

# **Static Analysis and Task Scheduling for Multi-mode Software-Defined Radio Applications**

by  
**Feiteng Yang**

Document Type: Master thesis Eindhoven University of Technology (TU/e)  
MSc Work: Performed at ST-Ericsson, Technology & Tools, Advanced  
R&D, HTC, Eindhoven  
Period of Work: November 1<sup>st</sup>, 2008 – August 18<sup>th</sup>, 2009  
Supervisor: Prof.Dr.Ir. C. H. van Berkel  
Tutor: M.Sc. Orlando. Moreira (ST-Ericsson)



## Abstract

Synchronous Data Flow (SDF) Graphs can be scheduled on a multi-processors architecture with hard real-time constraints. The Heracles tool, designed by the researchers from ST-Ericsson, Eindhoven, can perform automatic scheduling and rigorous timing analysis of SDF graphs. However, SDF is too restrictive to model the baseband processing of radio applications.

This thesis describes the extension of Heracles tool to mode controlled data flow graphs to express data-dependent concurrency in software-defined radio applications. A new scheduling strategy, the combination of quasi-static and TDM/round robin is designed and implemented to schedule the mode controlled data flow graphs. Two new timing analysis methods, self-timed scheduling and static periodic scheduling, are introduced to do the timing analysis of the scheduled data flow graphs, and implemented on Heracles tool.



## Preface

This document is my Master's thesis and is the result of a graduation project to obtain the degree of Master of Science in Computer Science with a specialization in Embedded Systems. This work is carried out at ST-Ericsson Advanced R&D, Eindhoven.

I would like to thank: Kees van Berkel for his supervision in ST-Ericsson. He also helps me with the knowledge of software-defined radios. I also would like to thank Orlando Moreira for his help in understanding the SDF graph, scheduling strategies, timing analysis methods, and OCaml language. Orlando Moreira spent much time reviewing and correcting my thesis.

Feiteng Yang  
Eindhoven, August 2009

## Legal Information

© Copyright ST-Ericsson, 2009. All Rights Reserved.

### Disclaimer

The contents of this document are subject to change without prior notice. ST-Ericsson makes no representation or warranty of any nature whatsoever (neither expressed nor implied) with respect to the matters addressed in this document, including but not limited to warranties of merchantability or fitness for a particular purpose, interpretability or interoperability or, against infringement of third party intellectual property rights, and in no event shall ST-Ericsson be liable to any party for any direct, indirect, incidental and or consequential damages and or loss whatsoever (including but not limited to monetary losses or loss of data), that might arise from the use of this document or the information in it.

ST-Ericsson and the ST-Ericsson logos are trademarks of the ST-Ericsson group of companies or used under a license from STMicroelectronics NV or Telefonaktiebolaget LM Ericsson.

All other names are the property of their respective owners.

# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	Background	11
1.2	Software-Defined Radio	11
1.3	Multi-Processors System on Chip	12
1.4	Synchronous Data Flow Graph	13
1.4.1	Directed Graph	14
1.4.2	Data Flow Graph	14
1.4.3	Synchronous Dataflow Graph	15
1.4.4	Homogeneous Synchronous Dataflow Graph	15
1.4.5	Analytical Properties of SDF Graphs	16
1.4.6	SDF Conversion to HSDF	17
1.5	Application Model	17
1.6	Real-Time Constraints	18
1.7	Scheduling Strategy	18
1.8	Mapping Model	19
1.9	Building the Analysis Model	20
1.9.1	Modeling TDM Scheduling	20
1.9.2	Modeling Static-Order Scheduling	21
1.9.3	Modeling Combination of Static-Order and TDM	21
1.10	Timing Analysis	22
1.10.1	Maximum Cycle Mean	22
1.10.2	Self-timed Schedules	23
1.10.3	Static Periodic Schedules	23
1.11	Heracles	24
1.11.1	Scheduling and Analysis Procedure	24
1.12	Structure of Thesis	26
<b>2</b>	<b>Mode-Controlled Data Flow</b>	<b>27</b>
2.1	MCDF Overview	29
2.2	Data-Dependent Actors	31

2.3	Tunnel Representation	31
2.4	The Construction Rules of MCDF Graph	33
<b>3</b>	<b>Problem Description</b>	<b>35</b>
3.1	Contribution of This Thesis	36
<b>4</b>	<b>Overall Approach</b>	<b>37</b>
4.1	Timing Analysis	37
4.1.1	Mode Sequence	37
4.1.2	Self-timed Schedule with Mode Sequence	37
4.1.3	Static-periodic Schedule with Mode Sequence	39
4.1.4	Self-Timed Scheduling versus Static Periodic Scheduling	42
4.2	Building the Analysis Model	43
4.2.1	Modeling Quasi-static Ordering Schedule	43
4.2.2	Modeling Round Robin Schedule	45
4.2.3	Modeling Combination of Quasi-static Ordering and TDM/Round Robin Schedule	46
4.3	MCDF Scheduling Strategy	48
4.3.1	Adapting the Original Scheduler	49
4.3.2	Adding Mode Switch to Processor	49
<b>5</b>	<b>Analysis Tool Design</b>	<b>53</b>
5.1	Objective Caml	53
5.2	Heracles Analysis Tool	53
5.2.1	Graph File	53
5.2.2	Architecture File	55
5.2.3	Mode Sequence File	56
<b>6</b>	<b>Implementation</b>	<b>57</b>
6.1	MCDF Graph Scheduling Procedure	57
6.2	Self-timed Scheduling with Mode Sequence	58
6.2.1	Symbolic Simulation Procedure	58
6.2.2	Alternative to Building Sub Graph	60
6.2.3	Symbolic Simulation Modification with Mode Sequence	61

6.3	Static-periodic Schedule with Mode Sequence	63
6.3.1	The Procedure of Solving Linear Programming Problem	63
<b>7</b>	<b>Experiment and Results</b>	<b>67</b>
7.1	Case 1 DVB-T MCDF Graph	67
7.2	Case 2 Wireless Lan MCDF Graph	70
7.3	Case 3 Corner Example	74
7.4	Summary	76
<b>8</b>	<b>Conclusions and Future Work</b>	<b>79</b>
	<b>References</b>	<b>80</b>
	<b>Acronyms and Terms</b>	<b>81</b>
	<b>Appendix A: MPSOC PLATFORM FILE</b>	<b>82</b>
	<b>Appendix B: DVB-T Graph File</b>	<b>83</b>
	<b>Appendix C: Wireless LAN Graph File</b>	<b>84</b>
	<b>Appendix D: Corner Example Graph File</b>	<b>85</b>



# 1 Introduction

## 1.1 Background

Many multimedia applications are integrated on mobile terminals. Cellular phones nowadays not only serve as a tool to make phone calls, but also perform some other tasks like web browsing and media playing. The applications, which involve baseband radio processing, may start or stop at any time and run simultaneously.

At the Advanced R&D group of ST-Ericsson in Eindhoven, a project is being carried out to study the usage of embedded multiprocessor systems to process multiple Software-Defined Radios (SDRs) simultaneously. These SDRs may start/stop execution at any time and each one has its own rate of operation. The Heracles tool is designed to schedule the SDRs on a target Multi-Processor System on Chip and do the timing analysis of the scheduling. In Heracles, the SDR jobs are represented as Data Flow (DF) graphs. Currently, one restriction of Heracles is that the amount of data produced and consumed per activation of each processing task must be independent of the values of the input data.

## 1.2 Software-Defined Radio

A Software-Defined Radio (SDR) is a wireless communication system where components (e.g. mixers, filters, amplifiers, modulators/demodulators, detectors. etc.) are implemented using software on the embedded computing devices instead of using hardware. [2]

In the market of high-end mobile phones, multiple standards are required to be implemented on a single device. These mobile phones should support all kinds of cellular networking standards, wireless internet standards, mobile television standards, and peer-to-peer communication standards. The cellular networking standards include Global System for Mobile Communication (GSM), Wideband Code Division Multiple Access (WCDMA), and High-Speed Downlink Packet Access (HSDPA). The wireless internet standards include 802.11a/b/g. The Mobile Television standards include Digital Video Broadcasting-Handheld (DVB-H). The peer-to-peer standards include Bluetooth.

**Table 1 Mobile Phone Standards**

Standards	Wi-Fi ultra-wideband	802.11a/b/g	802.11n	Wireless broadband (WiBro)	Mobile WiMax	3G LTE (Cellular WAN)	Digital Video Broadcasting-Handheld	Digital Video Broadcasting-Terrestrial
Application	High-speed local interconnect, wireless USB	Medium-speed LAN	High-speed LAN	Mobile wireless access	Mobile wireless access	Mobile Data/Voice	Mobile TV	Mobile TV
Range	10m	80m	50-150m	1-5km	1-5km	1+km	Broadcast	Broadcast
Rate	480Mbps	11 Mbps(b) 54 Mbps(a/g)	100-600 Mbps	3-50 Mbps (downlink)	63 Mbps (downlink)	100 Mbps (downlink)	384 Kbps	7 Mbps
Frequency	3.1- 10.6 GHz	2.45/5.8 GHz	2.45/5.8 GHz	2-6/2,3 GHz	2-6/2,3 GHz	1.25/2.2/5/10/2 GHz	0.8 MHz, 1.6 GHz	0.8 MHz, 1.6 GHz

Table 1 [2] gives an overview of radio standards, including wireless access, mobile TV, mobile data/voice, and local connectivity. If all these standards are implemented with dedicated hardware, several different hardware devices should be assembled in the mobile phone which is very costly to realize. However, if the functions are implemented with software-defined radios, changing the service type or the modulation protocol can be done simply by selecting and launching the appropriate computer programs. Furthermore, there may be several applications running simultaneously.

The ultimate goal of SDR engineers is to provide a single radio transceiver capable of playing the roles of cordless telephone, cell phone, wireless videoconferencing unit, Global Positioning System (GPS) unit, and other functions still in the realm of science fiction, operable from any location on the surface of the earth, and perhaps in space as well.

## 1.3 Multi-Processors System on Chip

The Multi-Processor System on Chip (MPSoC) has emerged in the past decade as an important class of Very Large Scale Integration (VLSI) systems. An MPSoC combines multiple processing elements, memory and other digital functions (like I/O ports). [3] All these components are linked to each other by a bus or Network-On-Chip (NOC) interconnect.

As an embedded hardware platform, MPSoC provides a good balance between cost, power efficiency, and flexibility. Typically, Embedded MPSoCs are heterogeneous with general processing elements (such as ARM processor cores) and application-specific processing elements (such as a Turbo decoder). The flexible functions which vary according to different applications will be implemented on general processors. The functions that will be the same for every application will be implemented on application-specific processors which bring low power cost. In order to allow maximum flexibility at the lowest cost, applications share computation, storage, and communication resources.

Mobile terminals may run several applications simultaneously. In order to design a flexible hardware platform with low power cost, embedded MPSoCs are employed. Figure 1 shows a template of an MPSoC architecture for Software-Defined Radio. The MPSoC template contains three processors and Input/Output ports linked to each other by bus. Each processor has its own local memory which is represented as "MEM" in the figure. All the processors are connected via a Network Interface (NI) to the bus. Each NI has a number of input and output queues with limited buffer capacity. There are three types of processors which are Embedded Vector Processor (EVP) [1], ARM and ASIP for software decoding.

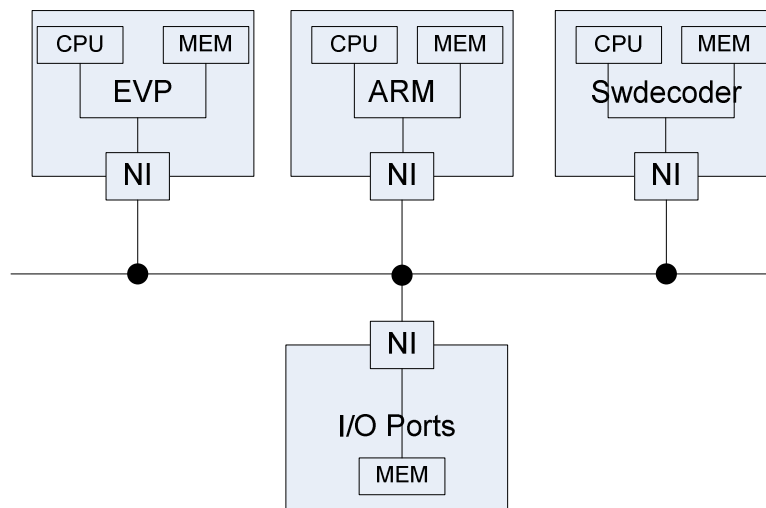


Figure 1 MPSoC Architecture

## 1.4 Synchronous Data Flow Graph

In the context of the ST-Ericsson Software-Defined Radio Project, data flow graphs are employed to model the SDRs. In this section, we introduce the graph definitions and terminology that we will need in the remainder of this document. A graph  $G$  is an ordered pair  $G = (V, E)$ . Here  $V$  is a set of **nodes** and  $E$  is a set of **edges**.

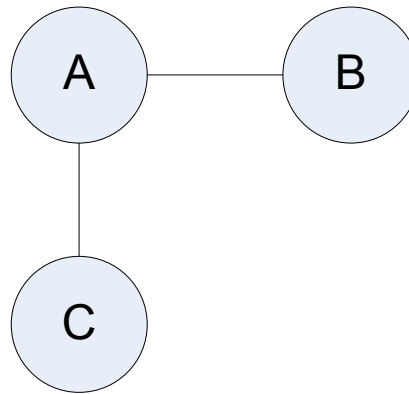


Figure 2 Normal Graph

### 1.4.1 Directed Graph

A directed graph is a graph  $G = (V, E)$  where every edge is an ordered pair  $(v_1, v_2)$ , where  $v_1, v_2 \in V$ . For an edge  $e = (v_1, v_2) \in E$ , we say that  $e$  is directed from  $v_1$  to  $v_2$ ,  $v_1$  is the source of  $e$ , and  $v_2$  is the sink of  $e$ . In a directed graph, we can not have two or more edges with both the same source and sink. [7]

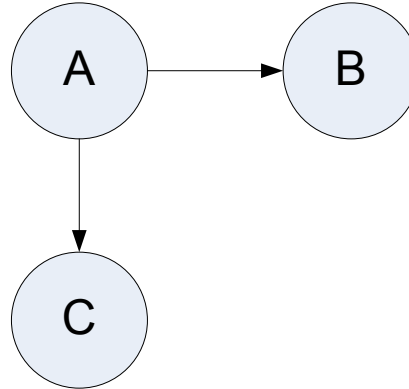


Figure 3 Directed Graph

### 1.4.2 Data Flow Graph

A data flow graph is a directed graph (See **Figure 3**), where the actors represent computation and edges represent First-In-First-Out (FIFO) queues. Data is transferred between actors in tokens of fixed size. Edges can have an initial number of tokens stored in them at execution start time; this number is often referred to as the delay of the edge.

When an actor is activated by data availability, it is said to be **fired**. The **firing** of an actor comprises three steps. First, the actor consumes tokens from its inputs. Then the actor performs the computation. Finally, the actor produces tokens to all its outputs.

### 1.4.3 Synchronous Dataflow Graph

Synchronous Data Flow (SDF) graph (See **Figure 4**) is a restricted dataflow model proposed by Lee and Messerschmitt [6]. In the SDF model, each actor consumes/produces the same fixed-number of tokens in each input/output edge in every firing. An actor can fire whenever all its input edges have enough tokens for a firing of the actor to consume. There may be data dependencies across iterations. These dependencies are modeled as **initial tokens** on the FIFO edges. The delay of an edge is represented by a function  $d(e)$  that for an edge  $e$  returns the initial number of tokens stored in that edge. The classical SDF model is un-timed. However, there is an extension to SDF in which a fixed execution time is associated with each actor [7]. This extension makes the model amenable to timing analysis. We define the function  $e(a_i)$  as the execution time of actor  $a_i$ .

**Definition:** An **iteration** of an SDF graph is a sequence of actor firings such that the graph is brought back to its initial token distribution.

Take the SDF graph shown in Figure 4 as an example. An iteration of this SDF graph is a sequence of firings of the actors in this graph such that the graph again has one token in edge (A, C) and zero token in edge (A, B).

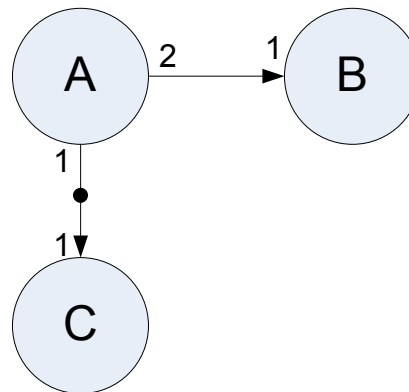


Figure 4 Synchronous Dataflow Graph

### 1.4.4 Homogeneous Synchronous Dataflow Graph

An SDF graph in which every actor consumes and produces only one token from each of its inputs and outputs is called a Homogeneous Synchronous Dataflow (HSDF) graph (See **Figure 5**). [7]

An HSDF graph actor fires when it has one or more tokens on each of its input edge. One token from each of the actor's inputs is consumed. When it finishes a firing, one token is produced on each of its output edges. In the graphical representation of HSDF graphs, one can omit the productions and consumptions, as these are none to be all equal to 1, as shown in **Figure 5**.

An **iteration** of a HSDF graph is that all the actors in this graph are fired once. The function  $s(i,k)$  denotes the time at which the  $k^{th}$  iteration of actor  $a_i$  is fired. The iteration number is counted from 0, so  $k^{th}$  iteration means the  $(k+1)^{th}$  execution of actor  $a_i$ . Let  $d(i,j)$  denote the delay number of edge from actor  $a_i$  to actor  $a_j$ , and let  $e(a_i)$  denote the execution time of actor  $a_i$ . With HSDF, we have the basic rule below:

$$s(j,k) \geq s(i,k - d(i,j)) + e(a_i) \quad (1-1)$$

The rule means that for an edge with source actor  $a_i$  and sink actor  $a_j$ , the start time of actor  $a_j$  should be later than the finishing time of actor  $a_i$  of delay number of iterations before.

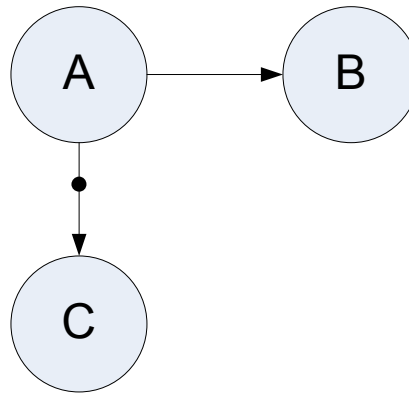


Figure 5 HSDF Graph

### 1.4.5 Analytical Properties of SDF Graphs

An SDF graph can be compactly represented by its topology matrix  $\Gamma$ . The matrix represents the graph structure, and contains one column for each vertex, and one row for each edge in the SDF graph. The value of entry  $(i, j)$  in the matrix corresponds to the number of tokens produced by the actor numbered  $j$  onto the edge numbered  $i$ . If the actor  $j$  consumes tokens from the edge  $i$ , the value of entry  $(i, j)$  is negative. Also, if the actor  $j$  neither produces nor consumes tokens from the edge  $i$ , the value of entry  $(i, j)$  is set to be zero. For example, the topology matrix of the graph in **Figure 4** is:

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

Where the actors  $A$ ,  $B$  and  $C$  are numbered as 1, 2, and 3, and the edges of  $(A,B)$  and  $(A,C)$  are numbered as 1, 2 and 3.

**Definition:** The repetitions vector  $q$  for an SDF graph with  $s$  actors is a column vector of length  $s$ , with the property that if each actor  $i$  is invoked a number of times equal to the  $i_{th}$  entry of  $q$ , then the number of tokens on each edge of the SDF graph remains unchanged. Furthermore,  $q$  is the smallest integer vector for which this property holds.

**Theorem:** The repetitions vector of an SDF graph with consistent sample rates is the smallest integer vector in the null space of its topology matrix. That is,  $q$  is the smallest integer vector such that

$$\Gamma q = 0 \quad (1-3)$$

For the example above,  $q$  is equal to  $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ . More details and proof can be found in [7].

The repetitions vector indicates how many times each actor should execute to bring the graph back to its initial token distribution. In this example, to return the graph back to its initial token distribution, i.e. zero tokens in the edge  $(A,B)$  and one token in the edge  $(A,C)$ , one execution, actor  $A$  and  $C$  should be executed once and actor  $B$  should be executed 2 times.

With the repetitions vector, we can define an **iteration** of an SDF graph to be a sequence of actor firings such that each actor in the graph executes a number of times equal to its repetition vector entry.

## 1.4.6 SDF Conversion to HSDF

In order to analyze an SDF graph's throughput and schedule it, it is convenient to first perform a conversion to its equivalent HSDF. The resulting equivalent HSDF graph has as many actors as specified by the original dataflow graph repetitions vector  $q$  and each of these actors represents an execution of the original actor. Detailed SDF-to-HSDF conversion algorithm can be found in [7].

## 1.5 Application Model

In our application model, we define a **job** as a particular multi-processor program that can be started or stopped independently of other jobs. It can contain several processing **tasks**, which can communicate data with each other. Each job is represented as a dataflow graph. In the graph, each node represents a task and each edge represents a FIFO channel.

## 1.6 Real-Time Constraints

SDR jobs typically have real-time requirements. Real-time applications can be classified depending on how strict their real-time requirements are. For example, the deadlines of **hard real time** jobs must not be missed, while the deadlines of **soft real time** jobs can be missed, but the rate of the misses must be kept below a specific value.

There are also two essentially different types of deadlines. Some applications are required to generate a certain number of outputs with a time period. This is referred to as a **throughput** requirement. For example, a television signal receiver must display a certain number of frames per second to avoid flickering.

There are also applications like online gaming where the temporal requirement is that the system responds to a button click within 10 milli-seconds. This is a **latency** requirement. Latency is the time interval between an input event and an output event. For example, we can measure the latency as the difference between the start times of two specific firings of two actors. [5]

There are also examples having both throughput and latency requirements. Like the video conference meeting, it is required displaying a good quality video and sending the voice in time. The requirement of displaying is a throughput requirement. The requirement of voice sending in time is a latency requirement.

## 1.7 Scheduling Strategy

This section describes the scheduling strategy. Scheduling an application graph involves assigning actors in the graph to processors, ordering the execution order of these actors on each processor, and determining when each actor fires such that all data precedence constraints are met. Each of these three steps may be performed either at run-time (dynamic strategy) or at compile-time (static strategy).

With the assumption that the assignments to the processors of the actors are given before scheduling, a fully dynamic schedule is avoided. The researchers from ST-Ericsson designed a scheduling strategy that allows a heterogeneous MPSoC to handle a dynamic mix of hard-real-time jobs which can start or stop independently. To solve this problem, a combination of Time Division-Multiplex (TDM) schedule and static-order of actors per processor is applied. [4]

As jobs may start or stop independently, the scheduler has to decide whether a new requested job should be started or not during run time. The scheduler tries to map the actors of the job graph on the processor. The resource requirement including computation period and memory size will be analyzed according to the real-time requirement of the job. If the scheduler finds enough resource for this job, the job is scheduled. Otherwise, the job is denied or asked to have lower real-time requirement.

If a job is accepted to execute on the platform, the scheduler then applies the combination of TDM and static-ordering schedule to the job graph. Assuming that a TDM wheel period  $P$  is implemented on the processor and that a time slice with duration  $S$  is allocated for an actor  $a_p$ , such that  $S \leq P$ . The actor starts to execute when the time slice arrives. If the actor cannot finish execution in one time slice, it has to wait for  $P-S$  time to get the resource again.

A group of actors may be mutually exclusive as they are from the same single-delay cycle. Here, mutually exclusive means when one of the actors from the group is executing on the processor, the other actors can not execute. If they are mapped on the same processor, allocating different slices to each actor wastes resources. If they share the same time slice, each actor in the group can use the whole time slice when enabled. A more detailed discussion of this can be found in [4]. The static ordering scheduling is used to order these actors even in some case where they do not have data dependencies in between.

Mixing static ordering and TDM scheduling is to take the statically ordered actors as a big actor executing on the processor. TDM scheduler assigns time slice to the big actor instead of to each actor. For example, we have actors  $A$  and  $B$  which are both mapped on processor  $P$ . Instead of assign time slices individually to actor  $A$  and actor  $B$ , we take these two actors as a big actor  $AB$ . And the actor  $AB$  is assigned a time slice by the TDM scheduler.

## 1.8 Mapping Model

After mapping the actors of an SDF graph on the given processors, we build the timing analysis model by calculating the worst case response time of the actors that corresponds to that mapping.

Several definitions are defined below to help building the timing analysis model. We define **execution time**  $e(a)$  to be the time it takes for the actor to execute on the processor without resources contention. After an actor is enabled, the actor may not get the resource immediately. The time between when an actor is enabled and when it starts to execute is defined as **arbitration time**  $a(a)$ . We calculate the worst case for this arbitration time. After an actor starts to execute, the execution may be preempted. So we define **processing time**  $p(a)$  as the time from the moment an actor starts to execute until it finishes on the processor. Here we have:

$$p(a_i) \geq e(a_i) \quad (1-4)$$

The **response time**  $r(a)$  counts from the moment that an actor is enabled until it finishes. So we have:

$$r(a_i) = a(a_i) + p(a_i) \quad (1-5)$$

## 1.9 Building the Analysis Model

This section describes the modeling of scheduling strategy. The analysis model built with the assumption of worst case will be used in timing analysis. To build the analysis model, the response time, arbitration time and processing time of each actor will be calculated according to the scheduler and the actor's execution time.

### 1.9.1 Modeling TDM Scheduling

Under TDM scheduling, the worst-case execution time of the actors represents the worst-case response time when that scheduler is used. This response time counts from the moment when the actor meets its enabling condition until this actor finishes firing. Assuming that a TDM wheel period  $P$  is implemented on the processor and that a time slice with duration  $S(a_i)$  is allocated for the actor  $a_i$ , such that  $S(a_i) \leq P$ . Two different effects will affect the actual response time of actor  $a_i$ . The first one is the arbitration time which counts from the moment that  $a_i$  is enabled until the moment that resource is granted to  $a_i$ . The worst case would be that  $a_i$  is enabled just after the slice  $S$  for  $a_i$  finishes. In the worst case, actor  $a_i$  has to wait for  $P-S$  time. The arbitration time is calculated in the equation (1.6).

$$a(a_i) = P - S(a_i) \quad (1-6)$$

The second effect is when the slice assigned to the actor  $a_i$  is smaller than the execution time of the actor  $a_i$ . So the actor  $a_i$  can not finish execution in one time slice. Therefore, the processing time it takes to fire the actor  $a_i$  is equal to:

$$p(a_i) = \lceil e(a_i) / S(a_i) \rceil \cdot P - (e(a_i) \bmod S(a_i)) \quad (1-7)$$

The total response time of actor  $a_i$  will be the sum of the two times above:

$$r(a_i) = (P - S(a_i)) \cdot \lceil e(a_i) / S(a_i) \rceil + e(a_i) \quad (1-8)$$

After all the response times are calculated for the actors, timing analysis methods can be applied to the graph to evaluate whether the real-time constraints are met or not.

### 1.9.2 Modeling Static-Order Scheduling

Assume that a set of actors  $A = \{a_1, a_2, \dots, a_n\}$  are mapped to the processor  $P$ . Extra precedence constraints are added to this set of actors in the way that  $a_k$  will be the first to execute, followed by  $a_m$ , and so on up to  $a_l$ . After the last actor is executed, the sequence is reset, and execution order restarts for the next iteration of the graph. This strategy is called static-order scheduling. [4]

Any extra dependency added by static order scheduling can be represented as an edge without token in the graph. From the last actor to the first actor, an edge with one token is added to represent that when the current iteration is finished, it restarts from the first actor in that static-order for the next iteration.

### 1.9.3 Modeling Combination of Static-Order and TDM

Instead of assigning a time slice to each actor, we assign the time slice to a group of actors and statically order these actors. To model this combination, extra edges are added between the actors in the group to represent the static-ordering. After one actor is mapped onto a processor, the system groups the actor with other actors mapped onto the same processor. The serialization of the actors in a group is represented in the data flow model delay-less edges. An edge with one delay is added for each group from the last actor of the group to the first actor of the group. **Figure 6** shows an example of combination of static ordering and TDM scheduling. In **Figure 6**, actors C and B are grouped together. Extra delay-less edge from actor C to actor B is added to represent the static execution order. One edge with one token is added from actor B to actor C.

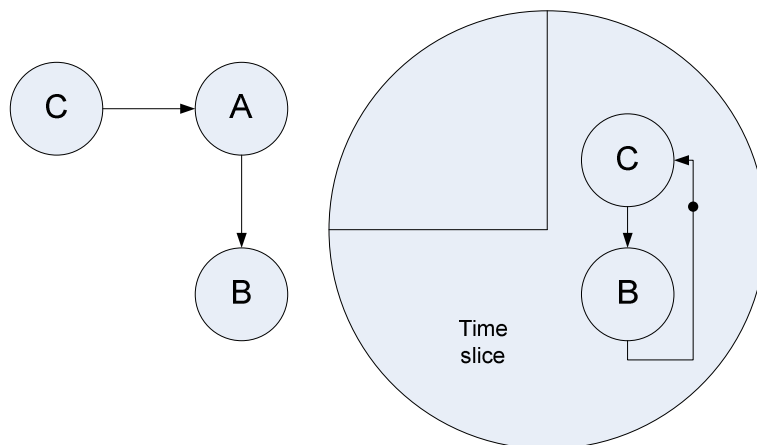


Figure 6 Combination of Static Ordering and TDM

As illustrated in [4], the timing analysis model is built with response model. The **time valuation** of each grouped actor will be set as the **processing time**. In order to model the arbitration time, an extra actor is created if this group of actors has input edges from outside this group. Take the mapping example shown in the left of **Figure 7**, actors *A* and *B* are grouped together and actor *A* has an input edge from another group. A time slice *S* is assign to the group of actor *A* and actor *B*. The graph shown in the right part of **Figure 7** is the response model of the group. An extra actor  $a_s$  is added to represent the arbitration time of this group. The time of the actor  $a_s$  is set to be  $P - S$ . The **time** for actor *A* is set to be its processing time:

$$\lceil e(A) / S \rceil \cdot P - (e(A) \bmod S)$$

The **time** for actor *B* is set to be its processing time:

$$\lceil e(B) / S \rceil \cdot P - (e(B) \bmod S)$$

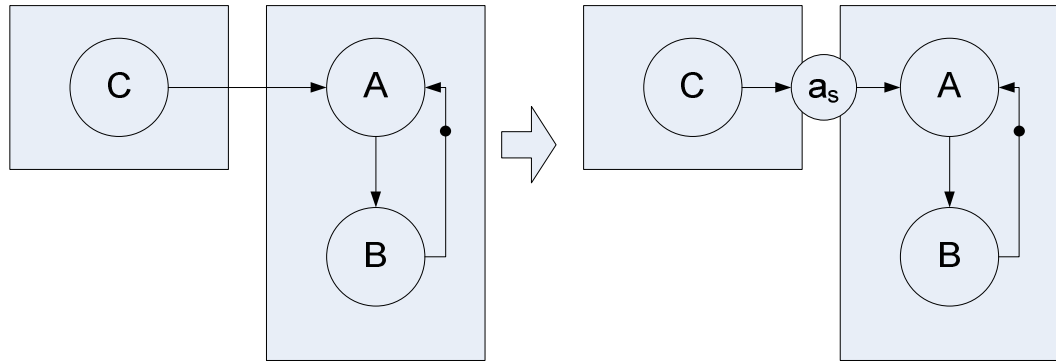


Figure 7 Analysis Model of combination of Static Ordering and TDM

## 1.10 Timing Analysis

The analysis model is built according to the scheduling strategy. In this section, 3 timing analysis methods are explained. The goal of timing analysis is to find whether the scheduling meets the real-time requirement or not.

### 1.10.1 Maximum Cycle Mean

In order to calculate the throughput of a timed SDF graph, the concept of Maximum Cycle Mean (MCM) is introduced. Assume there is a timed SDF graph:  $G = (V, E)$ . A directed cycle  $c$  in the graph  $G$  is a path from a node to itself that traverses each node in it only once [9]. The cycle mean of cycle  $c$  in graph  $G$  is calculated in the equation (1.9):

$$\mu(c) = \frac{\sum_{a_i \in V(c)} e(a_i)}{\sum_{e \in E(c)} d(e)} \quad (1-9)$$

Here  $V(c)$  and  $E(c)$  denote, respectively, the sets of nodes and edges belonging to the directed cycle  $c$ . And  $d(e)$  denotes the delay value of edge  $e$ . The maximum cycle mean of the graph  $G$  is calculated in the equation (1.10):

$$\mu(G) = \max_{c \in C(G)} \left( \frac{\sum_{a_i \in V(c)} e(a_i)}{\sum_{e \in E(c)} d(e)} \right) \quad (1-10)$$

Where  $C(G)$  is the set of cycles in graph  $G$ .

The inverse of the *MCM* provides the minimum guaranteed throughput. By assigning the amount of *MCM* as the period of the execution, the start times of all the actors in each iteration can be predicted.

## 1.10.2 Self-timed Schedules

The generation of a self-time schedule can be used as an alternative way to calculate the throughput. A self-timed schedule (STS) is also known as an as-soon-as-possible schedule. By applying STS to an SDF graph, the firing of each actor starts immediately if there are enough tokens in all its input edges.

The Worst Case Self-Time Schedule (WCSTS) is a schedule of an SDF graph such that all the actors in the graph take worst case time to execute. The WCSTS of an SDF graph has a property: after a transition phase of  $K$  iterations, it will reach to a periodic pattern.[8] The period is  $N(G) \cdot \mu(G)$  time units, where  $N(G)$  is the minimum among the sums of delays of the critical cycles of the graph. The schedule of the periodic pattern is:

$$s(i, k + N(G)) = s(i, k) + N(G) \cdot \mu(G) \quad k \geq K(G) \quad (1-11)$$

During the periodic execution,  $N(G)$  firings of actor  $i$  happen in  $N(G) \cdot \mu(G)$  time units, which means that the throughput of the graph is  $1 / \mu(G)$ . For the period of transient phase, that is  $k < K(G)$ , the schedule can be derived by symbolic simulation with worst case execution time actors which will be explained in Section 6.

## 1.10.3 Static Periodic Schedules

Applications may require that each iteration finishes within a given period time. Then we can apply static periodic scheduling (*SPS*) to the application with the desired period. An *SPS* of an SDF graph is a schedule such that for all nodes  $i \in V$ ,

$$s(i, k) = s(i, 0) + T \cdot k \quad (1-12)$$

Here  $T$  is the desired period of the SPS. The SPS can be represented uniquely by the values of  $s(i, 0)$  for all nodes.

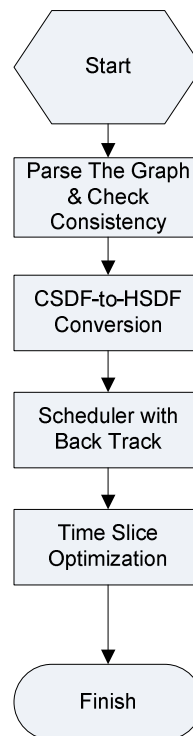
It is proved in [8] that for any SDF graph; it is always possible to find an SPS schedule, as long as  $T \geq \mu(G)$ . If  $T < \mu(G)$ , then no SPS schedule exists.

## 1.11 Heracles

This section describes the Heracles tool designed by the researchers from ST-Ericsson. Heracles takes an SDF graph file (including graph information, actor mapping, and real-time requirements) and a multiprocessor platform file description as inputs. As outputs, if run simply for analysis, Heracles verifies if the timing constraints are feasible. If run as a scheduler, Heracles can produce a partial mapping of the graph to the multiprocessor template, where actors are assembled in statically-ordered groups, and for each statically ordered group, a scheduler budget (i.e., an amount of required resources for a specific single-core scheduler) is associated. The budget also includes sufficient buffer sizes for all inter-group FIFOs, which Heracles can also compute.

### 1.11.1 Scheduling and Analysis Procedure

This subsection describes the scheduling and analysis procedure carried out by Heracles (as shown in Figure 8).

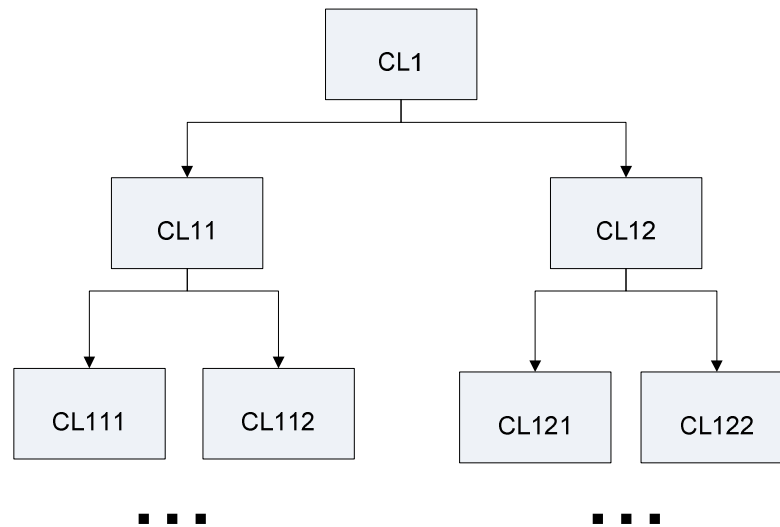


*Figure 8 Heracles Scheduling Procedure*

The graph and multiprocessor platform are stored in text files. The tool first parses the graph into the system. The input graph is then checked in several aspects listed below:

- Whether all the actors are mapped on one of the processors;
- Whether the type of the processor that an actor is mapped onto matches the type of the actor or not;
- Whether the slice time assigned to each actor overflows the whole period of the processor this actor mapped on;
- The input file may require that some actors are placed in the same group, which means that these actors should be mapped on the same processor. The tool then checks whether these actors are required to run on the same processor/processor type or not.

A correct input graph will then be converted into an HSDF graph. After having a correct HSDF graph, the procedure enters the scheduling loop with backtracking. The actors without input or with only delay inputs are the candidates ready to be fired. The system initializes the candidate list with these ready actors and enters the backtracking phase. We show the backtracking procedure with the HSDF graph example depicted in Figure 5. In the graph, the actor  $C$  is ready to be fired at the beginning as it has no delay-less inputs. Assume there are two processors,  $P_1$  and  $P_2$ , to be mapped on for the actors, and each actor can be mapped on any processor. So we create a Candidate List ( $CL$ ) in the form of  $\{(C, P_1), (C, P_2)\}$ . The system maps actor  $C$  according to the head of  $CL$ . After mapping, the analysis model is built to calculate MCM. If the MCM meets the real-time requirement, the  $CL$  is updated with the possible mappings of newly activated actors. Then it falls into the loop of scheduling again. If the requirement is not met, the system undoes the mapping and back tracks to the previous states. The method of back tracking is show in Figure 9. The  $CL$  is the candidate list. We start with  $CL_1$ , which has two elements. By doing mapping according to  $CL_1$ 's first element, an updated  $CL_{11}$  is obtained. As the same trick,  $CL_{11}$  can be extended to  $CL_{111}$  and  $CL_{112}$ . If both  $CL_{111}$  and  $CL_{112}$  fail to guarantee the requirement, which means that the branch of  $CL_{11}$  fails, the system then back tracks to the state of  $CL_1$  and try the second element of  $CL_1$ . The second element will lead to some other sub branches.



*Figure 9 Backtracking*

When the candidate list is empty, the system then checks whether all the actors are mapped or not. If yes, the system tries to reduce the slice time for each group and leads to a solution. If not all the actors are mapped, the scheduler fails.

For computing the MCM, the analysis model is built according to the combination of TDM and static-order scheduling. After one actor is mapped on a processor, the system groups the actor with other actors mapped on the same processor. The actors are grouped with delay-less edges. An edge with one delay is added for each group from the last actor of the group to the first actor of the group. Response actors are added if the mapped actor has inputs from other processors. The executing time are set for both mapped actor and added response actors.

## 1.12 Structure of Thesis

In Section 2, the Mode-Controlled Data Flow (MCDF) computation model is introduced. The problem description will be explained in Section 3. We present the approach used to solve the problem in Section 4. In Section 5, the analysis tool is explained. We show the implementation in Section 6. Three case studies are discussed in Section 7 with conclusion and future work in Section 8.

## 2 Mode-Controlled Data Flow

Many SDR jobs need to execute different sub graphs depending on the received data. Often, this will cause the data consumption/production rates of tasks to also become dependent on the values of the input data. For example, a Wireless Local Area Network (WLAN) receiver must attempt to synchronize to the incoming packet until it is successful. When this happens, the WLAN receiver must decode the packet, which implies the activation of a different task. As a result, these jobs cannot be represented by static data flow graphs like Single-Rate DF, Multi-Rate DF, or Cyclo-Static DF. A new dataflow model called Mode Controlled Data Flow (MCDF, see an example in **Figure 10**) is proposed to express these jobs.

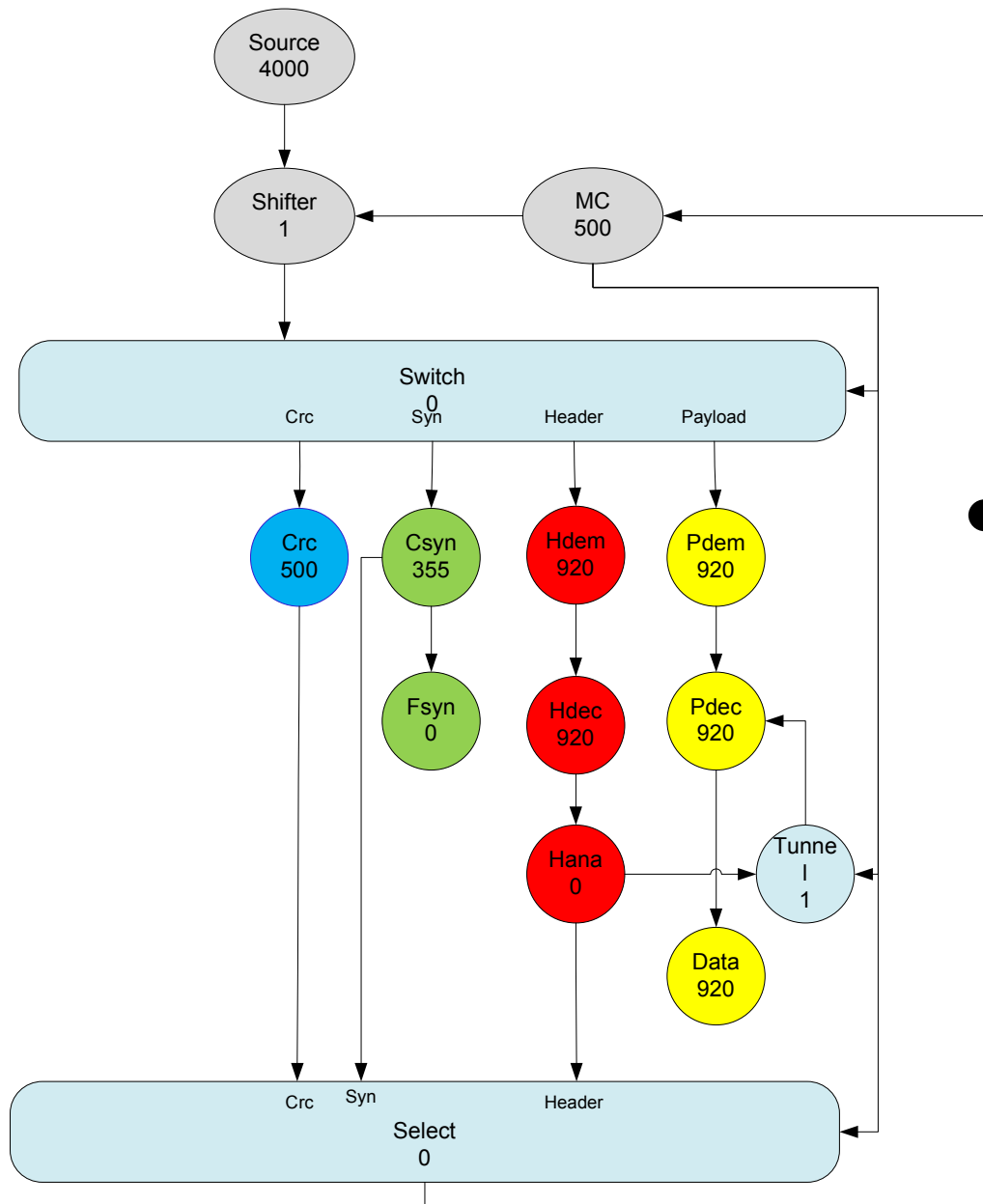


Figure 10 MCDF Graph of Wireless LAN

## 2.1 MCDF Overview

An MCDF graph  $G = (V, E)$  has a set of modes  $M$ :

$$M = \{m_1, m_2, \dots, m_n\}$$

MCDF actors are classified into modal actors and non-modal actors. An actor belongs to a mode  $m$  if it is connected to another actor that belongs to mode  $m$  or to a data-dependent port associated with mode  $m$ . If an actor belongs to a mode it is referred to as a **modal** actor.

For each actor  $a_i \in V$ , we have:

$$\text{mode}(a_i) = \begin{cases} m_j & m_j \in M \\ null & \end{cases} \quad (2-1)$$

Here we define  $\text{mode}(a)$  as the function which returns the mode attribution of actor  $a$ . For modal actors, the function returns the mode name of the actor. For non-modal actor, the function returns *null*. The MCDF graph may execute in certain mode  $m_j$ . When the MCDF graph executes in mode  $m_j \in M$ , only the non-modal actors and modal actors belong to mode  $m_j$  are enabled. Therefore, we define the concept of **sub-graph**  $G_j = (V_j, E_j)$  for **mode**  $m_j$  as:

$$V_j = \forall a_i \in V \text{ Such that } \text{mode}(a_i) = null \text{ or } m_j \quad (2-2)$$

$$E_j = \forall e \in E \text{ Such that } \text{source}(e) \in V_j \& \text{sink}(e) \in V_j \quad (2-3)$$

Here  $\text{source}(e)$  denotes the source actor of edge  $e$ ; and  $\text{sink}(e)$  denotes the sink actor of edge  $e$ . When the MCDF graph  $G$  executes in mode  $m_j$ , the sub-graph  $G_j$  should be built. For each sub-graph  $G_j$  of graph  $G$ ,  $MCM_j$  can be calculated which is the MCM calculated from the sub-graph  $G_j$ . If the current mode of an MCDF graph is  $m_j$ , we define the current **iteration** of the MCDF graph is an iteration of the sub-graph  $G_j = (V_j, E_j)$ .

MCDF contains a special actor, the Mode Controller (*MC*). *MC* is fired in every iteration. We define **port** to be the interface for each MCDF actor to send or receive data. *MC* has one special output port referred to as the **mode control port**. All the data-dependent actors have **control input ports**. At every iteration of *MC*, the value produced in the mode control port is used to drive the control input ports of all data-dependent actors in the MCDF. The main point of an MCDF graph is that after each firing of *MC*, a specific sub-graph will be executed according to the control signal produced by *MC*. The sub-graphs are **iteration exclusive**, i.e. only one of them will be executed for each iteration. A special actor called Tunnel is designed to model the communication between two sub-graphs.

The MCDF actors can also be classified into data-dependent actors and data-independent actors. Data-dependent actors are switch actors, select actors and tunnel actors. The rest are data-independent actors.

A simple example of MCDF is shown in Figure 11. There is one mode controller. Actors *A* and *B* are modal actors belonging to *mode1*; actors *C* and *D* are modal actors belonging to *mode2*. Actors “*source*” and “*sink*” are non-modal actors. The pair actor Mode Switch and Mode Select are data-dependent actors will be explained in the next subsection. Also, actor tunnel is a data-dependent actor.

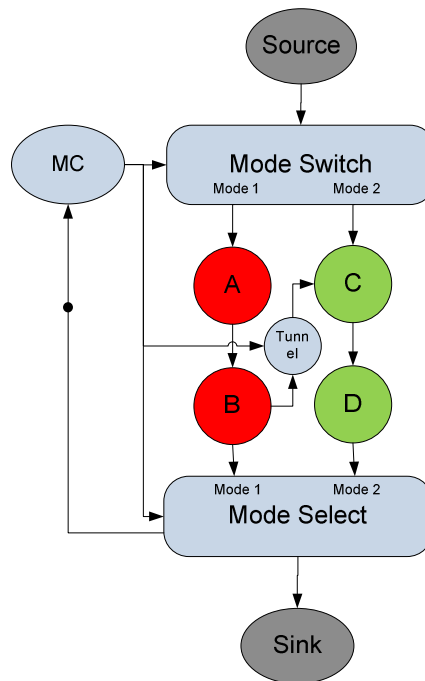


Figure 11 Mode Control Data Graph

## 2.2 Data-Dependent Actors

Data-dependent actors are classified into 3 types: mode switch, mode select and mode tunnel. Mode switch and mode select branch the graph into sub-graphs according to the control signal from *MC*. Tunnel is a special actor that allows the communication between sub-graphs without causing deadlocks while keeping determinist.

A **Mode Switch** actor has one control input port, one data input port, and  $M$  output ports. Each output port is associated with a mode. When a token is present on the control input port and on the data input port, the Mode Switch actor is fired. It consumes both input tokens and produces a token in the output port associated with the Mode indicated by the token consumed on the control input port. The output token has the same size and value as the token consumed on the data input port.

A **Mode Select** actor has a mode control input port,  $M$  data input ports and one output port. Each input port is associated with a mode. When a token is present on the control input port, its value is read and used to decide from which input port to consume a token. The actor is fired when a token is present in the control input port and in the data input port associated with the mode indicated by the token in the control input port. When fired it consumes both of these tokens. At the end of the firing, it produces on the output port a token with the same size and value as read from the modal input port. If the input port was not connected, the output token will have some pre-defined default value. Note that the firing rule must be evaluated in two steps.

A **Mode Tunnel** actor has one input control signal from the mode controller, one data input from Mode 1 ( $M1$ ) and one data output to Mode 2 ( $M2$ ). When the tunnel receives a control signal of  $M1$ , it fires, consumes one token from the input and stores this token in its internal memory. When the tunnel receives a control signal of  $M2$ , it copies the token stored in its internal memory to the output actor. If there are more than one executions of Mode 1, the internal token of tunnel will be replaced by the latest one. So the actor from Mode 2 only reads the latest token from tunnel.

## 2.3 Tunnel Representation

In a MCDF graph, actors represent the computation only. However, tunnels store tokens of previous iteration which can not be modeled as a single actor. Therefore an expanded model is designed to represent tunnels (See Figure 12).

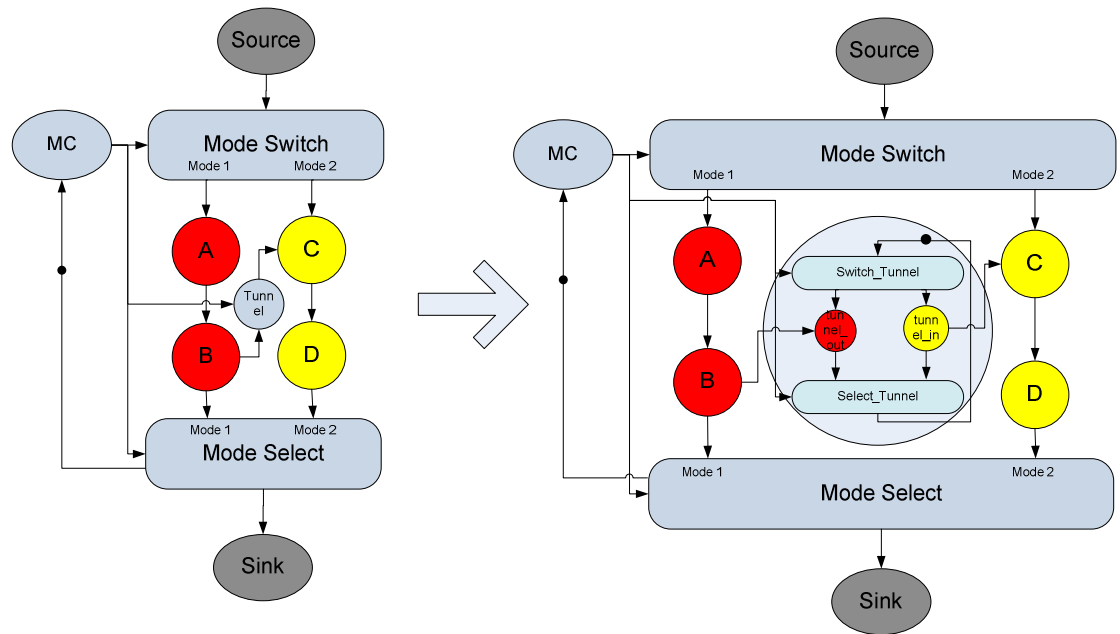


Figure 12 Tunnel Representation

As shown in Figure 12, this MCDF graph has actor set  $V = \{MC, Source, Mode\ Switch, Mode\ Select, Tunnel, Sink, A, B, C, D\}$ . Here the red colored actors are modal actors of *mode1*, and the yellow colored actors are modal actors of *mode2*.

The tunnel actor is replaced by four actors: one switch, one select, one tunnel out and one tunnel in. When MC sends *Mode1* control signal to switches and selects in the graph, all the actors in the sub-graph of *Mode1* fire in order. The token produced by *tunnel select* actor is stored in the FIFO edge. When MC sends *Mode2* control signal to switches and selects, the *tunnel switch* actor consumes the token from the FIFO edge and produces it to actor *tunnel in*. If the graph has been executing in *mode1* for more than one time before *mode2* is activated, only the data produced by the last iteration will be stored in the FIFO edge. This procedure well models the behavior of the tunnels with the existing types of actors and FIFO edges.

## 2.4 The Construction Rules of MCDF Graph

The construction rules of MCDF are designed to guarantee that each firing of the Mode Controller (MC) enables all the actors within a mode and the data dependent actors to fire until all edges return to the initial state, without any other activation of the MC being required. The construction rules are listed below:

- An MCDF graph has one mode controller, one or more switch actors, one or more select actors, zero or more tunnel actors, and arbitrary number of non-modal actors.
- An actor of an MCDF graph belongs to at most one mode
- The control ports of switch, select, and tunnel actors are connected to MC with delay-less edges.
- There is no delay-less cycle in the graph that does not traverse at least one Tunnel actor.



### 3 Problem Description

Based on the description above, the challenge of my thesis work is to adapt the existing scheduler to allow hard real-time quasi-static ordering schedule of MCDF graphs.

The existing scheduling strategy which uses the combination of TDM and static-order scheduling is no longer suitable for MCDF graphs. There are three effects that make the existing scheduling strategy fail to schedule MCDF graphs.

First, static-order scheduling adds dependent delay-less edges between the actors mapped on the same processor which is not always allowed by the construction rules of MCDF. Assume two actors *A* (belonging to mode1) and *B* (belonging to mode2) are mapped on the same processor. Under the static-order scheduling strategy, one delay-less edge will be added either from *A* to *B* or from *B* to *A*. According to the construction rules of MCDF graph, there should not be any delay-less edges between two actors belonging to two different modes.

Second, the current timing analysis method MCM computation provides the minimum guaranteed throughput. This method can not provide the tightest bounds on throughput or latency of the MCDF graph.

If we create the sub-graphs for each mode in an MCDF graph, each sub-graph may have quite different MCM values. Take the example of the timed MCDF graph shown in Figure 13 (a) as an example, the number in each actor represents the execution time. If all the actors have the information of execution time, this graph is a timed graph. Figure 13(a) is an MCDF graph. Figure 13(b) and Figure 13(c) are the sub-graphs of mode1 and mode2 of that MCDF graph.

The *MCM* of the graph is  $1+1+10+1=13$  which is also the *MCM* for the sub-graph of *mode2*. However, the *MCM* for the sub-graph of *mode1* is 4. Assume that the application executes in *mode1* for 10 iterations. If we simply use *MCM* of the whole graph, the throughput of these 10 iterations is  $1/13$ . However, if we know the *MCM* for the sub graph of *mode1*, the throughput of these 10 iterations will be improved to  $1/4$ . By computing the *MCM* for each mode, a much tighter timing analysis result can be found.

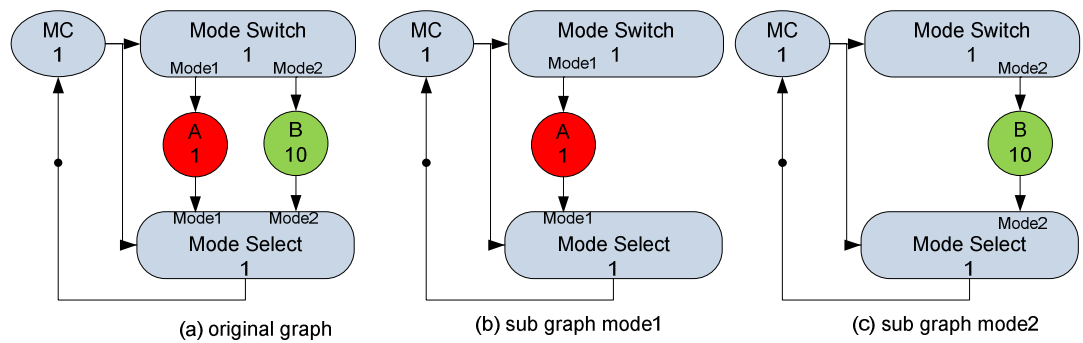


Figure 13 Timed MCDF Graph

Third, the timing analysis methods we have now can not provide latency analysis of the analysis model. There may be requirement like that the time the platform takes to execute in *mode1* once followed by executing in *mode2* once should be less than 100 microseconds. This latency information can not be calculated with the current timing analysis method.

Furthermore, not all the processors support preemptive scheduling. For instance, preemption is not allowed by some processors like EVP [1]. However, TDM scheduling is preemptive which means that high priority task can take the resource away from lower priority task. So the combination of TDM and static-order can not be applied on all the processors. New schedule strategy should be designed and schedule strategy for each processor should be extended to other strategies but not only TDM schedule.

### 3.1 Contribution of This Thesis

In this thesis, new scheduling strategies are introduced to schedule MCDF graph applications. Several timing analysis methods are implemented to analyze the temporal performance of MCDF graph running on the MPSoC. The contribution is listed below:

- A scheduling strategy of combination of quasi-static ordering and TDM/Round Robin scheduling is designed for MCDF graphs;
- A timing analysis method of Self-Timed Scheduling Timing analysis is introduced to analyze the scheduling mentioned in the first item;
- A second timing analysis method of Static-Periodic Scheduling Timing analysis is introduced to analyze the scheduling mentioned in the first item.

## 4 Overall Approach

In this section, we describe the approach used to schedule the MCDF graph on MPSoC and timing analysis methods used to analyze the schedule in the way of throughput or latency with certain mode sequence. They will be introduced in the following order: Timing Analysis Methods, Analysis Model Building and Scheduling Strategy.

### 4.1 Timing Analysis

As explained in Section 3, the throughput calculated from MCM for the whole MCDF graph is conservative comparing with MCM for each mode of the MCDF graph. In order to save more resources for other applications, it is necessary to investigate the MCM for each mode. Furthermore, we are interested in the time consumption of execution the graph in the order specified by a mode list. Therefore we want to define a new concept called Mode Sequence.

#### 4.1.1 Mode Sequence

A mode sequence is a valid sequence of control signals produced by the mode controller. The MCDF graphs will execute in the order of the mode sequence. We let  $(A^{num})$  represent a mode sequence where the graph executes mode  $A$  for  $num$  times. Therefore, the mode sequence  $A^n B^m C^p$  means the graph executes  $n$  times in mode  $A$  followed by  $m$  times in mode  $B$  followed by  $p$  times in mode  $C$ .

#### 4.1.2 Self-timed Schedule with Mode Sequence

As explained in Section 3, the MCM for different modes may be very different. In order to find tighter timing analysis results, we try to execute the MCDF graph in a Self-Timed Schedule (STS). To show the effect of STS, we try to do the timing analysis on the graph shown in **Figure 14**. **Figure 14** is an analysis model of an MCDF graph. First, we calculate the MCM of the analysis graph. Then, we try to execute the MCDF graph static periodically with a mode sequence of  $Mode1^1 Mode2^1$ . For each iteration, the period is assigned as the calculated MCM for the whole graph.

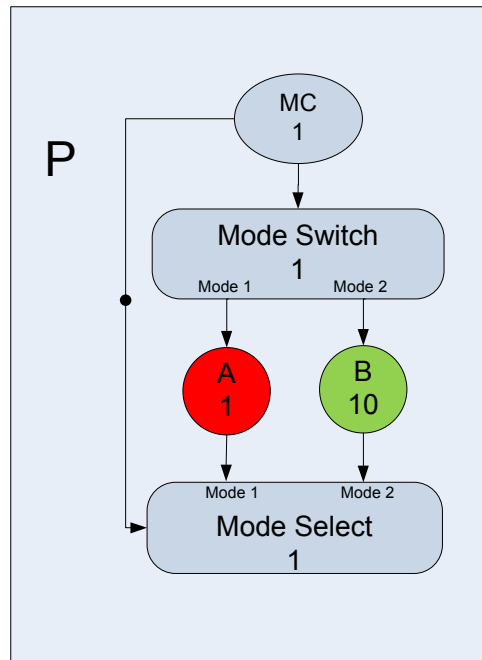


Figure 14 Timed Analysis Graph

The Gantt chart depicted in Figure 15 shows the periodic execution with mode sequence *Mode1' Mode2'* which assigns the MCM as the period for every mode. The horizontal axis is time. The block means that the processor is assigned to the actor shown on the block. The shadowed block means the processor is idle during this time interval. We can find out that the processor is idle for 9 time units because it takes less time to execute in *mode1* than in *mode2*.

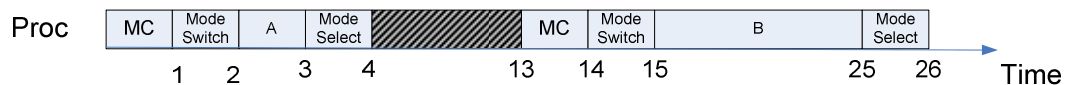


Figure 15 Periodic Schedule

The Gantt chart depicted in Figure 16 shows the self-timed execution with mode sequence of *Mode1' Mode2'*. There is no idle time in the processor which leads to the tightest timing result for this mode sequence. It takes 17 time units to finish executing in this mode sequence. It is impossible to find a schedule which takes less than 17 time units.

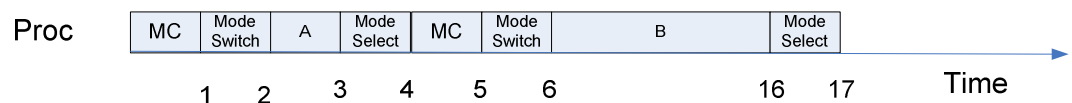


Figure 16 STS Schedule

Therefore, we can find the tightest timing analysis result for the MCDF graph with STS timing analysis method. However, performing self-timed timing analysis takes long time as it has to compute all the execution in the mode sequence.

### 4.1.3 Static-periodic Schedule with Mode Sequence

There is an alternative method, Static Periodic Schedule (SPS), to do the timing analysis. The SPS assigns a static period for every iteration. Since the MCM for each mode may vary a lot, we calculate the MCM for each mode and assign the period of each mode with its MCM. The graph is forced to execute in static periodically but each mode has a different period. By doing this, tighter timing analysis results can be obtained. Next, we show how to perform the static periodic timing analysis.

Assume there is an MCDF graph  $G = (V, E)$  with a mode sequence of  $A^n B^p$ , MCM for *mode A* and *mode B* are  $\mu_A$  and  $\mu_B$ . For *mode A* and *mode B*, we create sub-graph  $G_A = (V_A, E_A)$  and  $G_B = (V_B, E_B)$ . Under SPS, we force the execution into the periodic phase after the first iteration with the provided period.

It is stated in [12] that in order to not violate the firing rule, two conditions shown in (4.1) and (4.2) should be met.

$$\forall a_i \in V_A \quad s(a_i, 0) \geq 0 \quad (4-1)$$

$$\forall e(i, j) \in E_A \quad s(a_j, k) \geq s(a_i, k - d(i, j)) + e(a_i) \quad k \leq N \quad (4-2)$$

As the graph is executing under SPS, we have the following two properties in (4.3) and (4.4).

$$\forall e(i, j) \in E_A \quad s(a_i, k - d(i, j)) = s(a_i, k) - \mu_A \cdot d(i, j) \quad k \leq N \quad (4-3)$$

$$\forall a_i \in V_A \quad s(a_i, k) = s(a_i, 0) + \mu_A \cdot (k - 1) \quad k \leq N \quad (4-4)$$

From (4.2), (4.3), and (4.4), we can get the inequality shown in (4.5).

$$\forall e(i, j) \in E_A \quad s(a_j, 0) \geq s(a_i, 0) - \mu_A \cdot d(i, j) + e(a_i) \quad (4-5)$$

Therefore, (4.1) and (4.5) give the linear bounds of all the actors active in *mode A*. After executing for  $N$  iterations, the graph starts to execute in *mode B*. The SPS forces the execution to be periodic with the period of  $\mu_B$ . We denote the start times of the actors in executing in *mode B* as  $s'(a_i, 0)$ . According to the proof above, we have the following two linear bounds for actor start times in *mode B* shown in (4.6) and (4.7).

$$\forall a_i \in V_B \quad s'(a_i, 0) \geq 0 \quad (4-6)$$

$$\forall e(i, j) \in E_B \quad s'(a_j, 0) \geq s'(a_i, 0) - \mu_B \cdot d(i, j) + e(a_i) \quad (4-7)$$

The first iteration in *mode B* does not start from time 0. It depends on the last execution in *mode A*. As the graph executes periodically, the start time of the first iteration in *mode A* for each actor is  $(N - 1) \cdot \mu_A$  before the last iteration in *mode A* of that actor. Therefore, we can build the dependency of the first iteration in *mode B* on the first execution in *mode A*. The inequality is shown in (4.8). There will be such dependency only when there is an edge with delays larger than 1. This includes self-edges. Edges with more than 1 delay will be converted into several edges with 1 delay. This will be explained in the following subsection. So here we only talk about edges with 1 delay.

$$\forall e(i, j) \in E_B \ \& \ d(i, j) = 1 \quad s'(a_j, 0) \geq s(a_i, 0) + e(a_i) \quad (4-8)$$

Therefore, the start times of the actors in graph  $G$  during the transition phase from *mode A* to *mode B* are bounded by (4.1), (4.5), (4.6), (4.7) and (4.8). Furthermore, we get the execution time of mode sequence  $A^N B^P$  under SPS shown in equation (4.9).

$$f_{SPS}(A^N B^P) = f_{SPS}(A^1 B^1) + \mu_A \cdot (N - 1) + \mu_B \cdot (P - 1) \quad (4-9)$$

Here  $f_{SPS}(A^N B^P)$  means the time it takes to execute in the mode sequence of  $A^N B^P$  under SPS, and  $f_{SPS}(A^1 B^1)$  means the time it takes to execute in the mode sequence of  $A^1 B^1$  under SPS.  $f_{SPS}(A^1 B^1)$  is also called finishing time which can be calculated by adding the response time to the start time.

### 4.1.3.1 Expressing the Problem as a Linear Program

Many problems can be formulated as maximizing or minimizing an objective, given limited resources and competing constraints. If we can specify the objective as a linear function of certain variables, and if we can specify the constraints on resources as linear equalities or inequalities on those variables, then we have a Linear Programming (LP) problem. [11]

We have explained in section 4.1.3 that the execution time of an MCDF under SPS with a given mode sequence can be calculated from equation (4.9). In this subsection, we explain how to compute the start times of the actors in mode transition phase under SPS. As the start times of all the actors in the MCDF graph are bounded by inequalities of (4.1), (4.5), (4.6), (4.7) and (4.8), this problem can be represented as a linear program. The variables are the start times of the actors for every first execution of a specific mode in the mode sequence. The objective is to minimize the start time for each actor in the last mode of the mode sequence. Together with the inequalities mentioned above, a linear programming problem is built. By solving this linear programming problem, we get the solution for the start times of the actors.

To build the right time dependency in the current mode, an extra inequality should be added as the start time of all the actors depend on the start times of the previous executions. We can show this by an MCDF graph  $G = (V, E)$  with a mode sequence like  $A'B'C'$ . There are two edges  $e_1(i,j)$  and  $e_2(j,k)$ , where  $d(e_1) = 1$  and  $d(e_2) = 2$ . We define  $s_m(a_i, 0)$  to be the start time of actor  $a_i$  in the  $m^{th}$  execution of the mode sequence. So for these two delay edges, we have the following constraints:

$$s_3(a_j, 0) \geq s_2(a_i, 0) + e(a_i) \tag{4-10}$$

$$s_3(a_k, 0) \geq s_1(a_j, 0) + e(a_j) \tag{4-11}$$

The inequality means that the current start time of an actor depends on the start time of this actor  $d(i,j)$  number of iterations before. If the edges have delay like 1, 2, we need to know the start times of the actors 1 and 2 iterations before. If the graph has variance delay numbers, too much previous start time information will be needed to compute the current start time constraint. In order to avoid this problem, we try to make the current start time only depend on current execution or its last execution. This is done by distributing the delays bigger than one into sub delays (shown in **Figure 17**).

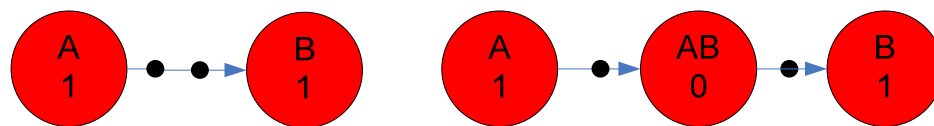


Figure 17 Delay Distribution

In the graph, there is a delay edge from actor A to actor B with 2 token sizes. To solve the linear programming problem, we have to store the start time information 2 iterations before. By adding an extra actor AB with execution time of 0, the start time of A is stored in actor AB. So we only need to get the start time information from the last iteration.

### 4.1.4 Self-Timed Scheduling versus Static Periodic Scheduling

Although STS can provide a tighter temporal performance, it also has disadvantages. We explain the disadvantage of STS with an MCDF graph executing in the mode sequence of  $A^N B^P$  such that  $N \geq 0, P \geq 0$ . If the values of  $N$  and  $P$  are rather small, it will be fast to perform the self-timed execution. However, when these two values become rather large, self-timed execution may take rather long time to do the timing analysis with the long mode sequence. In this situation, the static periodic execution can be performed fast because SPS is predictable. Figure 18 shows the Gantt chart of the static periodic execution in mode sequence of  $A^N B^P$ .

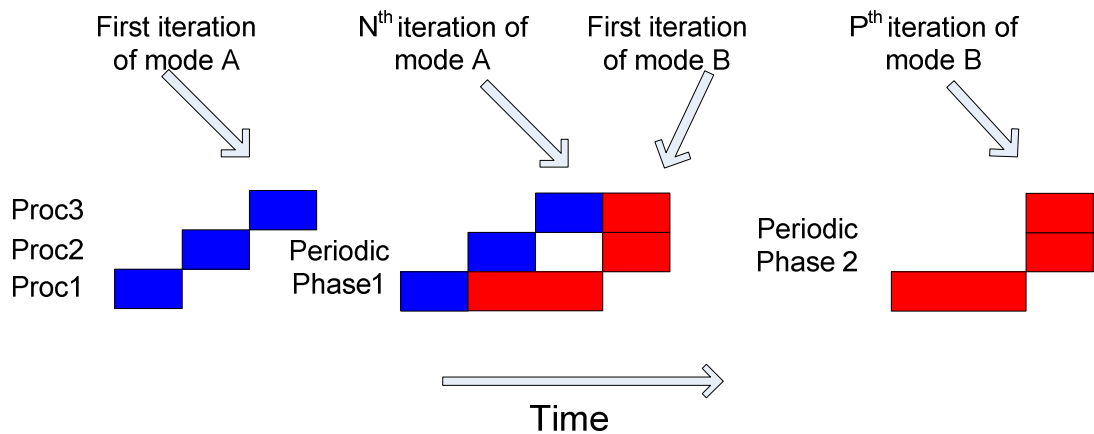


Figure 18 SPS Execution with Mode Sequence

We execute an MCDF graph with mode  $A$  and mode  $B$  on 3 processors. The blue blocks mean the processor is assigned to the actors from the sub-graph of Mode  $A$ . The red blocks mean that the processor is assigned to the actors from the sub-graph of Mode  $B$ . First, it executes in mode  $A$  which is depicted as blue blocks. After the first execution, it is forced to execute in a static periodic regime. The execution repeats for  $N-1$  times and starts to execute in Mode  $B$ . The first execution is depicted in left-most red blocks. The execution again is forced to execute in a static periodic regime. After repeating the red blocks for  $P-1$  times, the execution finishes. Therefore, we only compute the start times of all the actors for the transition phase from mode  $A$  to mode  $B$ . The complexity of SPS timing analysis does not grow with multiple consecutive executions of the same mode.

The time it takes to do SPS timing analysis depends on the complexity of the linear programming problem. We define the complexity of LP problem as a pair of  $(M, N)$ . Here  $M$  means the variables this LP has and  $N$  means the number of the inequalities of this LP. Assume we have an MCDF graph  $G = (V, E)$  with mode set  $\{A, B, C\}$ . We define  $X(G)$  as the number of actors the graph  $G$  has,  $Y(G)$  as the number of edges the graph  $G$  has, and  $D(G)$  as the number of delay edges the graph  $G$  has. We define the number of mode transition  $T$  to be the number of mode changes in the mode sequence. For mode sequence  $A^n B^p$ , the  $T$  is 1. To do SPS timing analysis for mode sequence, we build the LP problem. The complexity of this LP problem can be derived according to the inequalities (4.1), (4.5), (4.6), (4.7) and (4.8). Therefore, for the mode sequence of  $A^n B^p$ , we have the complexity  $(M_{LP}, N_{LP})$  in below:

$$M_{LP} = X(G_A) + X(G_B) \quad (4-12)$$

$$N_{LP} = Y(G_A) + Y(G_B) + D(G_B) \quad (4-13)$$

From the example shown above, we can find that the complexity of the LP problem grows with the growing of the number of mode transitions. Therefore, the time it takes to do SPS timing analysis depends on the number of mode transitions.

## 4.2 Building the Analysis Model

Before doing timing analysis, an analysis model must be built which includes mapping decisions that have already been taken. Based on the mapping and execution order of the actors, we generate response time models for all actors. In this subsection, we explain how to generate response time models for quasi-static ordered schedules running within run-time TDM or round robin schedulers.

### 4.2.1 Modeling Quasi-static Ordering Schedule

In a static ordering schedule, the actors mapped on the same processor execute in a static order. In a quasi-static ordering, each mode has a specific static ordering, but the decision on which mode to execute is taken independently at run-time for each iteration. In a quasi-static model, the modal actors start branching after mode switch. Each branch after mode switch contains only actors belonging to a specific mode. The modal actors belonging to the same mode are statically ordered.

For static ordering scheduling, there is a delay edge with one token from the last mapped actor of the group to the first mapped actor. While for quasi-static ordering, an edge with one token is added from the last mapped actor of each mode to the first mapped actor. Take the MCDF graph from Figure 11 as an example. Assume the platform to execute this graph has 2 processors  $P_1$  and  $P_2$ . The mapping list of the actor is  $[(MC, P_1), (Source, P_1), (Mode\ Switch, P_1), (A, P_1), (C, P_1), (Tunnel, P_1), (B, P_2), (D, P_2), (Select, P_2), (Sink, P_2)]$ . We then build the analysis model of quasi-static ordering schedule (See Figure 19).

In order to avoid more than one mode switch/select in the MCDF graph, we further expand tunnel switch and tunnel select shown in Figure 12 into 4 modal actors shown in Figure 19. Each tunnel switch is replaced by two modal actors in the two modes that the tunnel connects. Each tunnel select is replaced by two modal actors in the two modes that the tunnel connects. The two new modal actors generated from tunnel switch inherit all the input edges from the tunnel switch actor and inherit the output edges from the tunnel switch according to the mode. The two new modal actors generated from tunnel select inherit all the output edges from the tunnel switch actor and inherit the input edges from the tunnel switch according to the mode. Each modal actor generated from tunnel select has one delay edge to each modal actor generated from tunnel switch. In Figure 19, we can see that the tunnel switch is replaced by two modal actors "Swi\_T\_A" and "Swi\_T\_C" in mode 1 and mode 2. The tunnel select is replaced by two modal actors "Sel\_T\_A" and "Sel\_T\_C" in mode 1 and mode 2. They are connected to actors "T\_out" and "T\_in" which represent the actor *tunnel out* and actor *tunnel in*. There are four delay edges from each of the actor "Sel\_T\_A" and "Sel\_T\_C" to each of the actors "Swi\_T\_A" and "Swi\_T\_C".

To further simplify the analysis model, we delete the mode select actors. We define a mode merges to the select if there is an edge connects this select actor to an actor belongs to that mode. The non-modal actors after select actors will be scheduled in each mode which merges to the select actor. In Figure 19, both mode 1 and mode 2 merge to the select actor. There is a non-modal actor "sink" after the mode select. So this "sink" actor is scheduled in both mode 1 and mode 2.

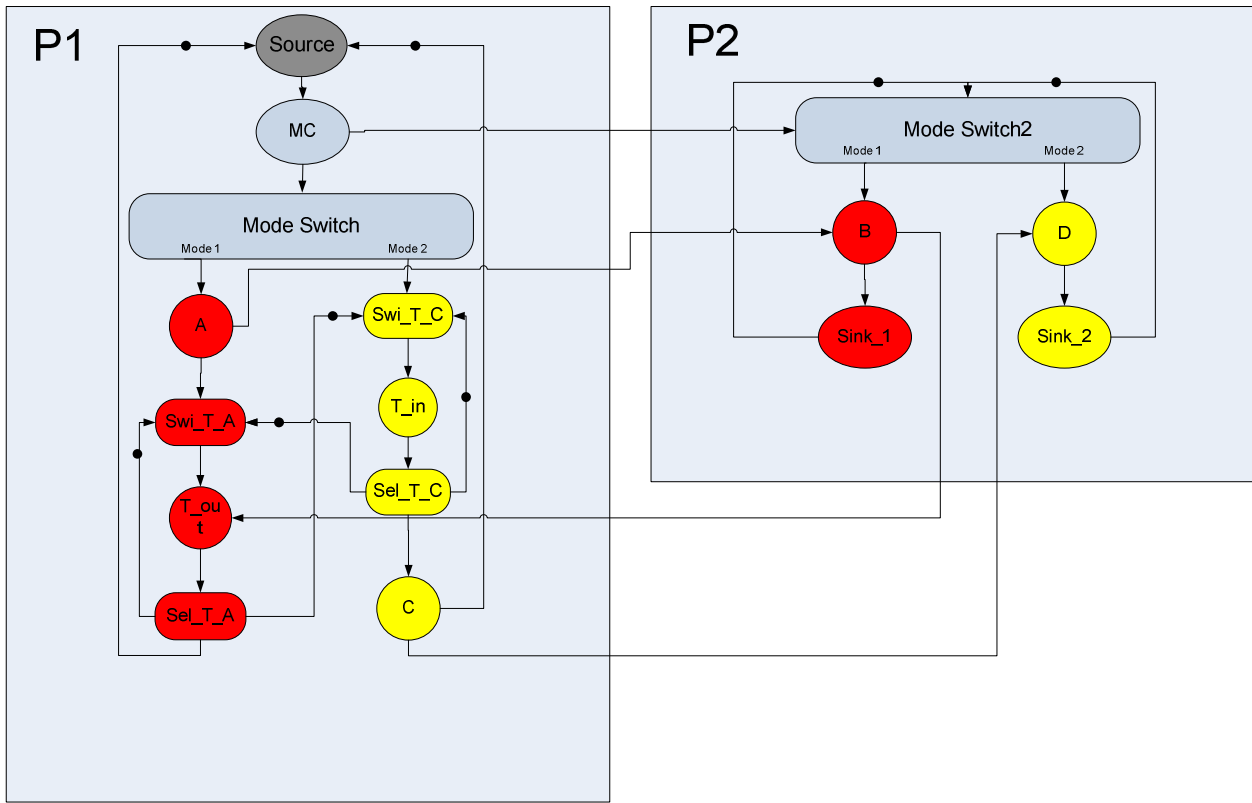


Figure 19 Quasi-static Ordering Analysis Model

From the analysis model, we can find that there are two branches corresponding to 2 modes after mode switch and inside each branch, modal actors are statically ordered.

For processor  $P_2$ , an extra mode switch actor is created which has an input edge from MC and output edges to the first actor of each mode mapped on  $P_2$ .

For processor  $P_2$ , actor *Sink1* is the last mapped actor in mode 1; actor *Sink2* is the last mapped actor of mode 2; while actor *Mode Switch2* is the first actor mapped on  $P_2$ . According to our model building method, an edge with one token delay is added from *Sink1* to *Mode Switch2*, also from *Sink2* to *Mode Switch2*.

## 4.2.2 Modeling Round Robin Schedule

We have explained in Section 3 that TDM scheduling strategy is not suitable for every processor. Therefore, we employ round robin as the local scheduler for some processors. In a similar way to TDM, the effects over the response times of task executing under a Round Robin (RR) scheduling can be modeled by generating a response time model. The response time of an actor  $a_i$  counts from the moment that  $a_i$  meets its enabling conditions until  $a_i$  finishes firing. Let  $e(a_i)$  is the execution time of actor  $a_i$ .

Assuming that the processor that actor  $a_i$  is mapped on has time period  $P$ , The time slice assigned to actor  $a_i$  is  $S$ , such that  $S \leq P$ . The **response time** of actor  $a_i$  is composed of two parts. One is the **arbitration time** which counts from the moment that  $a_i$  is activated until  $a_i$  gets the resource to start execution. The other one is the **processing time** when  $a_i$  starts execution until it finishes. The worst case of arbitration time happens if  $a_i$  is activated just after the slice time for  $a_i$  finishes. In this case,  $a_i$  has to wait for  $P-S$  time units. Round robin scheduling is non-preemptive, so the actor executes in a full slice time without being interrupted. This means that the processing time of  $a_i$  is equal to  $t(a_i)$ . As a result, the total worst-case response time of actor  $a_i$  is shown in equation 4-1.

$$r(a_i) = P - S + e(a_i) \quad (4-14)$$

### 4.2.3 Modeling Combination of Quasi-static Ordering and TDM/Round Robin Schedule

Figure 19 shows the quasi-static ordering analysis model of the graph shown in Figure 11. To model the combination of quasi-static ordering and TDM/round robin scheduling, the response time of all mapped actors should be adapted with the worst-case response time.

As described in the previous subsection, the worst-case response time of an actor is composed of two parts, arbitration time and processing time. Here actors mapped on the same processor are grouped by quasi-static ordering schedule. A group of actors are taken as a single big actor execution on the processor. Therefore, we calculate the worst case response time for this group of actors but not for each single actor. The arbitration time of the group starts when the first actor of this group is activated and finishes when this actor gets the resources to execute. As shown in Figure 20, one group of actors are mapped on  $P_1$ , while another group of actors are mapped on  $P_2$ . There are 4 edges between these two groups. Four Extra actors are added to the analysis model to represent the arbitration time of round robin scheduling. The response times for the actors are explained in the rest of this subsection.

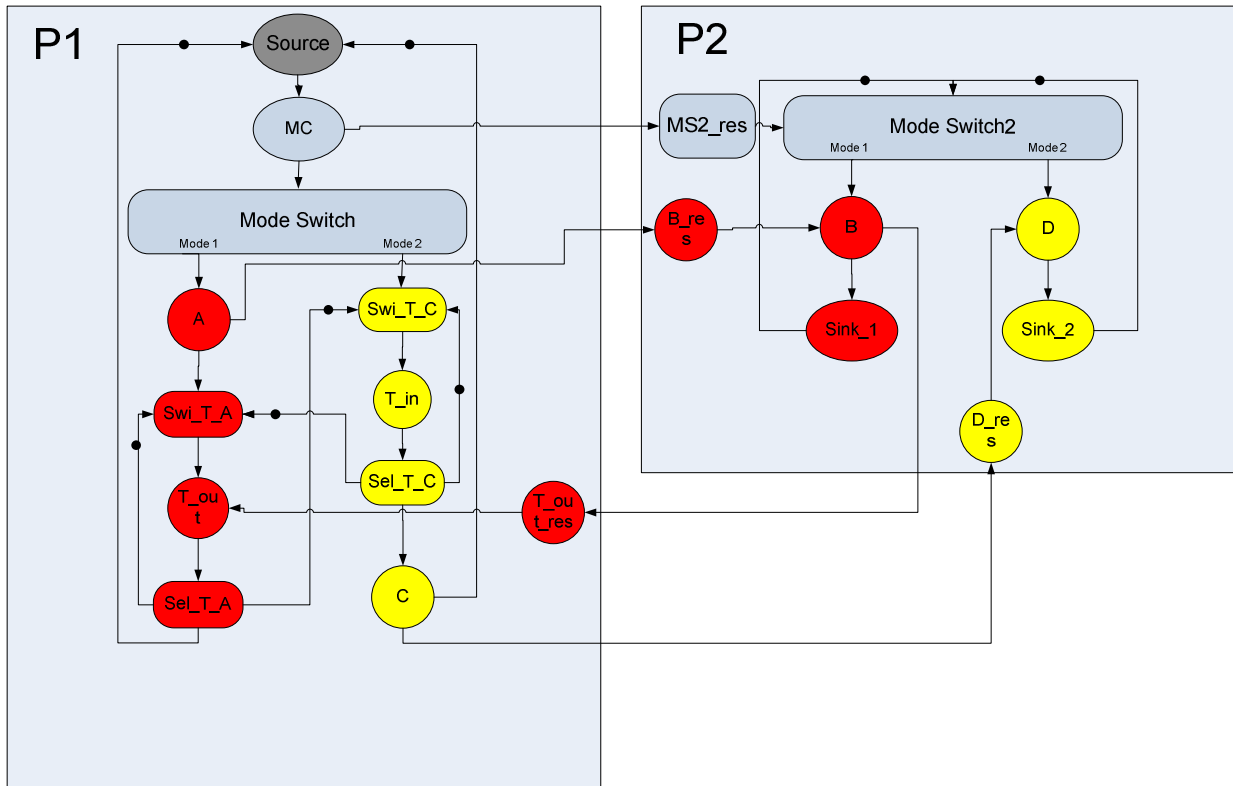


Figure 20 Modeling Combination of Quasi-static Ordering and Round Robin Schedule

Assume that the period of a processor is  $P$ . For quasi-static ordering, the round robin scheduler assigns a slice time  $S$  to a group of actors  $V = [a_1, a_2 \dots a_n]$  instead of only one actor. The group of actors should be assigned enough computation resources to ensure that they can finish execution in one slice time. The group of actors is divided into non-modal actors  $V_n$  and modal actors  $V_m$ . The mode set of this group of actors is  $M = \{m_1, m_2, \dots, m_k\}$ . So the resource requirement for mode  $m_k \in M$  is equal to:

$$Resource(m_k) = \sum_{a_i \in V_n} e(a_i) + \sum_{a_j \in V_m \& \text{mod}(a_j)=m_k} e(a_j) \quad (4-15)$$

The scheduler then assigns the maximum resource requirement of all modes to the group, which is:

$$Resource(group) = \sum_{a_i \in V_n} e(a_i) + \max_{m_k \in M} \sum_{a_j \in V_m \& \text{mod}(a_j)=m_k} e(a_j) \quad (4-16)$$

The worst case arbitration situation happens in the situation that the time slice assigned to this group just finishes and the first actor of this group is activated. This will cause the group to wait for the worst case arbitration time  $a(\text{group})$  shown in:

$$a(\text{group}) = P - \text{Resource}(\text{group}) \quad (4-17)$$

We model the arbitration time with extra actors. The arbitration time happens when there is an edge from other groups. And the response time for the actors in the group will be set as the execution time of that actor.

The analysis model for the scheduling strategy of combination of quasi-static ordering and TDM is the same as the combination of quasi-static ordering and round robin. The only difference is the value of time for each actor. As explained in 1.9.1, we have represented the arbitration time and processing time for actors under the combination of static ordering and TDM scheduling. For the combination of quasi-static ordering and TDM scheduling, the arbitration time and processing time are the same. They are:

$$p(a_i) = \lceil e(a_i) / S(a_i) \rceil \cdot P - (e(a_i) \bmod S(a_i)) \quad (4-18)$$

$$a(a_i) = P - S(a_i) \quad (4-19)$$

According to the scheduling strategy, the response time of the added extra actors will be set as the arbitration time of the group, and the response time of the actors in the group will be set as their processing times.

## 4.3 MCDF Scheduling Strategy

As explained in Section 3, static ordering scheduler cannot correctly schedule an MCDF graph. In an MCDF graph, you can not define a fixed execution order for the data-dependent actors. However, MCDF graph can have fixed execution order inside the mode branch. Therefore, we adapt the scheduler to have static order inside each mode branch which is called **quasi-static ordering**. When doing scheduling, we combine quasi-static ordering with TDM scheduling.

Further more, to solve the problem that not all processors support TDM scheduling, round robin scheduling is introduced as the local scheduler for processors. Thus, for processor who does not support TDM scheduling, we apply the combination of quasi-static ordering and round robin scheduling to it.

### 4.3.1 Adapting the Original Scheduler

For MCDF, the basic point of the scheduling strategy is to guarantee that only the actors belonging to one single mode are activated per each firing of mode controller. The scheduler is adapted in the following aspects:

- Per processor, before scheduling any modal actors, the scheduler first schedules a mode switch that reads a control signal from the mode controller and decides which mode branch to execute. The reason is explained in the following subsection.
- The mode select actors are deleted. Deleting mode select actors do not affect the temporal behavior of the graph. The actors after the mode select will be handled in the next rule.
- Any non-modal actors after mode select will be scheduled independently in every mode. That means these non-modal actors will be copied as many as the number of the modes, and each copy in different mode share the same properties (including execution time, and mapping) except the name.

### 4.3.2 Adding Mode Switch to Processor

In order to guarantee that only modal actors belonging to the current mode will be fired, we add mode switch to the processor on which more than two types of modal actors are mapped. We show the reason by comparing between two different analysis models built from the same MCDF graph.

Figure 21 depicts the analysis model without scheduling extra mode switch on each processor. Figure 22 depicts the analysis model with scheduling mode switch when there are more than two types of modal actors mapped on this processor. For both analysis models, actors *A*, *B*, and *C* are modal actors belonging to mode blue, and actors *D* and *E* are modal actors belonging to mode red. In Figure 22, an extra mode switch is scheduled on processor *P3* as there are two types of modal actors *C* and *E* mapped on this processor.

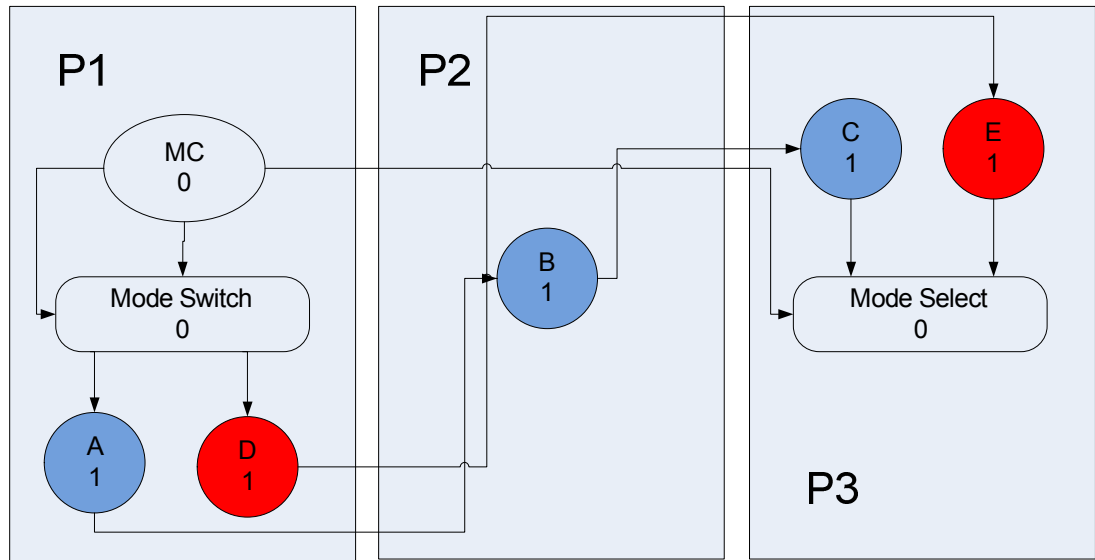


Figure 21 Analysis Model with Mode Switch

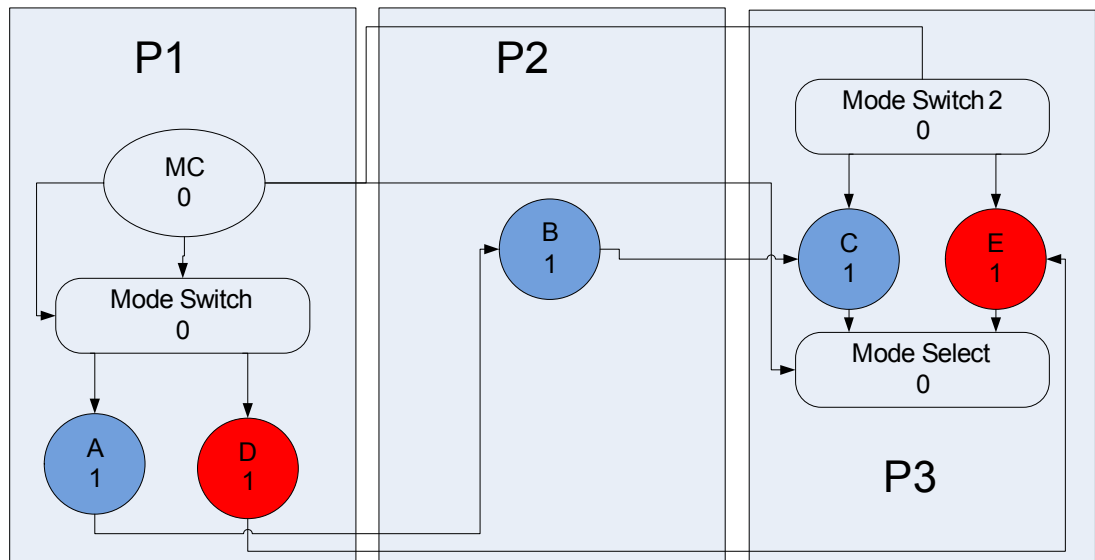
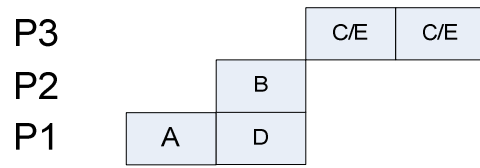
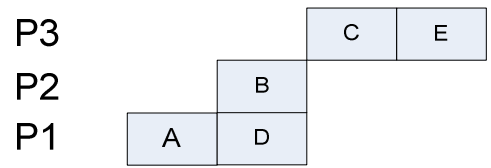


Figure 22 Analysis Model without Mode Switch

If we execute the graph in the mode sequence of mode blue once followed by mode red once, we obtain the Gantt chart for both of the schedule strategies. Figure 23 (a) shows the scheduler without adding an extra switch. With this schedule strategy, the worst case for both mode blue and mode red are 4 time units. Figure 23 (b) shows the scheduler with adding extra switches. With this schedule strategy, the worst case to finish executing in mode blue is 3 while for mode red is 4. From the example above, we can prove that it may be necessary to add extra mode switches on certain processors.



(a)



(b)

Figure 23 Gantt chart of the Self-time Execution of Switch Analysis Model



## 5 Analysis Tool Design

To model the real-time scheduling procedure, we use Objective Caml (OCaml) language to build the simulation Heracles tool. With Heracles, we build the analysis model according to the scheduling strategy. Timing analysis methods are implemented too.

### 5.1 Objective Caml

Caml is a general-purpose programming language, designed with program safety and reliability in mind. Caml supports functional, imperative, and object-oriented programming styles. It has been developed and distributed by INRIA, France's national research institute for computer science.

The Objective Caml system is the main implementation of the Caml language. It features a powerful module system and a full-fledged object-oriented layer. It comes with a native-code compiler that supports numerous architectures, for high performance; a bytecode compiler, for increased portability; and an interactive loop, for experimentation and rapid development. [10]

There are several properties that make Ocaml different from other programming languages. First, the functions can be taken as values in the code. So the name of the function can be passed as a parameter of other function. Second, the programmers do not have to give the type information of variables. The compiler will deduce the type information from the code. And also, the verification of the type will also be done during the compile period. Third, Ocaml can raise an exception when certain condition is met. With this, the programmer can get some intermediate result from the code. Also, Ocaml has imperative features. It has memory states that the programmer can modify like the mutable elements in the record structure. Supporting reference, loops, control structures are also the features of imperative language.

### 5.2 Heracles Analysis Tool

Heracles analysis tool is designed by the researchers from ST-Ericsson, Eindhoven. The tool takes graph file, platform architecture file and mode sequence file as inputs. It outputs the scheduling of the graph and possible temporal performance analysis. The scheduling includes the actor mapping and execution order. The temporal performance can be the throughput for the graph or for each mode, the self-timed execution of certain mode sequence, or the static periodic execution of certain mode sequence.

#### 5.2.1 Graph File

The implementation of the graph data type on Heracles was already done by previous developers. It is designed as below:

```
type (a', b') Graph = a' list * ((a' * a') * b') list
```

$a'$  is a template type which can be replaced by any type such as SDF actor type.  $b'$  is also a template type which can be replaced by any type such as edge type. The data structure of the graph is composed of two elements. The first element is a list of type  $a'$  such as a list of SDF actors. The second element is a list of edges. The edge is a tuple which represent the source and sink of the edge and an edge information type  $b'$ .

Figure 24 shows an example of graph file. It is the graph description of the MCDF graph shown in Figure 11. Each actor has a unique name which is used to identify the actor and a property "exec" which represents the execution time of this actor. Each actor may also has properties like "slice" means the slice time assigned by TDM scheduler, "group" which means the group number this actor belongs to, "proct" which means the processor type this actor should be mapped on, and "mode" which means the mode this actor belongs to.

Each edge should has the properties like "src" which means the source of this edge, "dst" which means the destination of this edge, "prod" which means the production rate of this edge, "cons" which means the consumption rate of this edge and "type" which indicates the type of the edge. Each edge may also have the property of "delay" which indicates the initial token on this edge.

```

1  actors
2
3  name="a" exec=1 slice=1 group=1 proct=1 mode=5;
4  name="b" exec=1 slice=1 group=2 proct=2 mode=5;
5  name="c" exec=1 slice=1 group=1 proct=1 mode=6;
6  name="d" exec=1 slice=1 group=2 proct=2 mode=6;
7  name="sink" exec=1 slice=1 group=2 proct=2;
8  name="mc" exec=0 slice=1 group=1 proct=1 type="mode_controller";
9  name="source" exec=1 slice=1 group=1 proct=1;
10 name="mswitch" exec=0 slice=1 group=1 proct=1 type="switch";
11 name="mselect" exec=0 slice=1 group=2 proct=2 type="join";
12
13 arcs
14
15 src="source" dst="mswitch" prod=1 cons=1 type="fifo";
16 src="mc" dst="mswitch" prod=1 cons=1 delay=0 type="control";
17 src="mc" dst="mselect" prod=1 cons=1 delay=0 type="control";
18 src="mswitch" dst="a" prod=1 cons=1 type="fifo";
19 src="mswitch" dst="c" prod=1 cons=1 type="fifo";
20 src="a" dst="b" prod=1 cons=1 type="fifo";
21 src="c" dst="d" prod=1 cons=1 type="fifo";
22 src="b" dst="mselect" prod=1 cons=1 type="fifo";
23 src="d" dst="mselect" prod=1 cons=1 type="fifo";
24 src="mselect" dst="sink" prod=1 cons=1 type="fifo";
25 src="mselect" dst="mc" prod=1 cons=1 delay=1 type="fifo";
26
27 end
28
29

```

Figure 24 Graph File

The parser reads the graph file and stores it in the data structure of Graph.

## 5.2.2 Architecture File

The graph is executed on an MPSoC platform. One example of the input architecture is shown in Figure 25. Each processor should have a unique *name* which identifies the processor, *type* which represents the type of the processor, *sched* which indicates the local scheduling strategy of the processor, *wheeltime* which represent the total resource of this processor, and *weight* which is used in reducing the slice time.

```

1 processor
2
3 name="EVP" wheeltime=1 type=1 sched="roundrobin" weight=100;
4 name="SwDecoder" wheeltime=1 type=2 sched="roundrobin" weight=100;
5 name="ARM" wheeltime=1 type=3 sched="tdma" weight=100;
6 name="Src" wheeltime=1 type=4 sched="tdma" weight=0;
7 name="Lat1" wheeltime=1 type=5 sched="off" weight=0;
8 name="Lat2" wheeltime=1 type=6 sched="off" weight=0;
9 name="Lat3" wheeltime=1 type=7 sched="off" weight=0;
10
11 end

```

Figure 25 Architecture File

### 5.2.3 Mode Sequence File

The real time requirement of executing in certain mode sequence is described in mode sequence file. Figure 26 shows an example of mode sequence file. In Heracles tool, we define a type:

*(name\*time)*

Here *name* indicates the mode name and *time* indicates how many times this mode should be executed. The data structure of a mode sequence is:

*(name\*time) list \* int*

It is composed of a list of *(name\*time)* and an integer which means the execution of the mode sequence should takes less time than the value of the integer.

```

1 mode_list
2
3
4 mode: "synch_mode" 1 mode: "header_mode" 1 mode: "payload_mode" 2 mode: "crc_mode" 1 time= 10000;
5
6 end

```

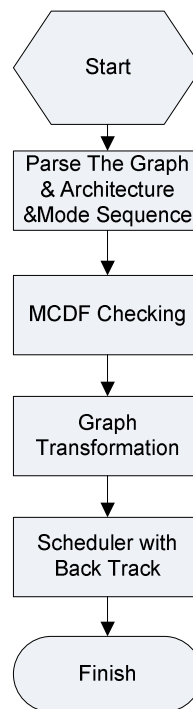
Figure 26 Mode Sequenc File

# 6 Implementation

We already described the data structure for graph, platform architecture and mode sequence in section 5. In this section, we describe the scheduling procedure for MCDF graph, analysis model building and timing analysis implementation.

## 6.1 MCDF Graph Scheduling Procedure

**Figure 27** shows the scheduling procedure and analysis model building procedure which is based on the original implementation designed by researcher from ST-Ericsson. We modified the implementation to schedule MCDF graph.



*Figure 27 Heracles Scheduling Procedure for MCDF Graph*

After parsing the MCDF graph into the system, the tool first checks the correctness of the input MCDF graph according to the construction rule explained in section 2.

After we get the correct input MCDF graph, we do the graph transformation. The graph transformation includes tunnel representation and graph adaptation to the scheduler. The tunnel representation is done according to description in section 2. The adaptation is implemented according to *section 4.3.1* which includes deleting switch actors as extra switch actors will be scheduled by the scheduler, deleting select actors, copying non-modal actors after select actors.

Currently, we only consider single rate MCDF, so MCDF is already HSDF without SDF-to-HSDF conversion. The system then initialize the candidate list and enters the backtracking procedure which is described in section 1. The modification to the scheduling with back track procedure includes changing the schedule strategy and analysis model building. After each mapping of the actor, the group of actors is quasi-statically ordered. For each processor, extra mode switch actor is added if necessary. The response time of the arbitration actors are set according to equation 4-17 and 4-19. Timing analysis methods include STS and SPS will be used to check the validation of the mapping.

## 6.2 Self-timed Scheduling with Mode Sequence

STS is used to calculate MCM for the graph in the existing analysis tool. To calculate the MCM, the system starts to execute the graph until it enters the periodic regime. The MCM then can be calculated from the periodic phase. If we want to know the throughput for executing certain mode, we create sub graph according to the mode and do symbolic simulation on this sub graph which is implemented by previous developers from ST-Ericsson and explained in the next subsection. Here we want to execute the MCDF graph with certain mode sequence under STS. We want to stop the execution when the systems finished executing the mode sequence no matter it enters the periodic phase or not. The following sub sections will explain the symbolic simulation to calculate MCM under STS and the modification we do to make symbolic simulation calculate the latency for mode sequence.

### 6.2.1 Symbolic Simulation Procedure

The state space is used to simulate the status of the graph during execution. By doing this, the transient phase and periodic phase of STS execution can be found. Further, the MCM can be calculated from the timing analysis of the periodic phase.

The state of a dataflow graph is a pair  $(\gamma, \nu)$ , composed by the state of each actor and the state of its FIFO edges.

As we may have several actors running concurrently, an actor state  $\nu$  is the sorted multi-set (one element per executing actor) of the remaining execution time for each of the actor firings.

The FIFO state  $\gamma$  is defined by how many tokens are stored within the edge.

Let us define a dataflow graph transition as changing from a state into another different state. We may denote a transition as  $(\gamma_1, \nu_1) \xrightarrow{\beta} (\gamma_2, \nu_2)$ , where  $\beta \in \{start, finish, update\}$  denotes the type of transitions. A start transition may only occur, accordingly to the dataflow graph firing rules, when enough tokens are present on all of the actor's input FIFOs.

A finish transaction is scheduled as soon as a start transition is, as we have knowledge of the actor execution/response time. An update transition is issued when a pre-determined actor executes so that we may update and compare the states.

A graph execution is therefore the set of transitions from  $t = 0$  until  $t = 1$ , where  $t$  stands for the simulation clock time. In [8] is demonstrated that a consistent graph execution flow is composed of two parts. The transient phase consisting of a finite number of transitions, and in the periodic phase, a sequence of transitions will repeat infinitely. Suppose we have a timed sub graph shown in Figure 28.

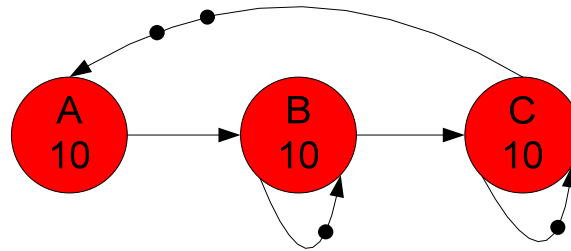


Figure 28 Simple Sub Graph

The state space exploration of Figure 28 is shown in Figure 29. The numbers in the circle means the time of the state. The shadowed rectangular near the circle contains the events happening at this state. We can find out that the state of 60 has the exactly the same state of 30 which means that the periodic phase of the STS execution is found.

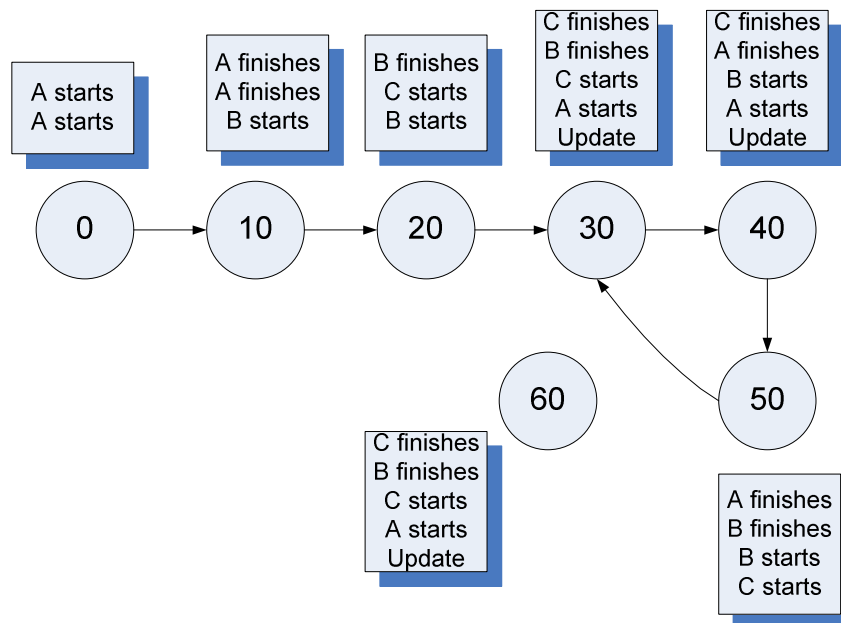


Figure 29 State Space Exploration

## 6.2.2 Alternative to Building Sub Graph

In order to calculate the MCM for certain mode or execute the graph in certain mode, we do not create sub-graph for that mode during implementation. Instead, we set the response time of the modal actors which do not belong to the current mode to be 0.

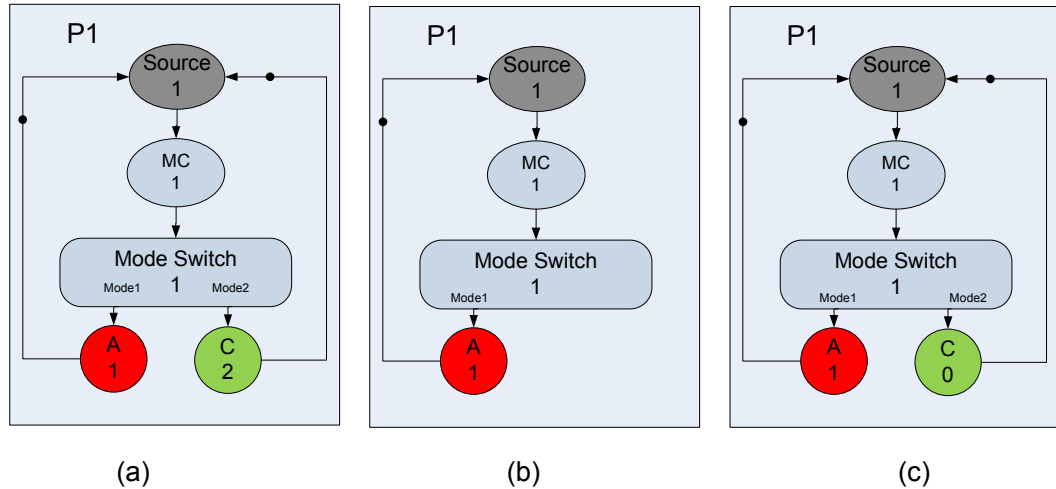


Figure 30 Sub Graph Building

Figure 30 shows an example about how we build the sub-graph and the alternative to the sub-graph. Figure 30 (a) is the original MCDF graph. To build the sub-graph of mode1, actor C is deleted and the edges associated with actor C are deleted too. The sub-graph for mode1 is shown in Figure 30(b). Instead of building sub-graphs, we design an alternative model to represent sub-graphs by setting the response times of the modal actors not belonging to the current mode to be 0. The alternative for sub-graph shown in Figure 30(b) is depicted in Figure 30 (c). During STS execution the alternative graph in Mode1, the actor C will be executed too. They will not affect the timing analysis result because their time valuations are zero. This means that by applying symbolic simulation to Figure 30 (b) and Figure 30 (c), we get the same timing analysis result. We explain this by doing state space exploration to the sub-graph shown in Figure 30(b) and to the graph shown in Figure 30(c). The state spaces are depicted in Figure 31(a) and Figure 31(b). We can get exactly the same timing analysis results from these two state space explorations.

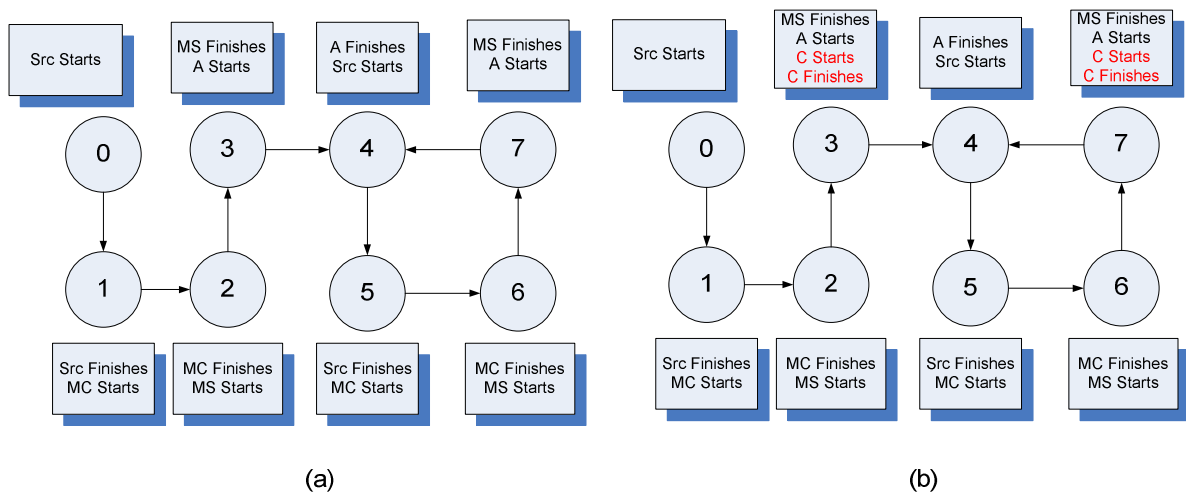


Figure 31 Symbolic Simulation State Space of Sub Graph

By applying the symbolic simulation to the alternative graph of sub-graph of mode1, we can compute the MCM for mode1. Although the alternative graph of sub-graph does not violate the time consumption during execution, it increases the buffer size of the output edges of the modal actors. The reason is the modal actors which do not belong to the current mode execute and produce tokens to their output edges. Therefore, the buffer size requirement is increased. When there is requirement about buffer size, the alternative graph for sub-graph can not be used.

### 6.2.3 Symbolic Simulation Modification with Mode Sequence

In the current version of Heracles, symbolic simulation is used to calculate the MCM for the input graph. However, we want to calculate the latency of a mode sequence. The symbolic simulation function should be modified in the way of changing the stop condition.

Assume we have the graph shown in Figure 30(a) as the input graph and a mode sequence of  $Mode1^1 Mode2^2$ . Symbolic simulation first calculates the length of the mode sequence, and unfolds the mode sequence into a list of {Mode1, Mode2, Mode2}. Each actor is assigned one property called number of firings.

During symbolic simulation, the system checks the number of firings of all the actors. No matter what the current mode it is, all the actors will be executed as the time valuations of those modal actors not belonging to the current are set to be 0. If all the actors have the number of firings bigger or equal than the length of mode sequence, the symbolic simulation stops and outputs the time of that moment. The time here is exactly how long it takes to execute in the entire mode sequence.

As for the MCDF graph, each actor executes once for every iteration. The number of firings of an actor is increased by 1 after each firing. Before every execution of modal actors, the system checks the number of firings of that actor. With this number, the system finds the next mode from the mode sequence. For example, if actor A has the number of firing an  $FN_A$ , then the systems searches the unfolded mode sequence in the position of  $(FN_A + 1)$ . The system matches the search mode with the mode of actor A. If it does not match, the response time of A is set to be 0. Otherwise, A executes with its original response time. We show the symbolic simulation procedure of the MCDF graph shown in Figure 30(a) with the mode sequence of Mode1' Mode2' below in Figure 32.

Time	Actor	Event	Fired Times
0	source	starts	0
1	source	finishes	1
1	MC	starts	0
2	MC	finishes	1
	Mode		
2	Switch	starts	0
	Mode		
3	Switch	finishes	1
3	A	starts	0
3	C	starts	0
3	C	finishes	1
4	A	finishes	1
4	source	starts	0
5	source	finishes	2
5	MC	starts	1
6	MC	finishes	2
	Mode		
6	Switch	starts	1
	Mode		
7	Switch	finishes	2
7	A	starts	1
7	C	starts	1
7	A	finishes	2
8	C	finishes	2

*Figure 32 Symbolic Simulation Mode Sequence*

From Figure 32, we can find out that modal actors A and C execute in each iteration. However, the response time of actor C is zero when executing in mode1 and the response time actor A is zero when executing in mode2. It takes 8 time units to execute the mode sequence of Mode1' Mode2'.

## 6.3 Static-periodic Schedule with Mode Sequence

To do the timing analysis of an MCDF graph in a mode sequence with SPS, we need to compute the transition time from one mode to another. As explained in section 4, the transition problem is translated into linear programming constraints. This section, we explain the methods to solve linear programming problem and the method we use in implementation.

### 6.3.1 The Procedure of Solving Linear Programming Problem

One of the classical methods to solve linear programming problem is simplex algorithm. GNU Linear Programming Kit (GLPK) solves the LP problem with primal and dual simplex methods. The GLPK package is intended for solving large-scale Linear Programming (LP), Mixed Integer Programming (MIP), and other related problems. [13]

As the Heracles tool is implemented in OCaml, we use OCaml-GLPK which is OCaml bindings to GLPK. We show how OCaml-GLPK solves LP problem with the MCDF graph  $G = (V, E)$  shown in Figure 30(a) with modes mode1 (M1),  $\mu_1 = 4$  and mode2 (M2),  $\mu_2 = 5$ . In order to do the timing analysis of the graph execute in mode sequence of  $M1^4M2^5$  ( $M1$  stands for mode1, and  $M2$  stands for mode2) under SPS, we build the linear programming problem to find the bounds for start times of the actors in the transition phase of  $M1^4M2^5$ . For each actor active in mode1, we have inequalities below according to inequalities (4.1):

---


$$\begin{aligned} s(src,0) &\geq 0 \\ s(MC,0) &\geq 0 \\ s(MS,0) &\geq 0 \\ s(A,0) &\geq 0 \\ s(C,0) &\geq 0 \end{aligned}$$


---

And inequalities (4.5):

---


$$\begin{aligned} s(MC,0) &\geq s(src,0) + t(src) = s(src,0) + 1 \\ s(MS,0) &\geq s(MC,0) + t(MC) = s(MC,0) + 1 \\ s(A,0) &\geq s(MS,0) + t(MS) = s(MS,0) + 1 \\ s(C,0) &\geq s(MS,0) + t(MS) = s(MS,0) + 1 \\ s(src,0) &\geq s(A,0) + t(A) - \mu_1 \cdot 1 = s(A,0) + 1 - 4 \end{aligned}$$

$$s(src,0) \geq s(C,0) + t(C) - \mu_1 \cdot 1 = s(C,0) + 0 - 4$$


---

And inequalities (4.6):

---

$$s'(src,0) \geq 0$$

$$s'(MC,0) \geq 0$$

$$s'(MS,0) \geq 0$$

$$s'(A,0) \geq 0$$

$$s'(C,0) \geq 0$$


---

And inequalities (4.7):

---

$$s'(MC,0) \geq s'(src,0) + t(src) = s'(src,0) + 1$$

$$s'(MS,0) \geq s'(MC,0) + t(MC) = s'(MC,0) + 1$$

$$s'(A,0) \geq s'(MS,0) + t(MS) = s'(MS,0) + 1$$

$$s'(C,0) \geq s'(MS,0) + t(MS) = s'(MS,0) + 1$$

$$s'(src,0) \geq s'(A,0) + t'(A) - \mu_2 \cdot 1 = s(A,0) + 0 - 5$$

$$s'(src,0) \geq s'(C,0) + t'(C) - \mu_2 \cdot 1 = s(C,0) + 2 - 5$$


---

And inequalities (4.8):

---

$$s'(src,0) \geq s(src,0)$$

$$s'(MC,0) \geq s(MC,0)$$

$$s'(MS,0) \geq s(MS,0)$$

$$s'(A,0) \geq s(A,0)$$

$$s'(C,0) \geq s(C,0)$$

$$\begin{aligned}s'(src,0) &\geq s(A,0) + t(A) = s(A,0) + 1 \\s'(src,0) &\geq s(C,0) + t(C) = s(C,0) + 0\end{aligned}$$

---

We set the objective function to minimize:

$$s'(src,0) + s'(MC,0) + s'(MS,0) + s'(A,0) + s'(C,0)$$

The start times of the actors in each mode will be the variables for the linear programming problem. With the inequalities and the objective function, we create a complete linear programming problem. That can be solved by OCaml-GLPK.



# 7 Experiment and Results

In this section, 3 cases will be studied to show how to apply our scheduling strategy and timing analysis to the MCDF graphs. We want to prove that the combination of quasi-static ordering and TDM/round robin scheduling can schedule MCDF graphs. Furthermore, with the 3 case studies, we want to find the advantage and disadvantage of timing analysis methods, STS and SPS.

In the first 2 cases, the MCDF graphs will be running on the MPSoC platform template depicted in Figure 1. It includes the type general core like ARM, to handle control and generic functionality, the type vector processor core, like EVP, to handle detection, synchronization, and demodulation, and the type of application-specific Software Codec processor that takes care of the baseband coding and decoding functions, and input/output (I/O) ports (see Figure 1). The platform is used to handle several radio standards (Wireless Lan, TD-SCDMA, DVB-H, and UMTS).

## 7.1 Case 1 DVB-T MCDF Graph

We apply the combination of quasi-static ordering and TDM/round robin to the MCDF graph of Digital Video Broadcasting-Terrestrial (DVB-T) baseband receiver shown running on the MPSoC platform template depicted in Figure 1. The MCDF graph of DVB-T baseband receiver (see Figure 33) has 3 modes: “*synch\_mode (syn)*”, “*drop\_mode (drp)*”, and “*dem\_mode (dem)*”. The actors colored in blue belong to “*dem\_mode*”, the actors colored in green belong to “*drop\_mode*”, and the actors colored in red belong to “*synch\_mode*”. There are 2 data-dependent actors switch and select, one mode controller MC, and one non-modal actor “*source*”. The numbers on the actors are the execution times with the time unit of Nano-Second (ns). The mappings of the actors are given in the graph file (See 0). Actor “*source*” is mapped on the ARM core. Actor “*dec\_sink*” and “*data\_out*” are mapped on the core of software decoder. The rest are mapped on EVP. The summarized DVB-T MCDF graph information is shown in Table 2. The input graph file of DVB-T can be found in Appendix B.

Table 2 DVB-T MCDF Graph Information

Actor_name	Short_name	Time	Mode	Map_on_Processor
Mode				
Controller	MC	427 ns	Null	EVP
Source	Src	0 ns	Null	ARM
Switch	Switch	683 ns	Null	EVP
Select	Select	250 ns	Null	EVP
Synch_acq	Syn	57200 ns	Syn	EVP
Drop	Drop	843 ns	Drop	EVP
Demode	Dem	57400 ns	Dem	EVP
Decode&Sink	Dec	335500 ns	Dem	Software Decoder
Data_out	Data	0	Dem	Software Decoder

The real-time constraint for DVB-T is that the throughput of mode “dem\_mode (dem)” should be larger than  $1/(896\mu s)$ . Here we can represent this constraint as the MCM for mode “dem\_mode” should be smaller than  $896\mu s$  which is equal to  $896000ns$ .

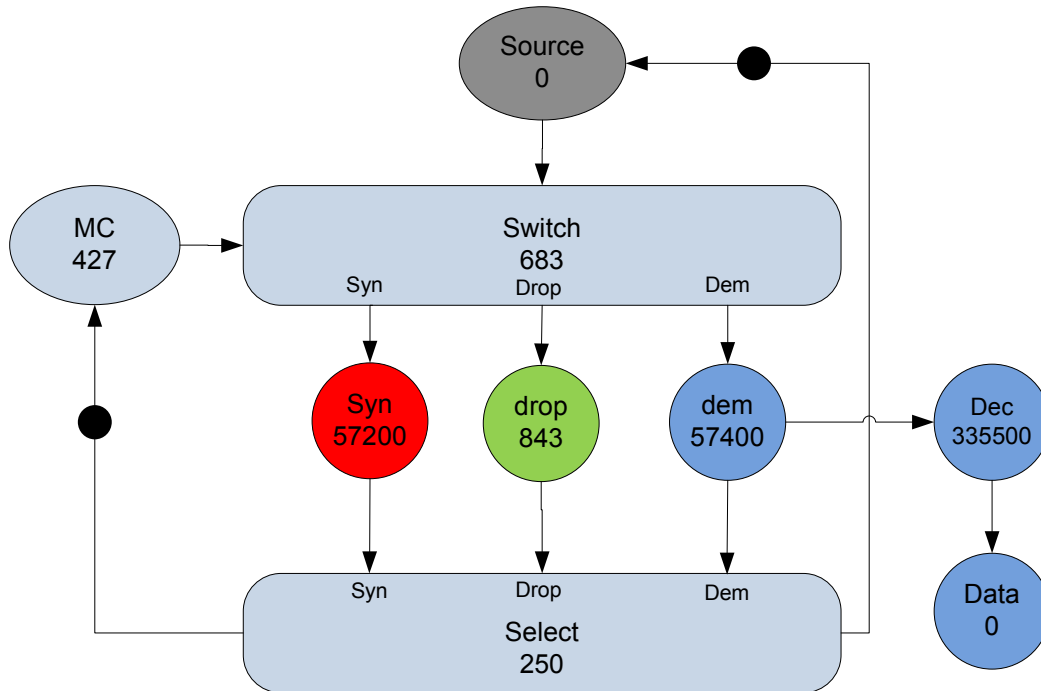


Figure 33 DVB-T Baseband Receiver

As there is only real-time constraint about throughput, we do the scheduling and compute the MCM as the timing analysis method. It is done with command:

```
./time Heracles -c2h -how -scc -cyc -f graphfile -s systemfile
```

Here *graphfile* is the graph file of the DVB-T, and the *systemfile* is the MPSoC platform file.

The DVB-T MCDF graph is scheduled by our scheduling strategy with the real-time requirement mentioned above. The Heracles tool outputs the schedule of the graph in Figure 34. The mapping of the actors is shown in the “bindings” information, and the execution order is shown in the “DAG” information. We can generation the analysis model with “bindings” and “DAG”. By running this MCDF graph on the MPSoC platform mentioned above, we get the result of the MCM for mode “dem\_mode” which is  $335500ns$ . This MCM is less than the required  $896000ns$  which means that it meets the real-time requirement. This example shows that our scheduling strategy can schedule an MCDF graph and perform timing analysis methods on the timing analysis model.

```
The graph satisfies the mudPrinting DAG
Actor Name: dec_sink_1_response Actor id: 33 Plist: 13
Actor Name: switch0 Actor id: 32 Plist: 17 16
Actor Name: synch_acq_1 Actor id: 11 Plist: 32 16 17
Actor Name: drop_1 Actor id: 12 Plist: 32 16 17
Actor Name: dem_1 Actor id: 13 Plist: 32 16 17
Actor Name: dec_sink_1 Actor id: 14 Plist: 33 13
Actor Name: data_out_1 Actor id: 15 Plist: 14
Actor Name: mc_1 Actor id: 16 Plist:
Actor Name: source_1 Actor id: 17 Plist: 16
Finished Printing DAG
```

```
-Printing Schedule-
```

```
-----
Printing DAG
Finished Printing DAG
```

```
Printing Bindings
data_out_1 proc_id 1
dec_sink_1 proc_id 1
dem_1 proc_id 0
drop_1 proc_id 0
synch_acq_1 proc_id 0
source_1 proc_id 0
mc_1 proc_id 0
```

```
Finished Printing Bindings
```

```
Printing Comap Bindings
Finished Printing Comap Bindings
```

```
Printing Group Bindings
Group 2 proc_id 1
Group 1 proc_id 0
```

```
Finished Printing Group Bindings
Current Th: 335500 Max Th: 896000
Finished Printing Schedule
```

```
1 solutions found
```

*Figure 34 DVB-T Schedule Result*

According to the inequality 4.16, we can reduce the slice time assigned to each group of actor. For the group mapped on software decoder processor, the time slice can be reduced to 335500 ns. For the group mapped on EVP processor, the time slice can be reduced to  $427+683+250+57400 = 58760$  ns. For the group mapped on ARM processor, the time slice can be reduced to 0 ns.

## 7.2 Case 2 Wireless Lan MCDF Graph

We apply the combination of quasi-static ordering and TDM/round robin to the MCDF graph of Wireless LAN baseband receiver shown in Figure 35 running on the MPSoC platform depicted in Figure 1. The graph file of Wireless LAN baseband receiver can be found in Figure 35. The number on each actor is the execution time of that actor. Wireless LAN MCDF graph has modal actors belonging to 4 modes: “*crc\_mode (crc)*” in blue, “*synch\_mode (syn)*” in green, “*header\_mode (hd)*” in red, and “*payload\_mode (pay)*” in yellow, 3 data-dependent actors *Mode Switch*, *Mode Select*, and *Tunnel*, one *Mode Controller*, and 3 non-modal actors *source*, *shifter*, and *blackhole*.

The mappings of the actors are given in the graph file (see Appendix C). Actor *Source* is mapped on ARM processor. Actor *header\_decode* and *payload\_decode* are mapped on Software Codec processor. Actor *data\_out* is mapped on external I/O ports. The rest are mapped on the EVP processor. The summarized Wireless LAN MCDF graph information is shown in Table 3.

Table 3 WLAN MCDF Graph Information

Actor_name	Short_name	Time	Mode	Map_on_Processor
Mode Controller	MC	500 $\mu$ s	Null	EVP
Source	Src	4000 $\mu$ s	Null	ARM
Shifter	Shifter	1 $\mu$ s	Null	EVP
Mode Switch	Switch	0 $\mu$ s	Null	EVP
Mode Select	Select	0 $\mu$ s	Null	EVP
Tunnel	Tunnel	1 $\mu$ s	Null	EVP
Crc	Crc	500 $\mu$ s	Crc	EVP
Cfensynch	Csyn	355 $\mu$ s	Syn	EVP
Ffensynce	Fsyn	0 $\mu$ s	Syn	EVP
Header_demode	Hdem	920 $\mu$ s	Header	EVP
Header_decode	Hdec	920 $\mu$ s	Header	Software Decoder
Header_analysis	Hana	0 $\mu$ s	Header	EVP
Payload_demode	Pdem	920 $\mu$ s	Payload	EVP
Payload_decode	Pdec	920 $\mu$ s	Payload	Software Decoder
Data_out	Data	920 $\mu$ s	Payload	Extern

A complete Wireless Lan job runs twice in “*synch\_mode*”, once in “*header\_mode*”, 1-255 times in “*payload\_mode*”, followed by once in “*crc\_mode*”. The real-time requirement for Wireless LAN is that the system should finish execution in  $(28 + n \cdot 4) \mu$ s with the mode sequence of  $syn^2 header^1 payload^n crc^1$ .

In this example, we want to:

- Prove that the combination of Quasi-static ordering and TDM/round robin scheduling can schedule MCDF graph
- Compute the MCM for each mode can be computed by symbolic simulation
- Perform STS timing analysis of the MCDF graph in certain mode sequence
- Perform SPS timing analysis of the MCDF graph in certain mode sequence

- Calculate the time it takes to get the timing analysis result

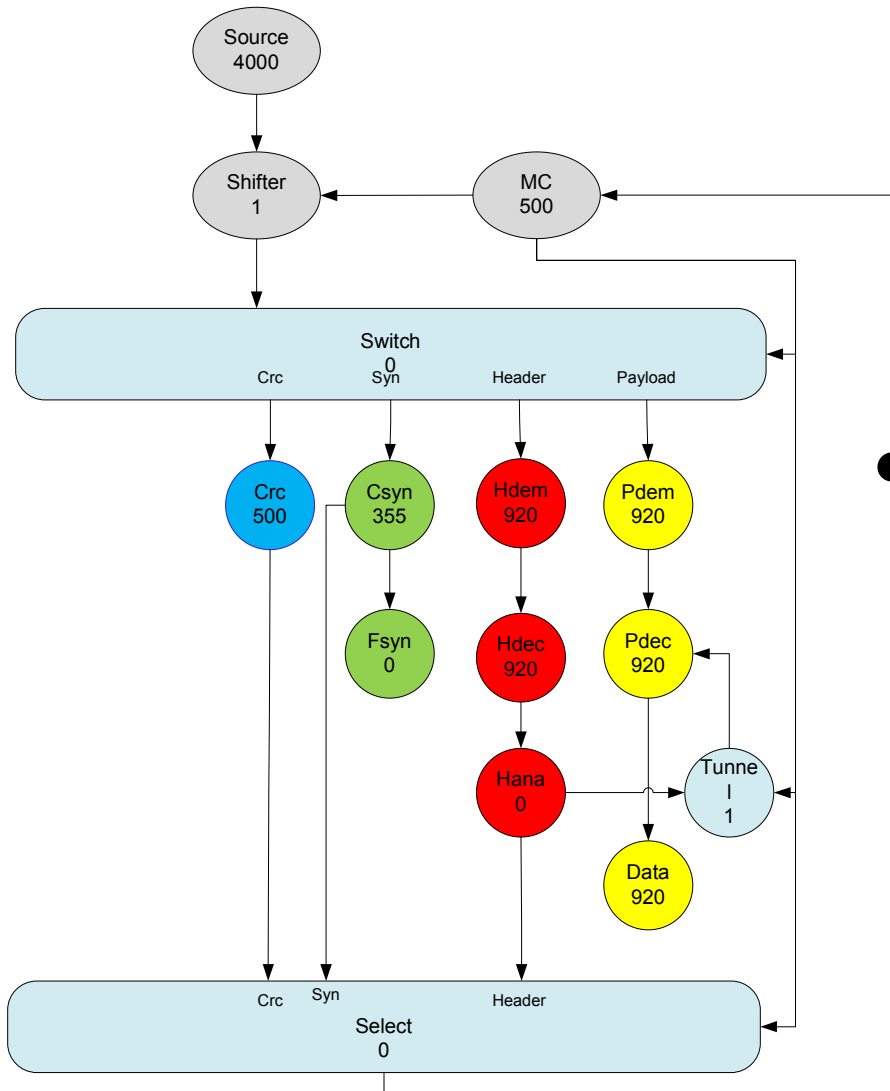


Figure 35 MCDF Graph of Wireless LAN Baseband Receiver

First, we only do the scheduling of the MCDF graph of Wireless LAN. The schedule is checked by computing MCM with the command of:

```
./time Heracles -c2h -how -scc -cyc -f graphfile -s systemfile
```

Here the time wheels of the processors under round robin scheduling are all set to be 3000. The tool computes the MCM for the MCDF graph to be 5997 and finds critical cycle depicted on the top of Figure 36. The time that the tool takes to do the scheduling with MCM computation timing analysis method is 0.26s.

```
Cycle:[ w11a_r_header_decode_1 w11a_r_header_analysis_1_response w11a_r_header_analysis_1 w11a_r_mc_1 w11a_r_tee2_1
switch_header_mode_w11a_r_tunnel1_1 w11a_r_header_demode_1 w11a_r_header_decode_1_response ]
MCM is 5997
Printing Group Bindings
Finished Printing Group Bindings
Current Th: 5997      Max Th: 10000
Finished Printing Schedule
1 solutions found
real    0m0.31s
user    0m0.24s
sys     0m0.02s
```

*Figure 36 Wireless LAN Schedule Result*

According to the inequality 4.16, we can reduce the slice time assigned to each group of actor. For the group mapped on software decoder processor, the time slice can be reduced to 920  $\mu$ s. For the group mapped on EVP processor, the time slice can be reduced to 1425  $\mu$ s. For the group mapped on ARM processor, the time slice can be reduced to 4000  $\mu$ s. For the group mapped on Extern, the time slice can be reduced to 4000  $\mu$ s.

Second, we use symbolic simulation to compute the MCM for each mode which will be further used in static periodic scheduling. The result is shown in Table 4. Here the left column denotes the name of the modes, and the right column denotes the MCM for the mode on its left.

*Table 4 MCM for Each Mode*

Mode Name	MCM
crc	4000
synch	4000
header	5997
payload	4000

Third, we do the scheduling with STS timing analysis method. The command we use is shown below. Here *mosefile* is the input mode sequence file.

```
./time Heracles -c2h -how -scc -sim -simms -f graphfile -s systemfile -ms mosefile
```

The result of self-timed execution of mode sequence is shown in Table 5. The column of *Mode Sequence* denotes that after each scheduling, the system check the real-time requirement with this mode sequence. The column of *time consumption* denotes the time it takes to execute in the mode sequence shown on its left. The column of *Real-Time Requirement* means that the time the result of STS timing analysis should be smaller than this requirement. The *analysis time* denotes that the time the tool spends to find the schedule with STS timing analysis by checking the mode sequence shown on the column of “*Mode Sequence*”. The application could have real-time requirements with more than one mode sequence. In this Wireless LAN example, after each scheduling, it is required to check 255 mode sequences which are  $Synch^2header^1payload^n crc^1$  where  $n = 1-255$ . For different mode sequence, different time constraint is checked. From the Table 5, we can find that it takes 21 minutes 37 seconds for the tool to find the schedule with STS timing analysis method with the 255 mode sequences.

Table 5 Self-timed Execution of Mode Sequence

Mode Sequence	Time Consumption	Analysis Time	Real-Time Requirement
$Synch^2header^1payload^1 crc^1$	26495 ns	1.22s	28000 ns
$Synch^2header^1payload^2 crc^1$	29076 ns	1.25s	32000 ns
$Synch^2header^1payload^3 crc^1$	32497 ns	1.26s	36000 ns
$Synch^2header^1payload^4 crc^1$	36497 ns	1.28s	40000 ns
$Synch^2header^1payload^5 crc^1$	40497 ns	1.32s	44000 ns
$Synch^2header^1payload^{255} crc^1$	1040497 ns	5.55s	1044000 ns
$Synch^2header^1payload^n crc^1$	-	21min37sec	$(28 + n \cdot 4) \cdot 1000ns$

Fourth, we show the scheduling with SPS timing analysis method. The command for Heracles too is:

```
./Heracles -c2h -how -scc -sps -simms -f graphfile -s systemfile -ms mosefile
```

If we have a mode sequence like  $synch^2header^1payload^n crc^1$  ( $N \geq 1$ ), we compute the start times of the actors during mode transition under SPS. We get the finishing time of executing the mode transition under SPS from the information of the start times of the actors. The result is shown in Table 6. We also have the time it takes for the analysis tool to get the analysis result which is 1.82s. If we subtract this time by the time the system takes to schedule and build analysis model, we get the time the tool takes to solve linear programming problem with OCmal-GLPK. The time is  $1.82 - 0.26s = 1.56s$ .

Table 6 Mode Transition under SPS

Mode Transition	Finishing Time	Analysis Time
$synch^1header^1payload^1 crc^1$	22495 $\mu$ s	1.82s

By applying equation 4.9, we can get the static periodic execution time of mode sequence  $synch^a header^b payload^c crc^d$  ( $a, b, c, d \geq 1$ ). The time consumption of executing in mode sequences that we are interested under SPS can be found in Table 7. The graph executes static periodically. Therefore, in order to compute the time consumption of certain mode sequence under SPS, we just add some certain numbers of the MCM of the repeating mode to the finishing time of mode transition. For this example, loads of analysis time is saved. Because the real-time requirement of this example is to check the time consumption of mode sequence  $Synch^2 header^1 payload^n crc^1$  where  $n = 1-255$ . With SPS timing analysis method, we only compute the time consumption of mode transition  $Synch^2 header^1 payload^1 crc^1$ . With simple mathematical computation, the time consumption of mode sequence  $Synch^2 header^1 payload^n crc^1$  can be obtained as shown in Table 7.

Table 7 SPS Timing Analysis of Mode Sequence

Mode Sequence	Time Consumption under SPS	Real-Time Requirement
$synch^1 header^1 payload^1 crc^1$	22495 ns	-
$Synch^2 header^1 payload^2 crc^1$	30495 ns	32000 ns
$Synch^2 header^1 payload^{10} crc^1$	62495 ns	64000 ns
$Synch^2 header^1 payload^{100} crc^1$	422495 ns	424000 ns
$Synch^2 header^1 payload^{255} crc^1$	1042495 ns	1044000 ns

The time consumptions of the mode sequences are all smaller than their real-time requirement. Therefore, the schedule is verified.

From this example, we can find that the timing analysis result of STS is tighter or equal than SPS.

## 7.3 Case 3 Corner Example

In order to show the difference between STS and SPS in the tightness of timing analysis, we artificially build the MCDF graph shown in Figure 37. This artificial graph has 2 modes: mode *red* and mode *blue*. The numbers on the actors are the execution times. The graph file is shown in Appendix D. The graph information is summarized in Table 8.

Table 8 Graph Information of Corner Example

Actor_name	Short_name	Time	Mode	Map_on_Processor
Mode				
Controller	MC	0 s	Null	Proc1
Switch	Switch	0 s	Null	Proc1
Tunnel	Tunnel	0 s	Null	Proc2
A	A	1 s	red	Proc3
C	C	3 s	red	Proc4
B	B	1 s	blue	Proc5

We assume there is always enough resource which means that the actor can start to execute when it is activated. This is done by assigning the full time wheel of a processor to the actor, and all the actors are mapped on the different processors. By doing this, there is no arbitration time for all the actors. We try to do the timing analysis with mode sequence of "red<sup>2</sup>blue<sup>2</sup>".

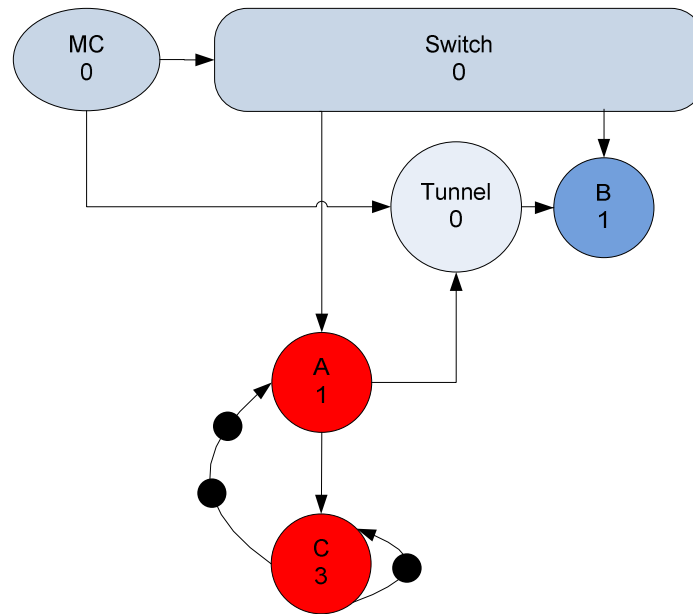


Figure 37 Corner Example

With the STS timing analysis result, we draw the Gantt chart for this STS execution in Figure 38. In this figure, we define  $X_i$  as the  $i^{th}$  execution of actor  $X$ . From the Gantt chart, we can find that the STS execution finishes at time 7, and the start time of actor  $B$ 's second execution is 2.

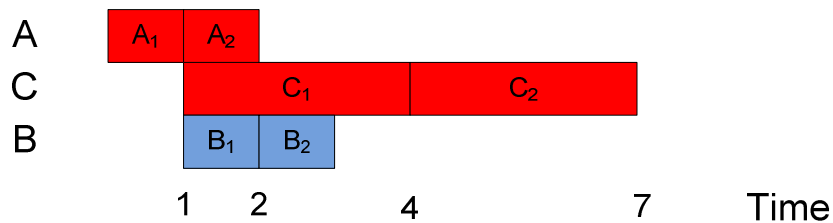


Figure 38 STS Gantt chart

We also try to do SPS timing analysis with the mode sequence of "red<sup>2</sup>blue<sup>2</sup>". The start times in mode *red* and in mode *blue* is shown in Table 9. With these start times, we draw the Gantt chart for SPS execution in Figure 39.

Table 9 Start Times under SPS

Mode Red	Mode Blue
start_A_1_1 -> 0	start_A_1_1 -> 4
start_B_1_1 -> 0	start_B_1_1 -> 3
start_C_1_1 -> 1	start_C_1_1 -> 4
start_mc_1_1 -> 0	start_mc_1_1 -> 3

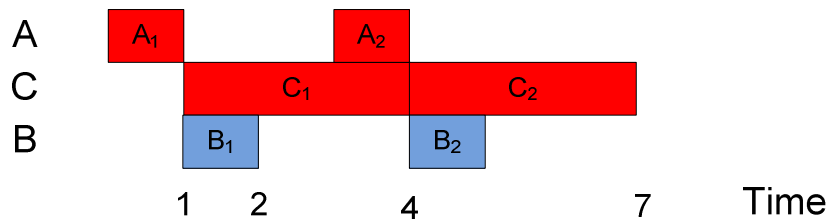


Figure 39 Gantt chart of SPS Execution

Both STS and SPS execution finish in time 7. However, the start time of  $B_2$  with STS is 2 while it is 4 with SPS. In this corner example, it is shown that the timing analysis result of STS is tighter than SPS.

Furthermore, we can do timing analysis to mode sequence  $(red^2blue^2)^n$  which means that the MCDF graph executes in the mode sequence of  $red^2blue^2$  for  $n$  times. We define  $s(B_2, i)$  as the second execution of actor  $B$  in the  $i^{th}$  execution of mode sequence  $red^2blue^2$ . We can find the  $s(B_2, i)$  from the timing analysis result with both STS and SPS. They are represented in the following equations.

$$s(B_2, i)_{SPS} = 6i + 4 \quad i \geq 0 \quad (7-1)$$

$$s(B_2, i)_{STS} = 3i + 2 \quad i \geq 0 \quad (7-2)$$

From the equations above, we can find the difference between  $s(B_2, i)_{SPS}$  and  $s(B_2, i)_{STS}$  is  $3i+2$ .

## 7.4 Summary

From the 3 cases above, we can conclude that when doing timing analysis with a mode sequence, STS timing analysis method computes tighter timing analysis result than SPS timing analysis method.

Further more, the time it takes for STS timing analysis method to get the result depends on the length of the mode sequence. Longer mode sequence leads to longer time consumed to do STS timing analysis. However, the time it takes for SPS timing analysis method to get the result depends on the complexity of the mode transition. For example, if an MCDF graph has  $n$  modes with mode set  $M = \{M_1, M_2, M_3, M_4, M_5, \dots, M_n\}$ . The real-time requirement is represent with a mode sequence like:  $(M_1 M_2 M_3 M_4 M_5 \dots M_n)$ , which means the graph execute in each of its mode once continuously. It has lots of mode transitions which will make the linear programming very complex and requires long time to solve the linear programming problem.

Therefore, we can conclude that if the mode sequence is short, STS timing analysis should be chosen as the timing analysis method. If there is a very long mode sequence which do not have complex mode transition, SPS timing analysis method will be the better choice.



## 8 Conclusions and Future Work

We have presented that the combination of quasi-static ordering and TDM/round robin can schedule the MCDF graphs. Timing analysis methods include MCM computation, STS, and SPS are explained and implemented. We also compare the advantage and disadvantage of the timing analysis methods. STS timing analysis method provides the tightest timing analysis result. It is fast to do STS timing analysis when the mode sequence is short enough. SPS timing analysis can be converted into linear programming problem. With the same mode sequence, it may be faster to do SPS timing analysis than STS timing analysis when the mode sequence has long mode sequence and not so many mode transitions.

However, there are still many aspects of the scheduling strategy and timing analysis methods should be improved. First, for STS timing analysis method, we can try to find a way to avoid simulating all the mode sequences. As illustrated in [8], for every consistent SDF graph, the self-timed execution will finally become periodic. Therefore, the symbolic simulation of mode sequence can stop when enter the periodic phase. Second, we can investigate the time consumption of executing in the mode sequence of  $(A^m B^n)^p$  which means the graph execute in mode sequence of  $A^m B^n$  for  $p$  times. The cyclical of executing in mode sequence  $(A^m B^n)^p$  can be researched. Third, MCDF graphs are assumed to be single rate. Our scheduling strategy can not schedule multi-rate graphs. Further research can be carried out for multi-rate MCDF graphs. Finally, the SPS timing analysis is converted into LP problem. The complexity of the LP problem is discussed in subsection 4.1.4. Further study can be carried out to reduce the complexity of the LP problem.

# References

- [1] K. van Berkel et al. "Vector Processing as an Enabler for Software-Defined Radio in Handheld Devices", *EURASIP Journal on Applied Signal Processing*, vol. 2005, pp. 2613-2625, January, 2005.
- [2] Ulrich Ramacher. "Software-Defined Radio Prospects for Multistandard Mobile Phones", *IEEE Computer*, vol. 40, issue. 10, pp. 62-69, October, 2007.
- [3] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology", *IEEE Trans. On CAD Integrated Circuits and Systems 27(10)*, pp. 1701-1713, October, 2008.
- [4] O. Moreira, F. Valente, and M. Bekooij, "Scheduling Multiple Independent Hard-Real-Time Jobs On a Heterogeneous Multiprocessor", *Proceeding of the 7<sup>th</sup> ACM & IEEE international conference on Embedded Software, Salzburg, Austria*, pp. 59-66, September 30 - October 3, 2007.
- [5] O. Moreira, and M.J.G. Bekooij, "Self-Timed Scheduling Analysis for Real-Time Applications", *EURASIP Journal on Analysis in Signal Processing*, vol 2007, Article ID 83710.
- [6] Edward A. Lee, and David G. Messerschmitt, "Synchronous Data Flow", *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235-1245 September, 1987.
- [7] S. Sriram, and S.S. Bhattacharyya, "Embedded Multiprocessors Scheduling and Synchronization", *Chapter 3 Background Terminology and Notation*, pp.31-53, ISBN: 0-8247-9318-8 MARCEL DEKKER, INC.
- [8] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, et al., "Throughput analysis of synchronous data flow graphs", in *Proceedings of the 6<sup>th</sup> International Conference on Application of Concurrency to System Design (ACSD '06)*, pp. 25-36, Turku, Finland, June, 2006.
- [9] O. Moreira, Jan-David Mol, M. Bekooij, and Jef van Meerbergen, "Multiprocessor Resource Allocation for Hard-real-time Streaming with a Dynamic job-mix", *In Proceedings of the 11<sup>th</sup> IEEE Real Time on Embedded Technology and Applications Symposium*, pp.332-341, 2005.
- [10] <http://caml.inria.fr/ocaml/index.en.html>
- [11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein "Introduction to Algorithms, Second Edition", *Chapter 7 Linear Programming*, pp.770-804, The MIT Press.
- [12] R. Govindarajan and G.R. Gao, "A Novel framework for multi-rate scheduling in DSP scheduling", *In Proceeding s of International Conference on Application-Specific Array Processors*, pp.77-88, Venice, Italy, October, 1993.
- [13] <http://www.gnu.org/software/glpk/>

# Acronyms and Terms

<b>SDR</b>	Software Defined Radio	<b>MIP</b>	Mixed Integer Programming
<b>DF</b>	Data Flow	<b>DVB-T</b>	Digital Video Broadcasting- Terrestrial
<b>GSM</b>	Global System for Mobile Communication	<b>WLAN</b>	Wireless Local Area Network
<b>WCDMA</b>	Wideband Code Division Multiple Access	<b>EVP</b>	Embedded Vector Processor
<b>HSDPA</b>	High-Speed Downlink Packet Access		
<b>DVB-H</b>	Digital Video Broadcasting- Handheld		
<b>GPS</b>	Global Positioning System		
<b>MPSoC</b>	Multi-Processor System on Chip		
<b>VLSI</b>	Very Large Scale Integration		
<b>NOC</b>	Network On Chip		
<b>FIFO</b>	First-In-First-Out		
<b>TDM</b>	Time Division-Multiplex		
<b>MCDF</b>	Mode Controlled Data Flow		
<b>MC</b>	Mode Controller		
<b>MCM</b>	Maximum Cycle Mean		
<b>STS</b>	Self-Timed Schedule		
<b>SPS</b>	Static Periodic Schedule		
<b>LP</b>	Linear Programming		
<b>RR</b>	Round Robin		
<b>OCaml</b>	Objective Caml		
<b>GLPK</b>	GNU linear programming kit		

# Appendix A: MPSOC PLATFORM FILE

processor

```
name="EVP" wheeltime=896000 type=1 sched="roundrobin" weight=100;  
name="SwDecoder" wheeltime=896000 type=2 sched="roundrobin" weight=100;  
name="ARM" wheeltime=896000 type=3 sched="roundrobin" weight=100;  
name="Src" wheeltime=1 type=4 sched="tdma" weight=0;  
name="Lat1" wheeltime=1 type=5 sched="off" weight=0;  
name="Lat2" wheeltime=1 type=6 sched="off" weight=0;  
name="Lat3" wheeltime=1 type=7 sched="off" weight=0;
```

end

## Appendix B: DVB-T Graph File

```
actors
name="synch_acq" exec=57200 slice=80000 group=1 proct=1 mode=1;
name="drop" exec=843 slice=80000 group=1 proct=1 mode=2;
name="dem" exec=57400 slice=80000 group=1 proct=1 mode=3;
name="dec_sink" exec=335500 slice=671000 group=2 proct=2 mode=3;
name="data_out" exec=0 slice=671000 group=2 proct=2 mode=3;
name="mc" exec=427 slice=80000 group=1 proct=1 type="mode_controller";
name="source" exec=1 slice=80000 group=3 proct=3;
name="switch" exec=683 slice=80000 group=1 proct=1 type="switch";
name="select" exec=250 slice=80000 group=1 proct=1 type="join";
arcs
src="source" dst="switch" prod=1 cons=1 type="fifo";
src="mc" dst="switch" prod=1 cons=1 delay=0 type="control";
src="mc" dst="select" prod=1 cons=1 delay=0 type="control";
src="switch" dst="synch_acq" prod=1 cons=1 type="fifo";
src="switch" dst="drop" prod=1 cons=1 type="fifo";
src="switch" dst="dem" prod=1 cons=1 type="fifo";
src="synch_acq" dst="select" prod=1 cons=1 type="fifo";
src="drop" dst="select" prod=1 cons=1 type="fifo";
src="dem" dst="select" prod=1 cons=1 type="fifo";
src="dem" dst="dec_sink" prod=1 cons=1 type="fifo";
src="dec_sink" dst="data_out" prod=1 cons=1 type="fifo";
src="select" dst="mc" prod=1 cons=1 delay=1 type="fifo";
constraints
mud=896000;
end
```

## Appendix C: Wireless LAN Graph File

### actors

```
name="mc" exec=500 slice=1425 group=1 proct=1 type="mode_controller";
name="source" exec=4000 slice=4000 group=2 proct=2;
name="shifter" exec=1 slice=1425 group=1 proct=1;
name="mswitch" exec=0 slice=1 group=1 proct=1 type="switch";
name="mselect" exec=0 slice=1425 group=1 proct=1 type="join";
name="mtunnel" exec=1 slice=1425 group=3 proct=3 type="tunnel";
name="blackhole" exec=1 slice=1425 group=1 proct=1;
name="cfensynch" exec=355 slice=1425 group=1 proct=1 mode=1;
name="ffence" exec=0 slice=1425 group=1 proct=1 mode=1;
name="header_demode" exec=920 slice=1425 group=1 proct=1 mode=2;
name="header_decode" exec=920 slice=2000 group=3 proct=3 mode=2;
name="header_analysis" exec=0 slice=1425 group=1 proct=1 mode=2;
name="payload_demode" exec=920 slice=1425 group=1 proct=1 mode=3;
name="payload_decode" exec=920 slice=2000 group=3 proct=3 mode=3;
name="data_out" exec=920 slice=920 group=4 proct=4 mode=3;
name="crc" exec=500 slice=1425 group=1 proct=1 mode=4;
```

### arcs

```
src="mc" dst="mswitch" prod=1 cons=1 delay=0 type="control";
src="mc" dst="mselect" prod=1 cons=1 delay=0 type="control";
src="mc" dst="mtunnel" prod=1 cons=1 delay=0 type="control";
src="mc" dst="shifter" prod=1 cons=1 type="fifo";
src="source" dst="shifter" prod=1 cons=1 type="fifo";
src="shifter" dst="mswitch" prod=1 cons=1 type="fifo";
src="mswitch" dst="cfensynch" prod=1 cons=1 type="fifo";
src="mswitch" dst="header_demode" prod=1 cons=1 type="fifo";
src="mswitch" dst="payload_demode" prod=1 cons=1 type="fifo";
src="mswitch" dst="crc" prod=1 cons=1 type="fifo";
src="mswitch" dst="blackhole" prod=1 cons=1 type="fifo";
src="cfensynch" dst="ffence" prod=1 cons=1 type="fifo";
src="header_demode" dst="header_decode" prod=1 cons=1 type="fifo";
src="header_decode" dst="header_analysis" prod=1 cons=1 type="fifo";
src="payload_demode" dst="payload_decode" prod=1 cons=1 type="fifo";
src="payload_decode" dst="data_out" prod=1 cons=1 type="fifo";
src="header_analysis" dst="mtunnel" prod=1 cons=1 type="fifo";
src="mtunnel" dst="payload_decode" prod=1 cons=1 type="fifo";
src="crc" dst="mselect" prod=1 cons=1 type="fifo";
src="cfensynch" dst="mselect" prod=1 cons=1 type="fifo";
src="header_analysis" dst="mselect" prod=1 cons=1 type="fifo";
src="mselect" dst="mc" delay=1 prod=1 cons=1 type="fifo";
```

### constraints

```
mud=100000;
```

```
end
```

## Appendix D: Corner Example Graph File

actors

```
name="A" exec=1 slice=10 group=1 proct=1 mode=1;
name="B" exec=1 slice=10 group=2 proct=2 mode=2;
name="C" exec=3 slice=10 group=3 proct=3 mode=1;
name="mc" exec=0 slice=10 group=4 proct=4 type="mode_controller";
name="switch" exec=0 slice=10 group=4 proct=4 type="switch";
name="tunnel" exec=0 slice=10 group=5 proct=5 type="tunnel";
```

arcs

```
src="mc" dst="switch" prod=1 cons=1 delay=0 type="control";
src="mc" dst="tunnel" prod=1 cons=1 delay=0 type="control";
src="switch" dst="A" prod=1 cons=1 type="fifo";
src="switch" dst="B" prod=1 cons=1 type="fifo";
src="A" dst="tunnel" prod=1 cons=1 delay=0 type="fifo";
src="tunnel" dst="B" prod=1 cons=1 delay=0 type="fifo";
src="A" dst="C" prod=1 cons=1 type="fifo";
src="C" dst="A" prod=1 cons=1 delay=2 type="fifo";
```

constraints

```
mud=10;
```

```
end
```