2008. SOC 2008. International Symposium on*, pages 1–4. IEEE, 2008.

[BCOQ92] F.L. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat. *Synchronization and linearity*, volume 2. Wiley New York, 1992.

[BHM+05] K.V. Berkel, F. Heinle, P.P.E. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal on Applied Signal Processing*, 2005:2613–2625, 2005.

[BR09] D. Brylow and B. Ramamurthy. Nexos: A next generation embedded systems laboratory. *ACM SIGBED Review*, 6(1):7, 2009.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[DVBa] DVB - EN 300 744 [Online accessed 4-02-12]. `http://www.etsi.org/WebSite/Technologies/DVB.aspx`.

[DVBb] DVBT - Wikipedia, the free encyclopedia [Online accessed 4-02-12]. `http://en.wikipedia.org/wiki/DVB-T`.

[Gei09] M. Geilen. Reduction techniques for synchronous dataflow graphs. In *Proceedings of the 46th Annual Design Automation Conference*, pages 911–916. ACM, 2009.

[Gei11] Marc Geilen. Synchronous dataflow scenarios. *ACM Trans. Embed. Comput. Syst.*, 10(2):16:1–16:31, January 2011.

[GTSKF03] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Aprove: A system for proving termination. *Rubio [Rub03]*, pages 68–70, 2003.

[JSS+11] R. Jordans, F. Siyoum, S. Stuijk, A. Kumar, and H. Corporaal. An automated flow to map throughput constrained applications to a MPSoC. *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, 18:47–58, 2011.

[Kum09] A. Kumar. *Analysis, design and management of multimedia multiprocessor systems*. PhD thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 2009.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[MAR] DSRC mobile WLAN component. `http://www.nxp.com/campaigns/connected-mobility/pdf/whitepaper_mk3_v05.pdf`.

[OH04] H. Oh and S. Ha. Fractional rate dataflow model for efficient code synthesis. *The Journal of VLSI Signal Processing*, 37(1):41–51, 2004.

[OPE] OpenComRTOS [Online accessed 14-04-12]. `http://www.altreonic.com/content/product-overview`.

[PRE] Preparation report of this thesis.

[SB09] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2009.

[SGB06] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF for free. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 276–278, 2006.

[SGM+11] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal. Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications. In *System on Chip (SoC), 2011 International Symposium on*, pages 14–21. IEEE, 2011.

[SGTB11] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404 –411, 2011.

[Stu07] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Technical University Eindhoven, 2007.

[TT97] M. Tofte and J.P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Yan09] F. Yang, 2009. Masters thesis: Static Analysis and Task Scheduling for Multi-mode Software-Defined Radio Applications.

[YWC] J. Yong, X. Wen, and G. Cyprian. Implementing a DVB-T/H Receiver on a Software-Defined Radio Platform. *International Journal of Digital Multimedia Broadcasting*, 2009.

Appendix A

Reduction of AXI models

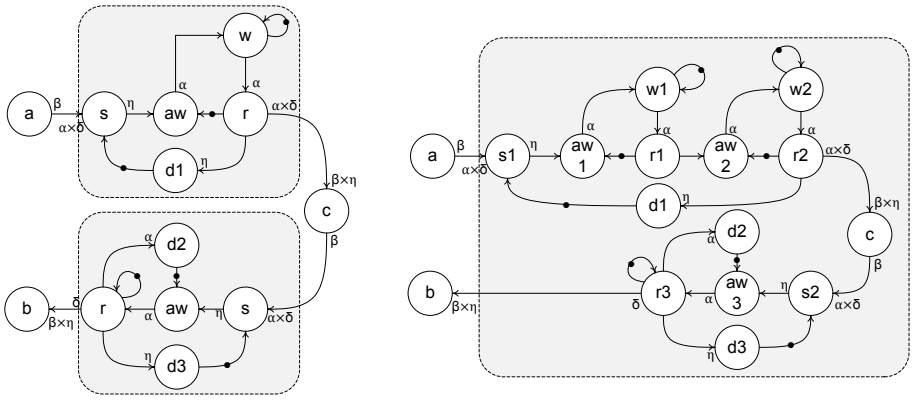

Figure A.1: Unoptimized AXI read-write model (a) and AXI read-write model over bridge (b).

The optimized AXI models were presented in Section 4.2. They were optimized from the models presented in Figure A.1 using the reduction approach presented in Chapter 5. This reduced the number of initial tokens present in the read-write model shown in Figure A.1(a) from six initial tokens to three initial tokens in the model presented in Figure 4.5(a). This reduces the initial tokens significantly in the automatically generated binding aware models. Similarly, Figure A.1(b) presents the unoptimized read-write model over the bridge. The reduction approach reduced the number of initial tokens from eight initial tokens to three initial tokens. However, the unoptimized models are more intuitive and near to the behavioral models of the AXI read-write bursts presented in Figure 4.4. It is of interest to specify intuitive models and the analysis tools automatically simplify the models using the reduction approach.

Appendix B

Modeling packet resizers in a DVBT decoder

name	count	name	count
src	17408	dfe isr	17408
acquisition	17408	agc	17408
fft	17408	tps sync	17408
equalization	17408	upl	17408
deqam	17408	flora t1	17408
packet resizer 1	193536	desc sync	193536
flora t2	193536	ts filter	193536
packet resizer 2	387072	usb out	8883
sink	8883		

Table B.1: The repetitions of the actors present in a DVBT decoder when packet resizers are modeled.

A construct to model packet resizers in an SDR is described in Section 4.1. It models resizing a of packet between two tasks when the tasks use packet pools. However, the models described in Chapter 7 do not model packet resizers. It is because the decoder requires several OFDM symbols to flush out the data buffered in a packet resizer. The SDF graph shown in Figure 7.1 was extended to model packet resizers. In order to model the packet resizers, the *packet resizer 1* and *packet resizer 2* actors were replaced by the packet resizer constructs as specified in Section 4.1. Table B.1 presents the repetition vector of the extended SDF graph. The repetition vector indicates that 17408 OFDM symbols are required to complete one iteration of the SDF graph. However, the approach used in this thesis provides guarantees for a single OFDM symbol. It requires an SDF graph iteration to complete with a single OFDM symbol. This limits modeling the packet resizers present in a DVBT decoder. In order to model the resizers, standard definition of *throughput constraint* requires an extension.