
Modelling and simulation of guaranteed throughput channels of a hard real-time multiprocessor system

A.J.M. Moonen

Information and Communication Systems
Department of Electrical Engineering
Eindhoven University of Technology

Eindhoven, 30th January 2004

Carried out	May 2003 - January 2004 at the Philips Research Laboratories, Eindhoven
Professor	prof.dr.ir. J.L. van Meerbergen
Supervisor	dr.ir. M.J.G. Bekooij

Abstract

Consumers have high expectations about the audio and video quality delivered by media processing devices. Applications running on these media processing devices, like high quality multi-window television and MPEG-video decoders, require a computational performance in the order of 10-100 giga operations per second. Embedded systems are quickly evolving towards heterogeneous multiprocessor systems, which can provide this performance. An important challenge is to build these systems in a way that they exhibit a predictable temporal behavior. This way it can be guaranteed that the applications will meet their deadlines.

The objective of the graduation project was to make a simulation model of the multiprocessor system that can be used to derive the timing behavior of an application that is executed on the system. A Synchronous Data Flow (SDF) graph is introduced in which computation as well as communication is expressed. This SDF graph is constructed in such a way that the worst-case arrival times of data can be observed. To simulate the SDF graph a simulator is built which makes it possible to verify the functional as well as the temporal behavior. From the simulation results it can for example be concluded how the performance of the application can be improved.

The last objective of the graduation project was to implement a Communication Assist (CA). The CA is a special piece of hardware which is mainly introduced to decouple computation from communication. The CA can be seen as an autonomous DMA with some additional functionality. In this thesis the functionality of the CA is described in more detail and the CA is implemented in such a way that the system exhibits a predictable behavior.

Preface

This thesis in front of you is a result of my graduation project performed at Philips Research Laboratories in Eindhoven. I would like to thank professor Jef van Meerbergen for giving me the opportunity to graduate within the Hijdra project of Philips Research and for his feedback which helped me to see the addressed problem from another perspective. The obtained research results would not have been possible without the help and confidence of my supervisor Marco Bekooij. Especially due to the time he invested and the discussions we had I was able to complete the objectives of the graduation project. At last I would like to thank the team members of the Hijdra project for their contribution and feedback, but most of all for giving me the feeling to be a team member of the Hijdra project.

Contents

Abstract	i
Preface	ii
1 Introduction	1
1.1 Background	1
1.2 Problem definition	2
1.3 Outline of the thesis	2
2 Multiprocessor Architecture	3
2.1 Multiprocessor Template	3
2.2 Actors on Processors	4
2.3 Communication Assist	4
2.4 Network on Silicon	5
2.5 Communication Channel	6
3 Model of computation, Communication & Scheduling	7
3.1 Synchronous Dataflow	7
3.1.1 Homogeneous Synchronous Dataflow	8
3.1.2 Cycles	9
3.1.3 Steady State	9
3.2 Models and their composition	12
3.3 Arbitration/Scheduling models	14
3.3.1 TDMA, N	14
3.3.2 RR, N	18
3.3.3 TDMA, $\leq N$	21
3.3.4 RR, $\leq N$	24
3.4 Model of the Communication Channel	27
4 Transaction Level Simulator	29
4.1 Functions overview	29
4.1.1 FIFO versus communication channel	29
4.1.2 The write function	30
4.1.3 The read function	31
4.1.4 The peek function	31
4.1.5 The peek wait function	32
4.1.6 The select function	32

4.1.7	The steady state function	32
4.2	Hard-Realtime Producer-Consumer Example	33
4.2.1	The Main Program	34
4.2.2	The Process Network	34
4.2.3	The Producer	38
4.2.4	The Consumer	38
4.2.5	Running the example	40
4.3	Implementation	40
4.3.1	Communication Channel	42
4.3.2	Steady State	43
5	Communication Assist Design	47
5.1	Assignment definition	47
5.2	Interface	47
5.3	Arbitration and scheduling performed by the Communication Assist	48
5.3.1	Arbitration of the Local-bus inside a tile	48
5.3.2	Scheduling between Network Connections	48
5.4	Configuration	53
5.5	Implementation	53
5.5.1	FIFO Status	53
5.5.2	Fire	53
5.5.3	Increment	55
5.5.4	Load	55
5.5.5	Round Robin	55
5.5.6	Switch Control	56
6	Conclusions and recommendations	58
	List of Figures	60
	List of Tables	62
	Bibliography	63
A	Related Work	65
B	The Simulation Cycle	67
C	HAPI main class diagram	69
D	Parameters and variables in the class hChannel	71
E	Scheduling implementation model	73
F	Architecture of the CA	77
G	Examples C-HEAP	80
	Distribution list	82

Chapter 1

Introduction

This thesis describes the modelling of communication channels of a multiprocessor architecture. These models are applied in a simulator. By making use of these models the worst-case timing behavior of an application on the architecture can be observed. From the simulation results conclusions can be drawn how to adapt for example the mapping of the application.

The organization of this chapter is as follows. In Section 1.1 some background information will be provided in order to become familiar with the context of the problem that is addressed in this thesis. The subject of this graduation project is defined in Section 1.2. Finally the outline of the thesis will be presented in the last section of this chapter.

1.1 Background

Embedded systems of today contain typically busses for communication. It is likely that this will change in the near future due to deep sub-micron problems as well as due to mapping problems. Especially the wiring delay will become dominant over the gate delay. The number of clock domains will increase and meeting a deadline will become increasingly difficult. As a consequence the design process will become less straightforward. Furthermore, central resources like busses or memories limit the scalability of these architectures. Therefore are architectures quickly evolving towards heterogeneous multiprocessor architectures consisting of tiles that communicate via an on-chip network [18] [19]. These tiles can contain general purpose processors, digital signal processors or application domain specific processors. The network consists of routers, network interfaces and network links.

Applications like high quality multi-window television with picture enhancement, graphics and MPEG-video decoders require a computational performance in the order of 10-100 giga operations per second. Such a performance can not be provided by a single processor and requires a multiprocessor system. An important problem of these systems is that their behavior is less predictable. A system is predictable if it respects predefined timing and quality requirements. A multiprocessor template will be introduced in order to handle the design of these complex systems. Models, analysis techniques and multiprocessor simulation tools will be introduced to design these complex systems.

Three simulation abstraction levels are assumed in this thesis. By going one level deeper more details are considered and the simulation speed will be slower. Table 1.1 shows the three levels of simulation.

Level 1	Behavior without timing
Level 2	Transactions with worst-case timing
Level 3	Cycle true

Table 1.1: Simulation abstraction levels.

Simulation of level 1 is used for verification of the functional behavior of the system; timing for example is not considered. At this level it is possible to see the data transport between processes. The level 2 simulation is introduced to simulate communication between processes with worst-case timing. Therefore the worst-case time that data can arrive is made visible. By doing this it is possible to reason about the throughput and the latency. The last level of simulation is the cycle true simulator. At this level the implementation is defined in great detail and a cycle true simulation of the implementation is possible. The behavior observed at every clock cycle of the simulation is the same as in the implementation.

1.2 Problem definition

There are three objectives of the graduation project. The first one is to come up with a model of the implementation that can be used in the level 2 simulator. The second objective is to build the level 2 simulator. The last objective is to design a communication assist.

1.3 Outline of the thesis

This thesis is organized as follows. First the multiprocessor template will be introduced in Chapter 2. Chapter 3 introduces a model which is used in the level 2 simulator. After explaining some basic building blocks the communication between two processes on different processors is modeled. This model is used in the level 2 simulator that is explained in Chapter 4. Chapter 5 describes the implementation of the communication assist. Finally in Chapter 6 conclusions are drawn and recommendations are given.

Chapter 2

Multiprocessor Architecture

As explained in the first chapter, embedded systems of today are evolving towards heterogeneous multiprocessor systems. This chapter will introduce such system and will describe its components in detail. Also the communication between two processes on different processors is examined.

The outline of this chapter is as follows. First a multiprocessor template will be introduced in Section 2.1. Subsequently some terminology is introduced in Section 2.2. Then the communication assist is described in Section 2.3 and the network is described in Section 2.4. The communication between two processes is explored in the last section of this chapter.

2.1 Multiprocessor Template

The multiprocessor template that is proposed by the Hijdra project [3] is shown in Figure 2.1. This template can be for example a homogeneous as well as a heterogeneous multiprocessor, depending whether the tiles are different or not. The tiles contain a processor with some local memory and a connection to the network, as shown in Figure 2.1. It is also possible to have a tile with a local memory but without a processor. The tiles are connected by an on-chip network. Every tile has a Network Interface (NI) which is connected to a router in the network. Routers are connected to network links in the desired network topology. Every tile might contain a processing unit, a

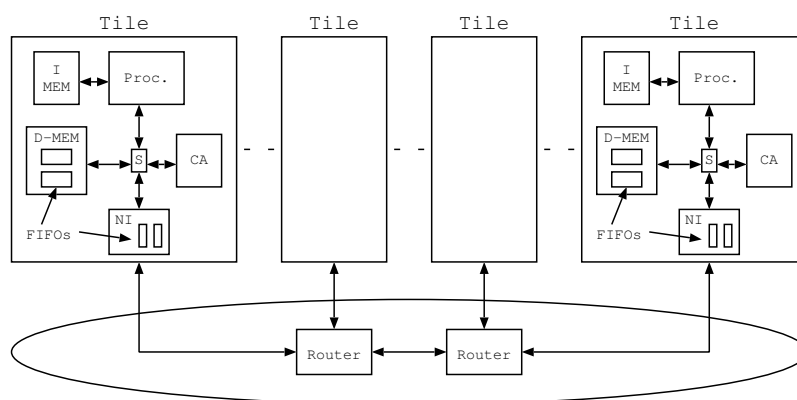


Figure 2.1: Multiprocessor template.

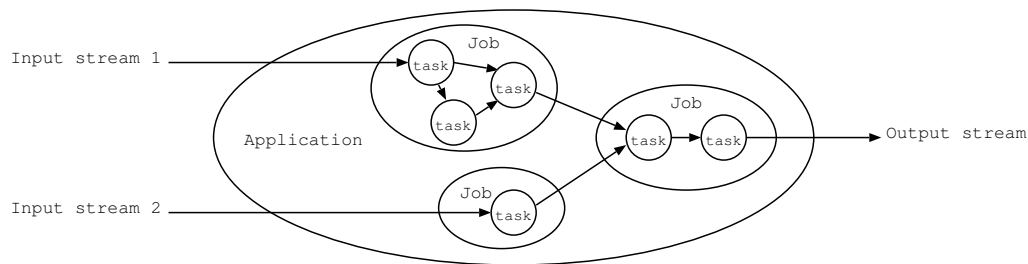


Figure 2.2: An application which consists of jobs and jobs which consist of tasks.

general purpose processor, a digital signal processor or an application domain specific processor. The processor in a tile has a separate instruction memory (I-mem) and data memory (D-mem). The advantage is that fetching an instruction and data load/store operations can be performed in parallel. It makes the worst-case execution time of an instruction shorter. In a tile there is also a block which is responsible for the data transfer between the data memory and the NI. This block is called the Communication Assist (CA), which is described in more detail in Section 2.3. The intra connection in the tile is done by a switch, which is necessary because both the processing unit and the CA need to access the data memory. The advantage of tiles with their own memory is that the latency for the processor to access its own memory is lower than in the case of a central memory. The advantage of the network in comparison with busses is the scalability.

2.2 Actors on Processors

First some terminology will be defined. An application is a collection of jobs which are executed on the multiprocessor system at a certain point in time. Jobs can be started and stopped by the user. Jobs consist of tasks. These tasks are created at the start of the job and are deleted when a job stops. An application is depicted in Figure 2.2. Task with some well defined properties are called actors. One of the properties is that an actor has a worst-case execution time. In this time it consumes a fixed number of tokens from every input and produces a fixed number of tokens on every output. A token is defined as a container in which a fixed amount of data can be stored. During every execution of the actor the same amount of tokens are consumed and produced.

It is possible to assign more than one actor to a processor. Actors assigned to the same processor are executed successively. In this thesis it is assumed that they are executed according to a predefined cyclic list. An actor checks on every input for data and on every output for space. When all the required input data and output space is available the actor starts its execution. After the actor finishes its execution or if the required input data or output space is not available it returns so that the next actor in the list can check if it can execute.

2.3 Communication Assist

The CA is an important block inside the tile. As mentioned before its job is the transportation of data between the data memory and the NI. Inside the data memory and the NI are First In First Out (FIFO) queues. Two processes can only communicate with each other if a connection exists between them. A connection between two processes that are executed on the same processor is performed by

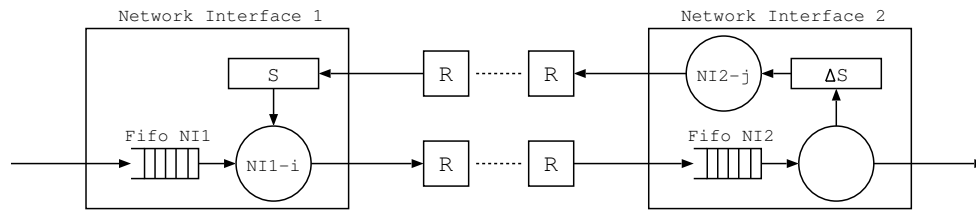


Figure 2.3: Implementation of a guaranteed throughput channel of the AETHER network.

a FIFO in the data memory. When the two processes are on different processors the connection has to go over the network. In this case the CA is responsible for the data transport between the FIFO in the data memory and the FIFO in the NI. Because there can be multiple connections the CA has to make sure that the data in every connection is transferred. Therefore the CA has to share its time between all the connections, which requires an arbitration mechanism in the CA. This arbitration mechanism should be predictable. Predictable means that a worst-case time can be derived in which the data is transferred. Another assignment of the CA is that it should arbitrate the switch in such a way that the processor as well as CA will at least obtain a predefined amount of time to access the data memory.

2.4 Network on Silicon

Communication between the tiles is done via an on-chip network. The network that is assumed is the AETHER network [18] [19]. This network consists of network interfaces and routers. Network interfaces connect the tiles to the network and routers and network links are connecting the network interfaces. The AETHER network provides connections with a specified service, among others Guaranteed Throughput (GT) service and Best-Effort (BE) service. But only the GT service is used due to the predictability requirement. In this case predictable means that guarantees can be given in the amount of time it takes for the network to transfer the data.

Figure 2.3 gives an overview of one connection between a producing tile and a consuming tile. The producing tile is connected to NI 1 and the consuming tile is connected to NI 2. The NI of the AETHER network is working with a Time Division Multiple Access (TDMA) wheel. The TDMA wheel is split up in a number of slots. The size of one slot can be chosen and has a granularity of a flit (1 flit=3 words and 1 word=32 bits). When setting up a connection one should reserve some bandwidth by reserving a number of slots. The NI checks for input data at the beginning of every slot. A minimal amount of data can be defined before the data is sent over the network. The advantage of this threshold can be that the network sends only data when the slots are filled with data. When slots reserved for GT connections are not filled and therefore not sent over the network they can be filled with data from the BE connections. Every consecutive block of slots has to send first a header of one word. In this header the path through the network is encoded. In the worst-case scenario every slot has its own header. This happens when there are no consecutive slots available.

The GT connection is working with a credit path, as shown in Figure 2.3. The NI 1 at the producing tile has, at the start of the connection, the knowledge of the capacity of *Fifo NI2* at the consuming tile. This capacity is stored in register *S*. Every time NI 1 sends something over the channel, it decreases the register *S* with the number of words that were sent over the channel. When the

register S reaches zero it stops sending data until it receives new credits form NI 2. When the consuming tile takes data from *Fifo NI2* a register ΔS is updated. The register represents the number of words that now became available in *Fifo NI2*. The NI 2 will send the credits ΔS back to NI 1 when the TDMA wheel of NI 2 reaches a reserved slot. The NI 1 will receive ΔS and add ΔS to register S .

2.5 Communication Channel

In this chapter the multiprocessor is explained in more detail so that an abstract view can be made of a connection between two processes on different processors. Such a connection is called a communication channel. Figure 2.4 depicts a communication channel between a producing process P1 and a consuming process P2. The producing process P1 writes the output data to its local data memory (*Fifo MEM1*). The CA of the producing tile is responsible for transporting the data from the data memory (*Fifo MEM1*) to the NI (*Fifo NI1*). The network will transfer the data from NI 1 (*Fifo NI1*) to its destination NI 2 (*Fifo NI2*). The CA in the consuming tile transfers the data to the data memory (*Fifo MEM2*) of its processor. The process P2 will consume its data from the data memory (*Fifo MEM2*). All the blocks in the multiprocessor architecture run in parallel. But the data will pass the communication channel as depicted in Figure 2.4.



Figure 2.4: Abstract view of the communication channel between P1 and P2.

The network node in Figure 2.4 can be replaced by the network implementation which is shown in Figure 2.3. This results in a more detailed view of the communication channel between the processes P1 and P2 as shown in Figure 2.5. In the next chapter the communication channel will be expressed in a formal model which is applied in the level 2 simulator.

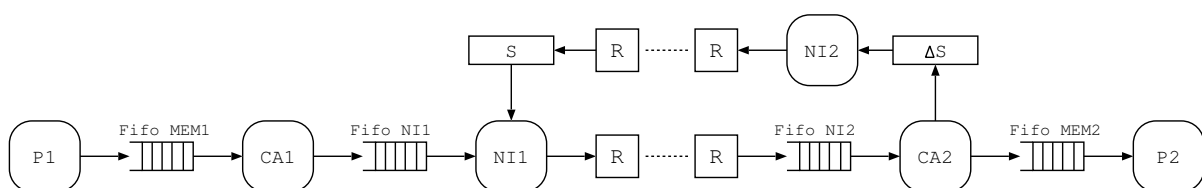


Figure 2.5: More detailed view of the communication channel between P1 and P2.

Chapter 3

Model of computation, Communication & Scheduling

Three levels of simulation are defined in Chapter 1. The simulation level 2 will simulate transactions with worst-case timing. A formal model will be introduced which is the basis for the simulation at this level. This formal model is used to model the communication channel between two processes on different tiles. By doing analysis on the formal model the minimal throughput can be derived.

The outline of this chapter is as follows. The formal model is introduced in Section 3.1. Section 3.2 explains the requirements for modelling the implementation. Then some techniques will be explained to model scheduling algorithms in Section 3.3. Eventually in Section 3.4 a model of the complete communication channel is derived.

3.1 Synchronous Dataflow

The level 2 simulator is based on the Synchronous Data Flow (SDF) graph model [17] [20] [3] proposed by Lee and Messerschmitt. The SDF model makes it possible to derive the minimal throughput and maximum latency with analytical techniques. An example of an SDF graph is given in Figure 3.1. The nodes in the SDF graph are called actors. Actors are tasks with a well defined input/output behavior and an execution time which is less than or equal to a specified worst-case execution time. The black bullets on the edges are tokens. A token is a container in which a fixed amount of data can be stored. The SDF model poses restrictions on the firing of actors. The number of tokens consumed by an actor on each input edge is a fixed number that is known at compile time. The number of tokens consumed on each of its input edges is annotated by numbers at the head of the edges. An actor is fired as soon as on every incoming edge at least the number of tokens are

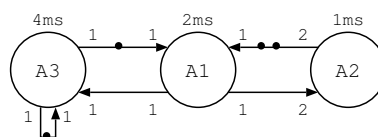


Figure 3.1: An SDF graph.

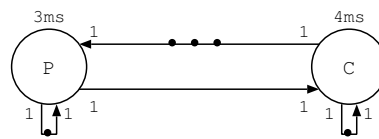


Figure 3.2: SDF model of a FIFO with a capacity of 3 tokens between a producing and consuming actor.

available as is specified at the specific edge. These tokens are consumed from the input edge before the execution of an actor finishes. The number of tokens produced by an actor on each output edge is a fixed number that is known at compile time. The number of tokens produced on each of its output edges is annotated by numbers at the tail of the edges. The tokens at the output edge are produced before the execution of an actor finishes. The internal state of an actor is made explicit with a self edge, like it is for actor A3 in Figure 3.1. This self edge is given one initial token such that the next execution cannot start before the previous execution is finished.

Tokens in the SDF model are consumed in the same order as they are produced. Therefore, FIFOs with a fixed capacity can be modelled in an SDF with a data edge from the consuming actor C to the producing actor P, as is depicted in Figure 3.2. The number of initial tokens on this edge should be equal to the maximum number of tokens that can be stored in the FIFO. This number is called the FIFO capacity.

3.1.1 Homogeneous Synchronous Dataflow

The advantage of the SDF model is that analysis can be performed on the model. The SDF graph is expanded into a Homogeneous SDF (HSDF) before it can be analyzed. The HSDF is a special kind of an SDF graph in which the execution of an actor results in the consumption of one token from every incoming edge and the production of one token on every outgoing edge. Every SDF can be transformed into an HSDF as is described in [20]. The HSDF graph that is obtained after transformation of the SDF graph in Figure 3.1 is shown in Figure 3.3. All the actors of the HSDF are executed in a self-timed fashion, which is defined as a sequence of executions of HSDF actors in which every actor start immediately when there are sufficient tokens on each of its inputs.



Figure 3.3: An HSDF graph

3.1.2 Cycles

The HSDF graphs of jobs in this thesis are assumed to be strongly connected due to the use of bounded FIFOs. A strongly connected HSDF graph in which the token order is preserved, has some useful properties. A property of such an HSDF graph is that it is deadlock-free if every simple cycle in the HSDF graph has at least one token. A simple cycle is a cycle that applies an actors maximally once. Another property of an HSDF graph is that the execution is monotonic, which means that decreased execution time of an actor cannot lead to a later start time of any actor in the graph.

By examining all the simple cycles in the HSDF graph it is possible to derive the minimal throughput of the job. Throughput is derived by calculating the Maximum Cycle Mean (MCM) of an HSDF graph. Take for example the graph $G(V, E)$ where V is a set of nodes representing actors and E the set of directed edges representing data dependencies. The actors have a Worst-Case Execution Time $WCET(v)$. Let C be the set of simple cycles in the graph G and $tokens(c)$ the number of initial tokens on a specific cycle $c \in C$. Then the MCM of this graph is equal to:

$$MCM(G) = \max_{c \in C} \left[\frac{\sum_{v \text{ on } c} WCET(v)}{tokens(c)} \right] \quad (3.1)$$

For example the MCM on the HSDF graph in Figure 3.3 equals 10 ms. This MCM is a result of the critical cycles $(A3', A3'', A1')$ and $(A3', A1'', A3'')$.

The MCM formula can also be used to determine the number of initial tokens on a cycle. To guarantee a throughput of $1/T$ one has to make sure that the $MCM \leq T$. The number of initial tokens can be chosen on places where a bounded FIFO was modelled. Indirect it is then possible to determine the FIFO capacities to guarantee a certain minimal throughput. This theory is not straightforward and out of scope of this document. The main difficulty is that the HSDF graph depends on the number of initial tokens in the SDF graph. Therefore the HSDF graph depends on the FIFO capacity. More information can be found in the literature [20].

3.1.3 Steady State

A property of strongly connected HSDF graphs with fixed actor execution times is that they enter a periodic regime [3] [17]. A formal definition of a periodic regime is as follows. There exist $K \in \mathbb{Z}^*$, $k \in \mathbb{Z}^*$ and $N \in \mathbb{Z}^*$, such that for all $v \in V$, $k > K$ the start time $s(v, k + N)$ of actor v in iteration $k + N$ is described by:

$$s(v, k + N) = s(v, k) + N \cdot MCM \quad (3.2)$$

Equation 3.2 states that the execution enters a periodic regime after K executions of an actor in the HSDF graph. The time of one period is $N \cdot MCM$ and is called the *cycle time* in the level 2 simulator. The number of MCM iterations in one period is denoted by N . The system is in the transient state before it enters the periodic regime. The only way to determine start times of the actor during the transient state is by simulating the SDF graph into the level 2 simulator. A simple example is used to makes the transient state and the periodic regime more visible. Take for example the SDF graph in Figure 3.2 which is also an HSDF graph. Equation 3.2 can be rewritten into Equation 3.3. This

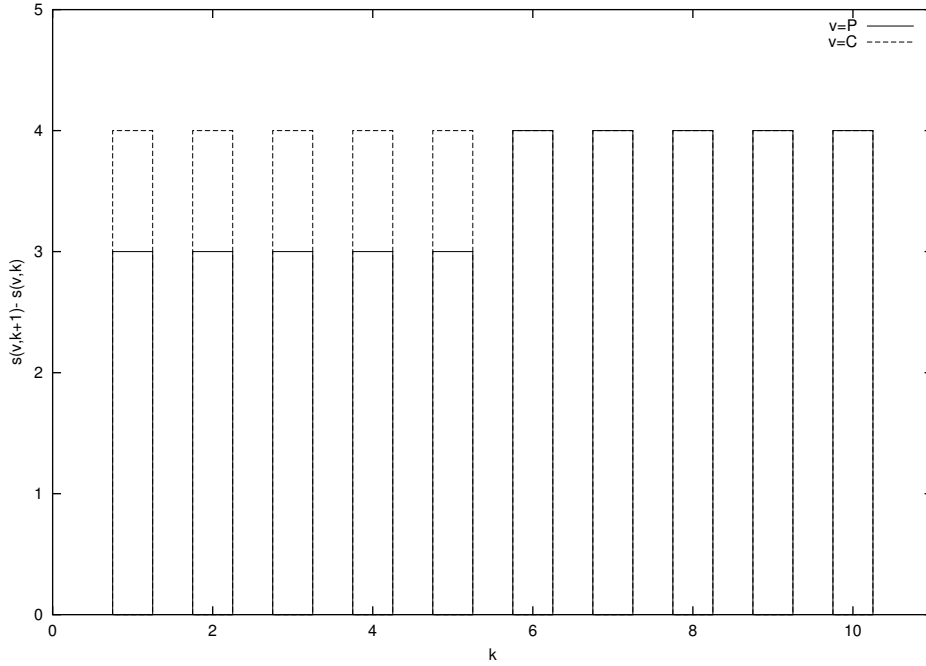


Figure 3.4: Plotting for all the actors the difference between successive start times of the specific actor.

equation holds if the HSDF is in the periodic regime, in other words when $k > K$. Figure 3.4 shows the difference in time between the execution k and $k + N$ of actor $v \in \{P, C\}$.

$$s(v, k + N) - s(v, k) = N \cdot \text{MCM} \quad (3.3)$$

By examining Figure 3.2 it is clear that the cycle which contain the self-edge of the consuming actor C is the critical cycle. Therefore the MCM should be equal to 4 ms. The value of $N = 1$. How the value of N can be derived from the HSDF graph can be found in [2]. Equation 3.3 states that all the actors in Figure 3.4 should be executed every $N \cdot \text{MCM} = 4\text{ms}$. By looking at Figure 3.4 it can be seen that the consuming actor C executes every 4 ms from the beginning. The producing actor P executes every 4 ms if $k > 5$. Therefore the periodic regime enters after 5 executions of the producing actor P.

Another way to detect the steady state is not by observing the start times of the actors but by observing the arrival times of the tokens. This is possible because there is a relation between the arrival times of the tokens and the start times of the actors.

Theorem 3.1 *When the start times of the actors is periodic then the arrival times of the tokens is also periodic.*

More precisely, there exist an $L \in \mathbb{Z}^*$, $l \in \mathbb{Z}^*$ and $N \in \mathbb{Z}^*$, such that for all $e \in E$, $l > L$ the arrival time $\text{ar}(e, l + N)$ of the $l + N$ th token on the edge e is described by:

$$\text{ar}(e, l + N) = \text{ar}(e, l) + N \cdot \text{MCM} \quad (3.4)$$

Proof: This follows from the periodicity of start times, because the arrival times of the tokens is directly related to the start times of the actors. For every firing of an actor v a token is produced on all the output edges of the actor. The arrival time of the output token is the start time of actor v plus the worst-case execution time of the actor, as shown in Equation 3.5. When the start times enter a periodic regime the arrival times also enter a periodic regime, because of the fixed worst-case execution times of the actor. \square

$$\forall_{\text{output edges } e \text{ of } v} [\text{ar}(e, l_e) = s(v, k) + \text{WCET}(v)] \quad (3.5)$$

By observing the arrival times of the tokens it is possible that the steady state is detected later than by looking at the start times of the actors. The reason is that the start time is the maximum of all the arrival times of the tokens on the input edges, as shown in Equation 3.6. If the token arrival time of a specific input edge e of actor v is periodic and the arrival time of the token is the dominant factor of the $\max[]$ function, the start time of the actor v is also periodic. But it does not have to be that the token arrival times of all the input edges of actor v are periodic. That is the reason that the start times of the actors can already hold to Equation 3.2 before the arrival times of the tokens hold to Equation 3.4.

$$s(v, k) = \max_{\text{input edges } e \text{ of } v} [\text{ar}(e, k)] \quad (3.6)$$

Theorem 3.2 *If the arrival times of the tokens is periodic then the start times of the actors is also periodic.*

Proof: Equation 3.4 holds if the arrival times of the tokens are in a periodic regime. The start times of the actors are according to Equation 3.6 and as a consequence it must be the case that also Equation 3.2 holds. \square

To implement the steady state detection for all the actors or tokens in the level 2 simulator is not straightforward. The simulator saves the state at a certain time t . The state of one token on edge e at time t is defined as: $t - \text{ar}(e, k)$. When comparing the state of time $t + N \cdot \text{MCM}$ with the state of time t , the states should be equal in case of a periodic regime. This can easily be verified by Equation 3.7 because this equation is equivalent to Equation 3.4, as shown below:

$$\overbrace{(t + N \cdot \text{MCM}) - \text{ar}(e, k + N)}^{\text{state}(t+N \cdot \text{MCM})} = \overbrace{t - \text{ar}(e, k)}^{\text{state}(t)} \quad (3.7)$$

$$t - \text{ar}(e, k + N) = t - \text{ar}(e, k) - N \cdot \text{MCM} \quad (3.8)$$

$$\text{ar}(e, k + N) = \text{ar}(e, k) + N \cdot \text{MCM} \quad (3.9)$$

The Producer-Consumer example of Figure 3.2 can be used to make the theory more clear. The state of an HSDF is defined by the place and the waiting times ($t - \text{ar}(e, k)$) of all the tokens in the HSDF graph at a certain time t . All the state changes of the HSDF graph is depicted in Figure 3.5.

This figure shows that the first time that two identical states are identical is at $t = 27\text{ms}$. The state at $t = 27\text{ms}$ is equal to the state at $t = 23\text{ms}$. The period between the states is $27 - 23 = 4\text{ms}$ and is equal to $N \cdot \text{MCM}$. It was given that $N = 1$ so the MCM is 4 ms. This example shows that observation of the arrival times of the tokens results in a later detection of the steady state than in the case the start times of the actors are observed. The periodic regime is entered after 7 executions of the producing actor P, according to Equation 3.4. While the periodic regime is entered after 5 executions of the producing actor P, according to Equation 3.2.

3.2 Models and their composition

The objective of modeling the implementation is to come up with an SDF model that is conservative. Conservative means that the start times of the actors and the arrival times of the tokens in the SDF model are worst-case. In other words it means that the start times of the actors in the implementation should be earlier or equal than the start times of the actors in the SDF model:

$$s_{\text{impl}}(v, k) \leq s_{\text{sdf}}(v, k) \quad (3.10)$$

It is possible to make a similar requirement for the arrival times of the tokens, as shown in Equation 3.11. This is not a necessary requirement but a sufficient requirement to guarantee Equation 3.10.

$$ar_{\text{impl}}(e, k) \leq ar_{\text{sdf}}(e, k) \quad (3.11)$$

In the case that the implementation consist of separate blocks it is easier to model every block separately. By doing this it has to be guaranteed that every model of the basic block is conservative. When all the individual models satisfy this requirement, it is sure that the composition is also conservative. Take for example Figure 3.6 and assume that for the input connection P holds $ar_{\text{impl}}(P, k) \leq ar_{\text{sdf}}(P, k)$. If the SDF model of the first block is conservative then $ar_{\text{impl}}(Q, k) \leq ar_{\text{sdf}}(Q, k)$ holds for the connection Q. The same goes for the second block. When $ar_{\text{impl}}(Q, k) \leq ar_{\text{sdf}}(Q, k)$ holds for the input of the second block and the SDF model is conservative then $ar_{\text{impl}}(R, k) \leq ar_{\text{sdf}}(R, k)$ holds for the connection R. Now it is clear that the composition holds to Equation 3.11 for every $e \in \{P, Q, R\}$. Therefore the conclusion can be drawn that the composition is also conservative.

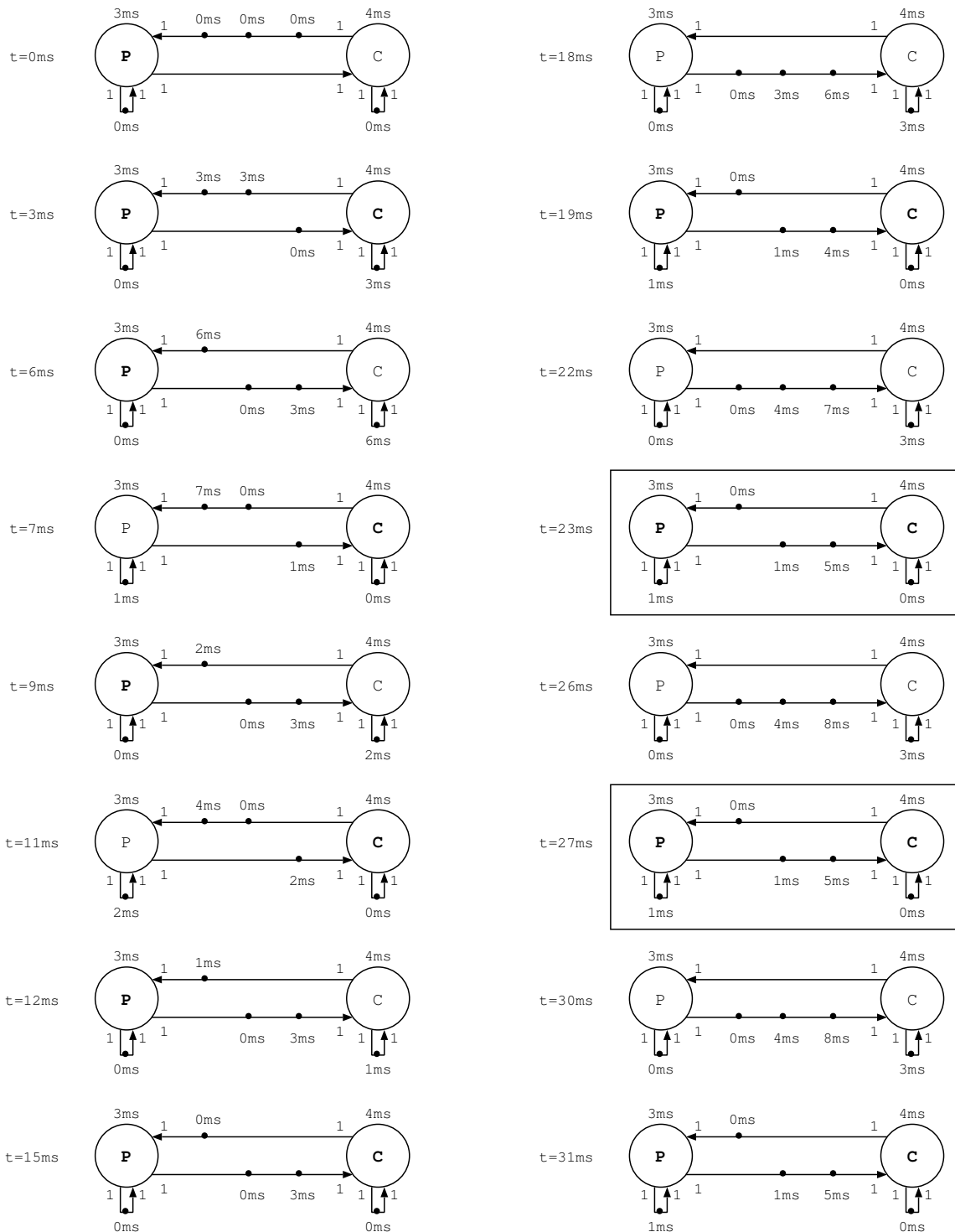


Figure 3.5: The state changes of an HSDF graph from a simple Producer-Consumer example.

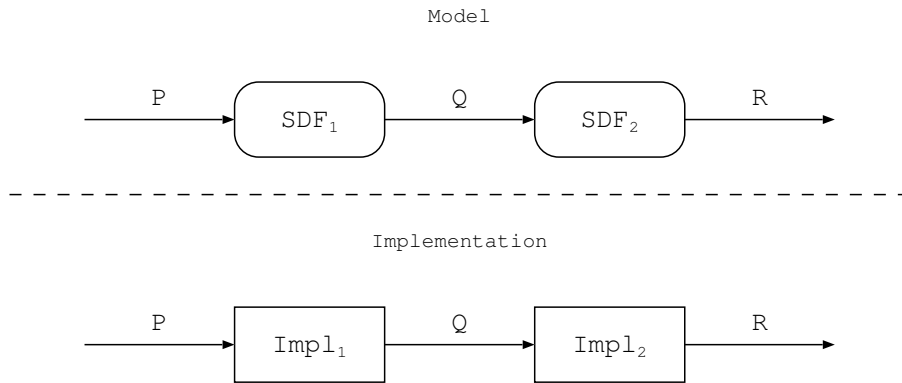


Figure 3.6: Composition of basic blocks and their SDF models.

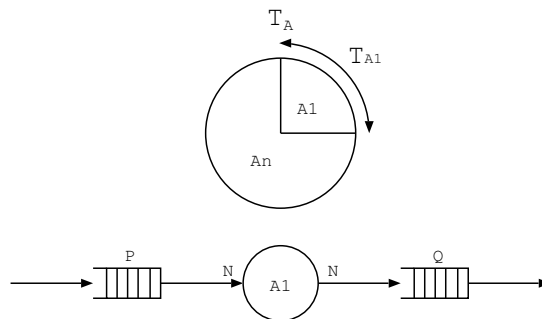


Figure 3.7: A implementation of the TDMA wheel with a threshold of N tokens.

3.3 Arbitration/Scheduling models

In this section two predictable arbitration/scheduling techniques will be explained. These are Time Division Multiple Access (TDMA) and Round Robin (RR). For every technique there are two variants: with a threshold of N tokens and without a threshold but a maximum of N tokens. All these techniques check only once per period for sufficient input tokens. In the case of TDMA the period is the time of the TDMA wheel and is fixed. The period of RR is not fixed but it is bounded. For the convenience some shorter names are defined:

TDMA, N	TDMA with a threshold of N tokens
RR, N	RR with a threshold of N tokens
TDMA, $\leq N$	TDMA with no threshold but a maximum of N tokens
RR, $\leq N$	RR with no threshold but a maximum of N tokens

Table 3.1: The arbitration/scheduling models that will be explained.

3.3.1 TDMA, N

In this subsection an SDF model is derived for TDMA arbitration with a threshold of N tokens. An implementation of a TDMA wheel is shown in Figure 3.7. The time T_A is the time for one rotation

of the TDMA wheel. The time T_{A1} is the time of the reserved slot. The first step is to express the characteristics of the implementation in an equation. The next step is to express the SDF model in an equation. When both equations are defined it is possible to proof that if $\text{ar}_{\text{impl}}(P, k) \leq \text{ar}_{\text{sdf}}(P, k)$ then $\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{sdf}}(Q, k)$ (for some $k \in \mathbb{Z}^*$) by making use of Theorem 3.3.

Theorem 3.3 *Principle of Mathematical Induction [12]. Let $S(n)$ denote an open mathematical statement (or set of such open statements) that involves one or more occurrences of the variable n , which represents a positive integer.*

a) If $S(0)$ is true; and

b) If whenever $S(k)$ is true (for some $k \in \mathbb{Z}^*$) then $S(k + 1)$ is true;

then $S(n)$ is true for all $n \in \mathbb{Z}^*$

Proof: The proof can be found in [12]. □

Implementation

By looking at the arrival times of token k a formula of the implementation can be defined. Because of the threshold N only the arrival time of the N 'th token, the $2N$ 'th token, etc are relevant. Another way to look at it is to see the N tokens as a bigger token which is a container of N small tokens. Now the small token k can be transformed into a big token l . See Equation 3.12 to calculate l from k .

$$k = \overbrace{0, 1, \dots, N-1}^{l=0}, \overbrace{N, N+1, \dots, 2N-1}^{l=1}, \overbrace{2N, 2N+1, \dots, 3N-1}^{l=2}, \dots$$

$$l = \left\lfloor \frac{k}{N} \right\rfloor \quad (3.12)$$

A regular expression is derived for the arrival times of token l in FIFO Q. The arrival times have a lower bound and an upper bound. When trying to proof $\text{ar}_{\text{impl}}(Q, l) \leq \text{ar}_{\text{SDF}}(Q, l)$ it is only interesting to look at the upper bound of $\text{ar}_{\text{impl}}(Q, l)$. Equation 3.13 and Equation 3.14 are the upper bounds of the token arrival times in FIFO Q.

$$\text{ar}_{\text{impl}}(Q, 0) \leq \text{ar}_{\text{impl}}(P, 0) + T_A + T_{A1} \quad (3.13)$$

$$\text{ar}_{\text{impl}}(Q, l) \leq \max \left[\text{ar}_{\text{impl}}(P, l) + T_A + T_{A1}, \text{ar}_{\text{impl}}(Q, l-1) + T_A \right] \quad l \geq 1 \quad (3.14)$$

SDF Model

The SDF model for the TDMA wheel with a threshold N is given in Figure 3.8. For this model also a regular expression can be derived. The regular expression can then be compared with the regular expression of the implementation. Because of the threshold, the same trick is used as with the implementation. Therefore the threshold of N tokens can be seen as a bigger token which is a

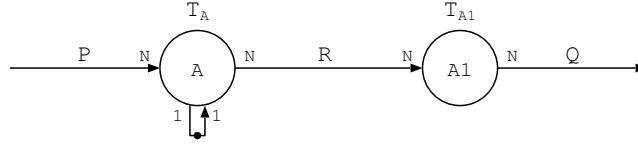


Figure 3.8: SDF model of the TDMA wheel with a threshold of N tokens

container of N small tokens. Equation 3.12 can be used to calculate l from k . The calculation of the regular expression can be seen below:

$$\text{ar}_{\text{sdf}}(R, l) = \max [\text{ar}_{\text{sdf}}(P, l), \text{ar}_{\text{sdf}}(R, l - 1)] + T_A$$

$$\text{ar}_{\text{sdf}}(Q, l) = \text{ar}_{\text{sdf}}(R, l) + T_{A1}$$

$$\text{ar}_{\text{sdf}}(Q, l) = \max [\text{ar}_{\text{sdf}}(P, l), \text{ar}_{\text{sdf}}(Q, l - 1) - T_{A1}] + T_A + T_{A1}$$

Now the regular expression can be rewritten in the same format as is used for the regular expression of the implementation, as shown in Equation 3.16. Equation 3.15 is a special expression for the first token $l = 0$. This is possible because the initial state of the SDF model is given.

$$\text{ar}_{\text{sdf}}(Q, 0) = \text{ar}_{\text{sdf}}(P, 0) + T_A + T_{A1} \quad (3.15)$$

$$\text{ar}_{\text{sdf}}(Q, l) = \max [\text{ar}_{\text{sdf}}(P, l) + T_A + T_{A1}, \text{ar}_{\text{sdf}}(Q, l - 1) + T_A] \quad l \geq 1 \quad (3.16)$$

Proof

The statement that has to be proven with Theorem 3.3 is:

$$S(l) : \text{ar}_{\text{impl}}(P, l) \leq \text{ar}_{\text{sdf}}(P, l) \Rightarrow \text{ar}_{\text{impl}}(Q, l) \leq \text{ar}_{\text{sdf}}(Q, l) \quad (3.17)$$

The first step a is called the *basis step*. Therefore statement $S(0)$ is examined:

$$\text{ar}_{\text{impl}}(Q, 0) \leq \text{ar}_{\text{sdf}}(Q, 0) \quad (3.18)$$

To verify Equation 3.18 the upper bound of Equation 3.13 is substituted for the start condition of the implementation and the right side of Equation 3.15 is substituted for the start condition of the SDF model:

$$\text{ar}_{\text{impl}}(P, 0) + T_A + T_{A1} \leq \text{ar}_{\text{sdf}}(P, 0) + T_A + T_{A1}$$

$$\text{ar}_{\text{impl}}(P, 0) \leq \text{ar}_{\text{sdf}}(P, 0) \quad (3.19)$$

Equation 3.19 was the condition of Equation 3.17, therefore $S(0)$ is true. Now step b of Theorem 3.3 will be examined, which is called the *inductive step*. Assuming that the result is true for $S(l)$ (for some $l \in \mathbb{Z}^*$) the *induction step* forces the truth of $S(l+1)$. The assumption of the truth of $S(l)$ is called the *induction hypothesis*. To establish the truth of $S(l+1)$ it has to be shown that:

$$\text{ar}_{\text{impl}}(Q, l+1) \leq \text{ar}_{\text{sdf}}(Q, l+1) \quad (3.20)$$

Now Equation 3.20 will be verified by using the equations of the implementation and the SDF model. For the implementation the upper bound of Equation 3.14 is taken and for the SDF model the right side of Equation 3.16 is used. After substituting $l+1$ and defining a and b for the implementation and c and d for the SDF model Equation 3.21 follows.

$$\max[a, b] \leq \max[c, d] \quad (3.21)$$

For a, b, c, d defined as:

$$a = \text{ar}_{\text{impl}}(P, l+1) + T_A + T_{A1}$$

$$b = \text{ar}_{\text{impl}}(Q, l) + T_A$$

$$c = \text{ar}_{\text{sdf}}(P, l+1) + T_A + T_{A1}$$

$$d = \text{ar}_{\text{sdf}}(Q, l) + T_A$$

Next it is important to show that Equation 3.21 is correct in all circumstances. To do that Lemma 3.1 is used.

Lemma 3.1 *The condition $\max[a, b] \leq \max[c, d]$ holds if $((c \geq a) \text{ and } (c \geq b))$ or if $((d \geq a) \text{ and } (d \geq b))$. A sufficient condition is that $c \geq a$ and that $d \geq b$.*

Proof: Lets assume $a \geq b$ then by the condition $c \geq a$ follows that the equation $\max[a, b] \leq \max[c, d]$ holds. By assuming $a < b$ the other condition $d \geq b$ can be used to proof that the equation $\max[a, b] \leq \max[c, d]$ holds. Therefore the conditions $c \geq a$ and $d \geq b$ are sufficient to proof the condition $\max[a, b] \leq \max[c, d]$. \square

Subsequently this lemma is used on Equation 3.21, first for $a \leq c$ and then for $b \leq d$.

$$a \leq c$$

$$\text{ar}_{\text{impl}}(P, l+1) + T_A + T_{A1} \leq \text{ar}_{\text{sdf}}(P, l+1) + T_A + T_{A1}$$

$$\text{ar}_{\text{impl}}(P, l+1) \leq \text{ar}_{\text{sdf}}(P, l+1) \quad (3.22)$$

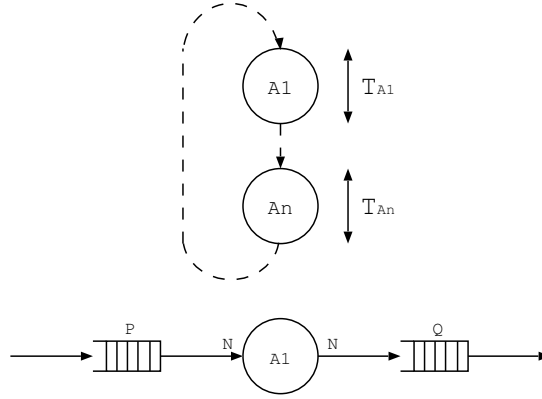


Figure 3.9: A implementation of RR with a threshold of N tokens.

Equation 3.22 was the condition of Equation 3.17, therefore the first part of the lemma is correct. Next the second part of the lemma will be looked at.

$$b \leq d$$

$$\text{ar}_{\text{impl}}(Q, l) + T_A \leq \text{ar}_{\text{sdf}}(Q, l) + T_A$$

$$\text{ar}_{\text{impl}}(Q, l) \leq \text{ar}_{\text{sdf}}(Q, l) \quad (3.23)$$

Equation 3.23 was assumed true by the *induction hypothesis*. Because Equation 3.22 and Equation 3.23 are true it follows that Equation 3.20 is true.

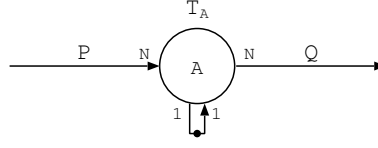
Consequently, by the principle of finite induction, $S(l)$ is true for all $l \in \mathbb{Z}^*$.

3.3.2 RR, N

In this subsection an SDF model is derived for RR arbitration with a threshold of N tokens. RR is a cyclic list where every actor is executed subsequently. If an actor can not execute it will return such that the next actor in the list can execute. An implementation of RR is shown in Figure 3.9. The worst-case time of one period is $T_A = T_{A1} + T_{An}$. The worst-case execution time of actor $A1$ is T_{A1} . The first step is to express the characteristics of the implementation in an equation. The next step is to express the SDF model in an equation. When both equations are defined it is possible to proof that if $\text{ar}_{\text{impl}}(P, k) \leq \text{ar}_{\text{sdf}}(P, k)$ then $\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{sdf}}(Q, k)$ (for some $k \in \mathbb{Z}^*$) by making use of Theorem 3.3.

Implementation

By looking at the arrival times of token k a formula of the implementation can be defined. Because of the threshold N only the arrival time of the N' th token, the $2N'$ th token, etc are relevant. Another way to look at it is to see the N tokens as a bigger token which is a container of N small tokens. Now the small token k can be transformed into a big token l . See Equation 3.12 to calculate l from k .

Figure 3.10: SDF model of RR with a threshold of N tokens.

A regular expression is derived for the arrival times of token l in FIFO Q. The arrival times have a lower bound and an upper bound. When trying to proof $\text{ar}_{\text{impl}}(Q, l) \leq \text{ar}_{\text{SDF}}(Q, l)$ it is only interesting to look at the upper bound of $\text{ar}_{\text{impl}}(Q, l)$. Equation 3.24 and Equation 3.25 are the upper bounds of the token arrival times in FIFO Q.

$$\text{ar}_{\text{impl}}(Q, 0) \leq \text{ar}_{\text{impl}}(P, 0) + T_A \quad (3.24)$$

$$\text{ar}_{\text{impl}}(Q, l) \leq \max \left[\text{ar}_{\text{impl}}(P, l) + T_A, \text{ar}_{\text{impl}}(Q, l - 1) + T_A \right] \quad l \geq 1 \quad (3.25)$$

SDF Model

The SDF model for RR with a threshold N is given in Figure 3.10. For this model also a regular expression can be derived. The regular expression can then be compared with the regular expression of the implementation. Because of the threshold, the same trick is used as with the implementation. Therefore the threshold of N tokens can be seen as a bigger token which is a container of N small tokens. Equation 3.12 can be used to calculate l from k . The calculation of the regular expression can be seen below:

$$\text{ar}_{\text{sdf}}(Q, l) = \max \left[\text{ar}_{\text{sdf}}(P, l), \text{ar}_{\text{sdf}}(Q, l - 1) \right] + T_A$$

Now the regular expression can be rewritten in the same format as is used in the implementation, as shown in Equation 3.27. Equation 3.26 is a special expression for the first token $l = 0$. This is possible because the initial state of the SDF model is given.

$$\text{ar}_{\text{sdf}}(Q, 0) = \text{ar}_{\text{sdf}}(P, 0) + T_A \quad (3.26)$$

$$\text{ar}_{\text{sdf}}(Q, l) = \max \left[\text{ar}_{\text{sdf}}(P, l) + T_A, \text{ar}_{\text{sdf}}(Q, l - 1) + T_A \right] \quad l \geq 1 \quad (3.27)$$

Proof

The statement that has to be proven with Theorem 3.3 is:

$$S(l) : ar_{impl}(P, l) \leq ar_{sdf}(P, l) \Rightarrow ar_{impl}(Q, l) \leq ar_{sdf}(Q, l) \quad (3.28)$$

The first step a is called the *basis step*. Therefore statement $S(0)$ is examined:

$$ar_{impl}(Q, 0) \leq ar_{sdf}(Q, 0) \quad (3.29)$$

To verify Equation 3.29 the upper bound of Equation 3.24 is substituted for the start condition of the implementation and the right side of Equation 3.26 is substituted for the start condition of the SDF model:

$$ar_{impl}(P, 0) + T_A \leq ar_{sdf}(P, 0) + T_A$$

$$ar_{impl}(P, 0) \leq ar_{sdf}(P, 0) \quad (3.30)$$

Equation 3.30 was the condition of Equation 3.28, therefore $S(0)$ is true. Now step b of Theorem 3.3 will be examined, which is called the *inductive step*. Assuming that the result is true for $S(l)$ (for some $l \in \mathbb{Z}^*$) the *induction step* forces the truth of $S(l+1)$. The assumption of the truth of $S(l)$ is called the *induction hypothesis*. To establish the truth of $S(l+1)$ it has to be shown that:

$$ar_{impl}(Q, l+1) \leq ar_{sdf}(Q, l+1) \quad (3.31)$$

Now Equation 3.31 will be verified by using the equations of the implementation and the SDF model. For the implementation the upper bound of Equation 3.25 is taken and for the SDF model the right side of Equation 3.27 is used. After substituting $l+1$ and defining a and b for the implementation and c and d for the SDF model Equation 3.32 follows.

$$\max[a, b] \leq \max[c, d] \quad (3.32)$$

For a, b, c, d defined as:

$$a = ar_{impl}(P, l+1) + T_A$$

$$b = ar_{impl}(Q, l) + T_A$$

$$c = ar_{sdf}(P, l+1) + T_A$$

$$d = ar_{sdf}(Q, l) + T_A$$

Next it is important to show that Equation 3.32 is correct in all circumstances. To do that Lemma 3.1 is used on Equation 3.32 first for $a \leq c$ and then for $b \leq d$.

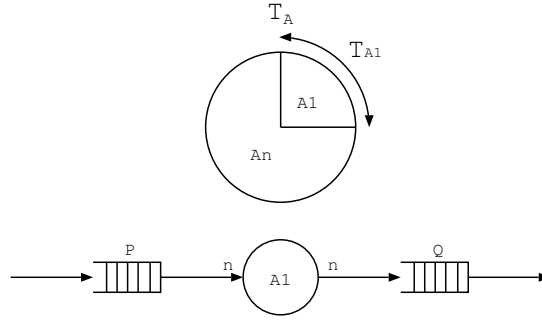


Figure 3.11: A implementation of the TDMA wheel with n the number of tokens $1 \leq n \leq N$.

$$a \leq c$$

$$\text{ar}_{\text{impl}}(P, l + 1) + T_A \leq \text{ar}_{\text{sdf}}(P, l + 1) + T_A$$

$$\text{ar}_{\text{impl}}(P, l + 1) \leq \text{ar}_{\text{sdf}}(P, l + 1) \quad (3.33)$$

Equation 3.33 was the condition of Equation 3.28, therefore the first part of the lemma is correct. Next the second part of the lemma will be looked at.

$$b \leq d$$

$$\text{ar}_{\text{impl}}(Q, l) + T_A \leq \text{ar}_{\text{sdf}}(Q, l) + T_A$$

$$\text{ar}_{\text{impl}}(Q, l) \leq \text{ar}_{\text{sdf}}(Q, l) \quad (3.34)$$

Equation 3.34 was assumed true by the *induction hypothesis*. Because Equation 3.33 and Equation 3.34 are true it follows that Equation 3.31 is true.

Consequently, by the principle of finite induction, $S(l)$ is true for all $l \in \mathbb{Z}^*$.

3.3.3 TDMA, $\leq N$

In this subsection an SDF model is derived for TDMA arbitration without a threshold but a maximum of N tokens per period. An implementation of the TDMA wheel with n the number of consumed and produced tokens ($1 \leq n \leq N$) is shown in Figure 3.11. The first step is to express the characteristics of the implementation in an equation. The next step is to express the SDF model in an equation. When both equations are defined it is possible to proof that if $\text{ar}_{\text{impl}}(P, k) \leq \text{ar}_{\text{sdf}}(P, k)$ then $\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{sdf}}(Q, k)$ (for some $k \in \mathbb{Z}^*$) by making use of Theorem 3.3.

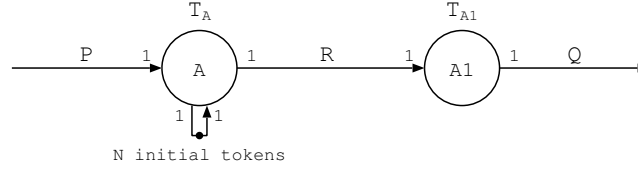


Figure 3.12: SDF model of the TDMA wheel with no threshold but a maximum of N tokens.

Implementation

By looking at the arrival times of token k a formula of the implementation can be defined. A regular expression is derived for the arrival times of token k in FIFO Q. The arrival times have a lower bound and an upper bound. When trying to proof $\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{SDF}}(Q, k)$ it is only interesting to look at the upper bound of $\text{ar}_{\text{impl}}(Q, k)$. Equation 3.35 and Equation 3.36 are the upper bounds of the token arrival times in FIFO Q.

$$\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{impl}}(P, k) + T_A + T_{A1} \quad 0 \leq k < N \quad (3.35)$$

$$\text{ar}_{\text{impl}}(Q, k) \leq \max \left[\text{ar}_{\text{impl}}(P, k) + T_A + T_{A1}, \text{ar}_{\text{impl}}(Q, k - N) + T_A \right] \quad k \geq N \quad (3.36)$$

SDF Model

The SDF model for the TDMA wheel without a threshold but a maximum of N tokens is given in Figure 3.12. For this model also a regular expression can be derived. The regular expression can then be compared with the regular expression of the implementation. The calculation of the regular expression can be seen below:

$$\text{ar}_{\text{sdf}}(R, k) = \max \left[\text{ar}_{\text{sdf}}(P, k), \text{ar}_{\text{sdf}}(R, k - N) \right] + T_A$$

$$\text{ar}_{\text{sdf}}(Q, k) = \text{ar}_{\text{sdf}}(R, k) + T_{A1}$$

$$\text{ar}_{\text{sdf}}(Q, k) = \max \left[\text{ar}_{\text{sdf}}(P, k), \text{ar}_{\text{sdf}}(Q, k - N) - T_{A1} \right] + T_A + T_{A1}$$

Now the regular expression can be rewritten in the same format as is used in the implementation, as shown in Equation 3.38. Equation 3.37 is a special expression for the first N tokens $1 \leq k < N$. This is possible because the initial state of the SDF model is given.

$$\text{ar}_{\text{sdf}}(Q, k) = \text{ar}_{\text{sdf}}(P, k) + T_A + T_{A1} \quad 0 \leq k < N \quad (3.37)$$

$$\text{ar}_{\text{sdf}}(Q, k) = \max \left[\text{ar}_{\text{sdf}}(P, k) + T_A + T_{A1}, \text{ar}_{\text{sdf}}(Q, k - N) + T_A \right] \quad k \geq N \quad (3.38)$$

Proof

The statement that has to be proven with Theorem 3.3 is:

$$S(k) : \text{ar}_{\text{impl}}(P, k) \leq \text{ar}_{\text{sdf}}(P, k) \Rightarrow \text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{sdf}}(Q, k) \quad (3.39)$$

The first step *a* is called the *basis step*. Therefore statement $S(k)$ is examined for the first N tokens $0 \leq k < N$:

$$\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{sdf}}(Q, k) \quad 0 \leq k < N \quad (3.40)$$

To verify Equation 3.40 the upper bound of Equation 3.35 is substituted for the start condition of the implementation and the right side of Equation 3.37 is substituted for the start condition of the SDF model:

$$\text{ar}_{\text{impl}}(P, k) + T_A + T_{A1} \leq \text{ar}_{\text{sdf}}(P, k) + T_A + T_{A1}$$

$$\text{ar}_{\text{impl}}(P, k) \leq \text{ar}_{\text{sdf}}(P, k) \quad 0 \leq k < N \quad (3.41)$$

Equation 3.41 was the condition of Equation 3.39, therefore $S(k)$ is true for the first N tokens $0 \leq k < N$. Now step *b* of Theorem 3.3 will be examined, which is called the *inductive step*. Assuming that the result is true for $S(k)$ the *induction step* forces the truth of $S(k+1)$ for some $k \geq N$. The assumption of the truth of $S(k)$ is called the *induction hypothesis*. To establish the truth of $S(k+1)$ it has to be shown that:

$$\text{ar}_{\text{impl}}(Q, k+1) \leq \text{ar}_{\text{sdf}}(Q, k+1) \quad (3.42)$$

Now Equation 3.42 will be verified by using the equations of the implementation and the SDF model. For the implementation the upper bound of Equation 3.36 is taken and for the SDF model the right side of Equation 3.38 is used. After substituting $k+1$ and defining a and b for the implementation and c and d for the SDF model Equation 3.43 follows.

$$\max[a, b] \leq \max[c, d] \quad (3.43)$$

For a, b, c, d defined as:

$$a = \text{ar}_{\text{impl}}(P, k+1) + T_A + T_{A1}$$

$$b = \text{ar}_{\text{impl}}(Q, k-N+1) + T_A$$

$$c = \text{ar}_{\text{sdf}}(P, k+1) + T_A + T_{A1}$$

$$d = \text{ar}_{\text{sdf}}(Q, k-N+1) + T_A$$

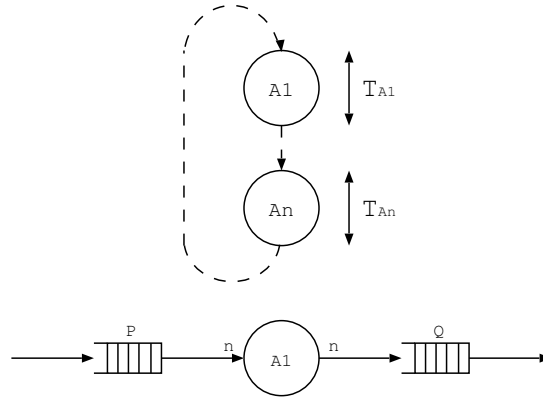


Figure 3.13: A implementation of RR with n the number of tokens $1 \leq n \leq N$.

Next it is important to show that Equation 3.43 is correct in all circumstances. To do that Lemma 3.1 is used on Equation 3.43 first for $a \leq c$ and then for $b \leq d$.

$$a \leq c$$

$$\text{ar}_{\text{impl}}(P, k + 1) + T_A + T_{A1} \leq \text{ar}_{\text{sdf}}(P, k + 1) + T_A + T_{A1}$$

$$\text{ar}_{\text{impl}}(P, k + 1) \leq \text{ar}_{\text{sdf}}(P, k + 1) \quad (3.44)$$

Equation 3.44 was the condition of Equation 3.39, therefore the first part of the lemma is correct. Next the second part of the lemma will be looked at.

$$b \leq d$$

$$\text{ar}_{\text{impl}}(Q, k - N + 1) + T_A \leq \text{ar}_{\text{sdf}}(Q, k - N + 1) + T_A$$

$$\text{ar}_{\text{impl}}(Q, k - N + 1) \leq \text{ar}_{\text{sdf}}(Q, k - N + 1) \quad (3.45)$$

Equation 3.45 was assumed true by the *induction hypothesis*. Because Equation 3.44 and Equation 3.45 are true it follows that Equation 3.42 is true.

Consequently, by the principle of finite induction, $S(l)$ is true for all $l \in \mathbb{Z}^*$.

3.3.4 RR, $\leq N$

In this subsection an SDF model is derived for RR arbitration without a threshold but a maximum of N tokens per period. An implementation of RR with n the number of consumed and produced tokens ($1 \leq n \leq N$) is shown in Figure 3.13. The first step is to express the characteristics of the implementation in an equation. The next step is to express the SDF model in an equation. When both equations are defined it is possible to proof that if $\text{ar}_{\text{impl}}(P, k) \leq \text{ar}_{\text{sdf}}(P, k)$ then $\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{sdf}}(Q, k)$ (for some $k \in \mathbb{Z}^*$) by making use of Theorem 3.3.

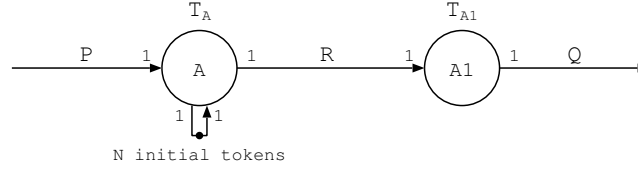


Figure 3.14: SDF model of RR with no threshold but a maximum of N tokens.

Implementation

By looking at the arrival times of token k a formula of the implementation can be defined. A regular expression is derived for the arrival times of token k in FIFO Q. The arrival times have a lower bound and an upper bound. When trying to proof $\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{SDF}}(Q, k)$ it is only interesting to look at the upper bound of $\text{ar}_{\text{impl}}(Q, k)$. Equation 3.46 and Equation 3.47 are the upper bounds of the token arrival times in FIFO Q.

$$\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{impl}}(P, k) + T_A + T_{A1} \quad 0 \leq k < N \quad (3.46)$$

$$\text{ar}_{\text{impl}}(Q, k) \leq \max \left[\text{ar}_{\text{impl}}(P, k) + T_A + T_{A1}, \text{ar}_{\text{impl}}(Q, k - N) + T_A \right] \quad k \geq N \quad (3.47)$$

SDF Model

The SDF model for RR without a threshold but a maximum of N tokens is given in Figure 3.14. For this model also a regular expression can be derived. The regular expression can then be compared with the regular expression of the implementation. The calculation of the regular expression can be seen below:

$$\text{ar}_{\text{sdf}}(R, k) = \max \left[\text{ar}_{\text{sdf}}(P, k), \text{ar}_{\text{sdf}}(R, k - N) \right] + T_A$$

$$\text{ar}_{\text{sdf}}(Q, k) = \text{ar}_{\text{sdf}}(R, k) + T_{A1}$$

$$\text{ar}_{\text{sdf}}(Q, k) = \max \left[\text{ar}_{\text{sdf}}(P, k), \text{ar}_{\text{sdf}}(Q, k - N) - T_{A1} \right] + T_A + T_{A1}$$

Now the regular expression can be rewritten in the same format as is used in the implementation, as shown in Equation 3.49. Equation 3.48 is a special expression for the first N tokens $1 \leq k < N$. This is possible because the initial state of the SDF model is given.

$$\text{ar}_{\text{sdf}}(Q, k) = \text{ar}_{\text{sdf}}(P, k) + T_A + T_{A1} \quad 0 \leq k < N \quad (3.48)$$

$$\text{ar}_{\text{sdf}}(Q, k) = \max \left[\text{ar}_{\text{sdf}}(P, k) + T_A + T_{A1}, \text{ar}_{\text{sdf}}(Q, k - N) + T_A \right] \quad k \geq N \quad (3.49)$$

Proof

The statement that has to be proven with Theorem 3.3 is:

$$S(k) : \text{ar}_{\text{impl}}(P, k) \leq \text{ar}_{\text{sdf}}(P, k) \Rightarrow \text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{sdf}}(Q, k) \quad (3.50)$$

The first step *a* is called the *basis step*. Therefore statement $S(k)$ is examined for the first N tokens $0 \leq k < N$:

$$\text{ar}_{\text{impl}}(Q, k) \leq \text{ar}_{\text{sdf}}(Q, k) \quad 0 \leq k < N \quad (3.51)$$

To verify Equation 3.51 the upper bound of Equation 3.46 is substituted for the start condition of the implementation and the right side of Equation 3.48 is substituted for the start condition of the SDF model:

$$\text{ar}_{\text{impl}}(P, k) + T_A + T_{A1} \leq \text{ar}_{\text{sdf}}(P, k) + T_A + T_{A1}$$

$$\text{ar}_{\text{impl}}(P, k) \leq \text{ar}_{\text{sdf}}(P, k) \quad 0 \leq k < N \quad (3.52)$$

Equation 3.52 was the condition of Equation 3.50, therefore $S(k)$ is true for the first N tokens $0 \leq k < N$. Now step *b* of Theorem 3.3 will be examined, which is called the *inductive step*. Assuming that the result is true for $S(k)$ the *induction step* forces the truth of $S(k + 1)$ for some $k \geq N$. The assumption of the truth of $S(k)$ is called the *induction hypothesis*. To establish the truth of $S(k + 1)$ it has to be shown that:

$$\text{ar}_{\text{impl}}(Q, k + 1) \leq \text{ar}_{\text{sdf}}(Q, k + 1) \quad (3.53)$$

Now Equation 3.53 will be verified by using the equations of the implementation and the SDF model. For the implementation the upper bound of Equation 3.47 is taken and for the SDF model the right side of Equation 3.49 is used. After substituting $k + 1$ and defining a and b for the implementation and c and d for the SDF model Equation 3.54 follows.

$$\max [a, b] \leq \max [c, d] \quad (3.54)$$

For a, b, c, d defined as:

$$a = \text{ar}_{\text{impl}}(P, k + 1) + T_A + T_{A1}$$

$$b = \text{ar}_{\text{impl}}(Q, k - N + 1) + T_A$$

$$c = \text{ar}_{\text{sdf}}(P, k + 1) + T_A + T_{A1}$$

$$d = \text{ar}_{\text{sdf}}(Q, k - N + 1) + T_A$$

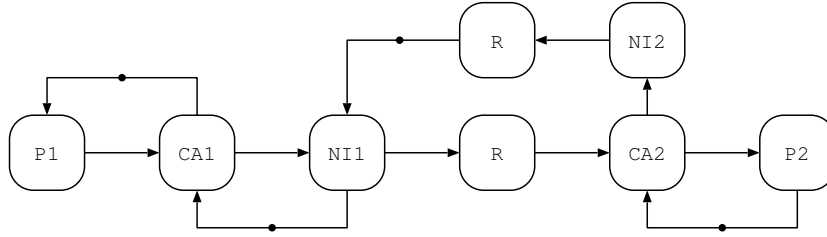


Figure 3.15: Abstract view of the communication channel with credit paths for fifo capacity.

Next it is important to show that Equation 3.54 is correct in all circumstances. To do that Lemma 3.1 is used on Equation 3.54 first for $a \leq c$ and then for $b \leq d$.

$$a \leq c$$

$$\text{ar}_{\text{impl}}(P, k + 1) + T_A + T_{A1} \leq \text{ar}_{\text{sdf}}(P, k + 1) + T_A + T_{A1}$$

$$\text{ar}_{\text{impl}}(P, k + 1) \leq \text{ar}_{\text{sdf}}(P, k + 1) \quad (3.55)$$

Equation 3.55 was the condition of Equation 3.50, therefore the first part of the lemma is correct. Next the second part of the lemma will be looked at.

$$b \leq d$$

$$\text{ar}_{\text{impl}}(Q, k - N + 1) + T_A \leq \text{ar}_{\text{sdf}}(Q, k - N + 1) + T_A$$

$$\text{ar}_{\text{impl}}(Q, k - N + 1) \leq \text{ar}_{\text{sdf}}(Q, k - N + 1) \quad (3.56)$$

Equation 3.56 was assumed true by the *induction hypothesis*. Because Equation 3.55 and Equation 3.56 are true it follows that Equation 3.53 is true.

Consequently, by the principle of finite induction, $S(l)$ is true for all $l \in \mathbb{Z}^*$.

3.4 Model of the Communication Channel

Section 3.3 explained some building blocks for arbitration and scheduling techniques. These building blocks will be used to model the complete communication channel between two actors which are executed on different processors. The communication channel was already introduced in Section 2.5. In Figure 3.15 the communication channel of Figure 2.5 is depicted in a slightly different way. FIFOs are represented with a edge with on this edge a number of tokens equal to the capacity of the FIFO in words. The routers are represented by a node R and the register S is replaced by S tokens, which represents the space in *Fifo NI2*.

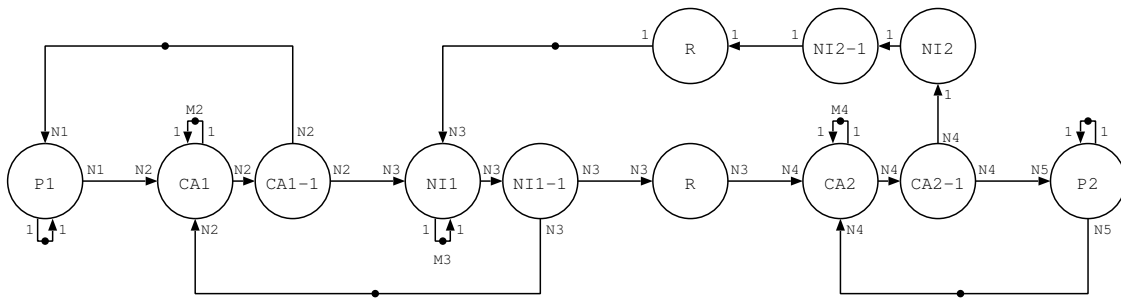


Figure 3.16: SDF model of the communication channel.

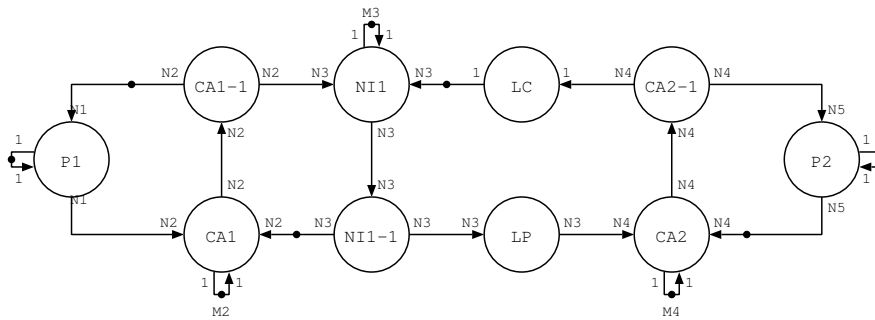


Figure 3.17: Simplified SDF model of a communication channel.

To come to a conservative SDF model the blocks have to be replaced by a SDF model. The router block R is just an actor with a worst-case execution time related to the worst-case delay of all the routers on the path in the network. In this SDF model the granularity of one token is one word. The rest of the blocks can be replaced by one of the basic blocks from Section 3.3. The actors that are executed on the processor use the model of Section 3.3.2, RR with a threshold of N tokens. The rest of the blocks use two parameters. One parameter for the threshold (N) and one parameter for the number of tokens on the self-edge (M). With these parameters one can adapt the model to any of the models introduced in Section 3.3. The result of substituting all these SDF models into Figure 3.15 is Figure 3.16. Finally the model can be simplified by merging the chains of actors which do not have self-edge, as is done for the latency in the credit path of the network ($T_{LC} = T_R + T_{NI2-1} + T_{NI2}$). For simplicity the router in the packet path of the network is replaced by latency packet ($T_{LP} = T_R$). The SDF model in which the actors are merged is shown in Figure 3.17.

Chapter 4

Transaction Level Simulator

Building the level 2 simulator was one of the goals of the graduation project. Another name for the level 2 simulator is the transaction level simulator. This chapter starts with an overview of the functions in the simulator. In Section 4.2 an example illustrates how the simulator works. Eventually some information is given about the structure in Section 4.3. This section goes also further into the implementation of the communication channel and the implementation of the steady state detection in the simulator.

4.1 Functions overview

In practice this simulator is also called the Hijdra Application Programmer's Interface (HAPI) Simulator. The HAPI simulator is built on top of the YAPI simulator [7]. With the HAPI simulator it is possible to simulate SDF graphs, where the YAPI simulator is intended for the simulation of Kahn process networks [13]. Information about programming in YAPI simulator can be found in the application programmer's guide [7]. This chapter will focus on the additions made to the YAPI simulator. The functions in the HAPI simulator are explained in the following subsections.

4.1.1 FIFO versus communication channel

For the connection between two processes there is a choice between a FIFO and a communication channel. The communication channel models the communication through the CA's and the network. The FIFO is just a normal FIFO that also existed in the original YAPI simulator. The addition of this FIFO compared with YAPI is that some functions are included, like *Peek()* and *Peek_Wait()*.

It is possible to declare a FIFO in three different ways as is shown in Figure 4.1. In the first case (1) a default FIFO is declared. This is a FIFO with a minimum size of 1 and a maximum size of 2K words (2048 words). The maximum default size of 2K words was a choice of the designer, it

```
Fifo(const Id& name_fifo); (1)
Fifo(const Id& name_fifo, unsigned int min); (2)
Fifo(const Id& name_fifo, unsigned int min, unsigned int max); (3)
```

Figure 4.1: Declaration example of a FIFO.

```

#include "hfifomappings.h"
table_mapping_fifos_to_channels.Add(
    char* name_fifo,
    unsigned int capacity_mem_write,
    unsigned int capacity_ni_write,
    unsigned int capacity_ni_read,
    unsigned int capacity_mem_read,
    unsigned int N_ca_write,
    unsigned int N_ni,
    unsigned int N_ca_read,
    unsigned int M_ca_write,
    unsigned int M_ni,
    unsigned int M_ca_read,
    double T_ca_write,
    double T_ca_writel,
    double T_ni,
    double T_nil,
    double T_ca_read,
    double T_ca_read1,
    double T_lp,
    double T_lc);

```

Figure 4.2: Example of mapping a FIFO to a communication channel.

could have bin every size bigger than zero. If the FIFO capacity exceeds the current size it will grow to the new size in the case the new size is smaller the maximum size. If the new size exceeds the maximum size then the FIFO size will be equal to the maximum size. In the second declaration (2) a FIFO with a minimum size is declared. With the third option (3) the minimum and maximum size can be defined. Given this option it is possible to declare a FIFO with a fixed capacity by selecting the same value for the minimum and maximum size.

In the case a communication channel is used then a FIFO is mapped to a channel. In this case the FIFO behavior changes into the behavior of a communication channel. This mapping is done by the function *table_mapping_fifos_to_channels.Add()*, as shown in Figure 4.2. The mapping parameters are explained in Chapter 3. These parameters are the same as the parameters in the model of the communication channel in Figure 3.17. The difference is that the actors P1 and P2 that run on the processors are modeled by an HAPI process. The behavior of the other actors in the communication channel are simulated within the connection between the HAPI processes. The parameters in the function *table_mapping_fifos_to_channels.Add()* have a unit of words for the capacity parameters and a unit of nano seconds for the timing parameters.

4.1.2 The write function

```

template<class T>
void write(OutPort<T>& s, const T& t);

template<class T>
void write(OutPort<T>& s, const T* p, unsigned int n);

```

Figure 4.3: Syntax of the function write.

The function *write()* copies data contained in a given variable to a given output port. The type of the output port must match with the type of the variable. This means that either both types are identical, or the variable is an array of the type of the output port. In the latter case one must provide a third argument to the write function which indicates the number of elements to be written, as shown in Figure 4.3. The write function is a blocking function. It means that if the function cannot write it will wait until the data can be written. After the function finishes writing the data it will return.

4.1.3 The read function

```
template<class T>
    void read(InPort<T>& s, const T& t);

template<class T>
    void read(InPort<T>& s, const T* p, unsigned int n);
```

Figure 4.4: Syntax of the function read.

The function *read()* consumes data from a given input port and stores this data in a given variable. The type of the input port must match with the type of the variable. This means that either both types are identical, or the variable is an array of the type of the input port. In the latter case one must provide a third argument to the read function which indicates the number of elements to be read, as shown in Figure 4.4. The read function is a blocking function. It means that if the function cannot read it will wait until the data can be read. After the function finishes reading the data it will return.

4.1.4 The peek function

```
template<class T>
    bool peek(InPort<T>& s);

template<class T>
    bool peek(OutPort<T>& s);

template<class T>
    bool peek(InPort<T>& s, unsigned int n);

template<class T>
    bool peek(OutPort<T>& s, unsigned int n);
```

Figure 4.5: Syntax of the peek function.

The function *peek()* checks if data or space is available at a given port. It returns a boolean which is true when the data or space is available. When reading or writing an array a second argument must be provided which indicates the number of elements of the array, as shown in Figure 4.5. The peek function is a non-blocking function. That means that the function returns always. The boolean is only true when the point in time of calling the *peek()* function is later then the time data or space is available. So if the arrival time of packets or credits is equal to the point in time the function

peek() is called it returns false. The *peek()* function can be used for connection that consist of control information. In that case an actor can check for control information.

4.1.5 The peek wait function

```
template<class T>
    bool peek_wait(InPort<T>& s);

template<class T>
    bool peek_wait(OutPort<T>& s);

template<class T>
    bool peek_wait(InPort<T>& s, unsigned int n);

template<class T>
    bool peek_wait(OutPort<T>& s, unsigned int n);
```

Figure 4.6: Syntax of the peek wait function.

The function *peek_wait()* checks if data or space is available at a given port and returns when it is available. When reading or writing an array one must provide a second argument which indicates the number of elements of the array, as shown in Figure 4.6. The difference with the function *peek()* is that it waits when data or space is not available. For that reason the peek wait function is a blocking function. The *peek_wait()* function is used for simulating SDF behavior.

4.1.6 The select function

The *select* function has the same behavior as in the YAPI simulator. The syntax can be found in the application programmers guide [7]. A select on a communication channel exhibits the same behavior as a select on a FIFO.

4.1.7 The steady state function

```
#include "state.h"
bool steadyState(const ProcessNetwork& pn);           (1)
bool steadyState(const ProcessNetwork& pn, unsigned int n); (2)
```

Figure 4.7: Syntax of the steady state function.

The function *steadyState()* is introduced to detect when the execution of the SDF enters the periodic regime. It can be interesting to know, because it can be used as a stop criteria for the simulator. The reason is that simulating one period is sufficient because all successive periods are identical. It would make no sense to simulate more than one period. The theory behind detection of the steady state in an SDF graph is explained in Section 3.1.3.

In the simulator there are two ways to detect the steady state as shown in Figure 4.7. The *steadyState()* function is called by one actor of the SDF graph. The first function (1) has to be used when there is no information about the SDF graph. When the number of executions in one period of the

actor, which is calling the *steadyState()* function, is known the second function (2) can be used. The second parameter in the steady state function is the number of executions in one period of the specific actor.

As mentioned before only one actor in the SDF graph may call the *steadyState()* function. The best way to do so is by the actor with the lowest repetition factor [20]. In this case the function *steadyState()* is called a minimal number of times before the steady state is detected, which results in a shorter simulation time.

The functions return *true* when the SDF graph is in a steady state. When the simulation reaches the steady state then the *steadyState()* function (1) will print the *Cycle time* and $N \cdot \text{Repetitionfactor}$. The definitions of *cycle time* and the number N can be found in Section 3.1.3. In case of *steadyState()* function (2) is used then only the *Cycle time* is printed because the $n = N \cdot \text{Repetitionfactor}$ was given. With the *Cycle time*, the $N \cdot \text{Repetitionfactor}$ and the number of tokens that are produced/consumed it is possible to derive the MCM.

When *steadyState()* function (1) detects a steady state it will simulate another period to determine the value of n . When the value of n is known *steadyState()* function (2) can be used. The advantage of *steadyState()* function (2) in comparison with *steadyState()* function (1) is the simulation time.

The HAPI simulator is compiled with the program SystemC [8]. The *steadyState()* function has to be called by a process. This creates the problem that the steady state is depending of the execution order of the processes in SystemC. This is not acceptable. The steady state function should be called when all the actors are finished executing. This is after step 6 of the simulation cycle in Appendix B. This can be forced by waiting a delta time before calling the *steadyState()* function. When doing so it has to be sure that it does not effect the MCM. Therefore it can be necessary to compensate the delta time as will be explained in Section 4.2. This solution only works when the delta is small enough. The delta must be smaller than the granularity of the waiting times in the system.

Note: At this point in time the steady state function is only working with communication channels and not with FIFOs. The reason is that FIFOs are used for control connections and communication channels are used for data connections. Although this is not the most efficient way, one is sure that it is always conservative. In future work it would be interesting to distinguish control and data in FIFOs and communication channels. In this case it is for example possible to take the data FIFOs into account during detecting of the steady state.

4.2 Hard-Realtime Producer-Consumer Example

To make the global structure of an application clear a hard real-time producer-consumer example will be illustrated. This application is one of the basics and is a *Hello World* type of applications of HAPI. Figure 4.8 shows an abstract overview of the application.

When running the hard real-time producer-consumer example on the HAPI simulator it is simulating running on the Hidra Multiprocessor platform. It is important to know more about the environment. For example, the mapping of the actors to the processors, the worst-case execution time of the ac-

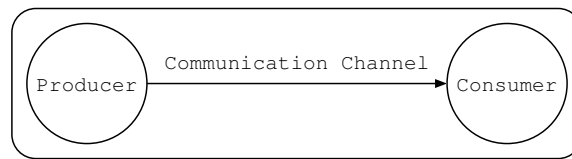


Figure 4.8: Producer-Consumer example.

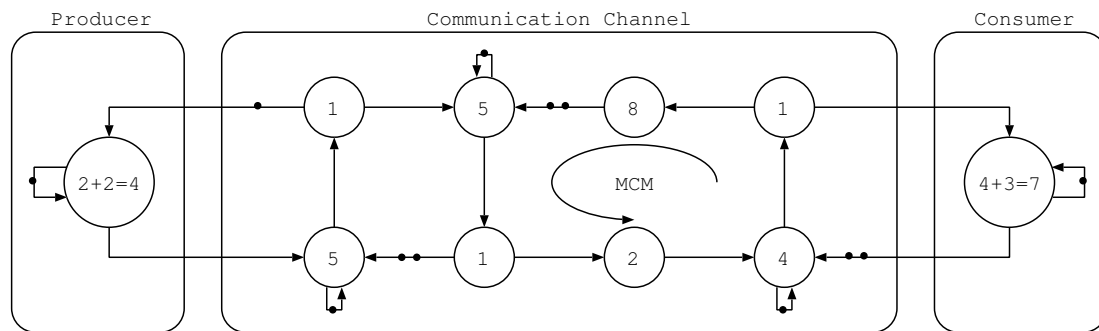


Figure 4.9: SDF Model of the Producer-Consumer example.

tors, the cycle times of the processors, etc. This is important information because of the setting of the parameters in the SDF model, as explained in Chapter 3. Let's assume that these parameters are known. Then it is possible to model the producer-consumer application like it is running on the Hijdra platform. For this example parameters are assumed and put into the SDF model. The result can be seen in Figure 4.9.

Now this application (SDF model) will be build into our simulator. Starting with the Main Program.

4.2.1 The Main Program

To run the HAPI application one must first create a process network and next call the function `start()` to execute it. Before starting execution, a pointer to the process network should be created. This pointer is needed to be able to call the steady state function from one of the processes. That is also the reason that the variable `pointer_pc` is a global variable which is declared outside the main function.

4.2.2 The Process Network

The Producer-Consumer process network is represented by the class PC. The declaration of this class can be found in the file `pc.h` (shown in Figure 4.11), and the implementation can be found in the file `pc.cc` (shown in Figure 4.12). The difference between YAPI and HAPI is that the standard FIFO can be mapped to a Hijdra communication channel. This is done by the `table_mapping_fifos_to_channels.Add()` function. The parameters of this function are shown in Figure 4.2 in Section 4.1.1.

```
#include "pc.h"
#include "rte.h"
#include "network.h"

ProcessNetwork* pointer_pc;

int main()
{
    // Create the process network
    PC pc( id("pc") );

    // Create pointer to process network
    pointer_pc=&pc;

    // start the process network
    start(pc);
}
```

Figure 4.10: The Main Program (main.cc).

```
#include "network.h"
#include "fifo.h"
#include "producer.h"
#include "consumer.h"

class PC : public ProcessNetwork
{
public:
    PC(const Id& n);
    const char* type() const;

private:
    Fifo<int> fifo;

    Producer prod;
    Consumer cons;
};
```

Figure 4.11: Process Network Declaration (pc.h).

```
#include "pc.h"
#include "hfifomappings.h"

PC::PC(const Id& n) :
    ProcessNetwork(n),
    fifo( id("fifo") ),
    prod( id("prod"), fifo),
    cons( id("cons"), fifo)
{
    table_mapping_fifos_to_channels.Add("fifo",
        1, 2, 2, 2,
        1, 1, 1,
        1, 1, 1,
        5, 1, 5, 1, 4, 1,
        2, 8);
}

const char* PC::type() const
{
    return "PC";
}
```

Figure 4.12: Process Network Implementation (pc.cc).

```
#include "process.h"
#include "port.h"
#include "network.h"

extern ProcessNetwork* pointer_pc;

class Producer : public Process
{
public:
    Producer(const Id& n, Out<int>& o);
    const char* type() const;
    const double state() const;
    void main();

private:
    OutPort<int> out;
    double self_edge;
};
```

Figure 4.13: Producer Process Declaration (producer.h).

```
#include "systemc.h"
#include "producer.h"
#include <iostream>

Producer::Producer(const Id& n, Out<int>& o) :
    Process(n),
    out( id("out"), o)
{
    self_edge=0;
}

const char* Producer::type() const
{
    return "Producer";
}

const double Producer::state() const
{
    return sc_simulation_time()-self_edge;
}

void Producer::main()
{
    cout << "Producer started" << endl;

    sc_time time_scheduling(2000,SC_PS);
    sc_time time_process(2000,SC_PS);

    int i=0;
    while ( true )
    {
        peek_wait(out);
        wait(time_scheduling);
        wait(time_process);
        write(out, i);
        self_edge=sc_simulation_time();
        i++;
    }

    cout << type() << " " << fullName() << ": "
        <<"  Writing="<<i
        <<"  Time="<<sc_simulation_time()<<endl;
}
```

Figure 4.14: Producer Process Implementation (producer.cc).

```
#include "process.h"
#include "port.h"

extern ProcessNetwork* pointer_pc;

class Consumer : public Process
{
public:
    Consumer(const Id& n, In<int>& i);
    const char* type() const;
    const double state() const;
    void main();

private:
    InPort<int> in;
    double self_edge;
};
```

Figure 4.15: Consumer Process Declaration (consumer.h).

4.2.3 The Producer

The producer process is represented by the class *Producer*. The class is declared in the included file *producer.h* (shown in Figure 4.13) and its implementation can be found in the file *producer.cc* (shown in Figure 4.14). Every process in HAPI must contain the function *state()*. This function is obligatory because it is declared as a *virtual* function in the interface. It is obligatory because the state of every actor is needed to determine when the steady state is reached. In SDF terms it would be the waiting time of the token on the self edge of the actor. Indirectly every actor in HAPI has a self edge. The reason is that in HAPI every actor has to wait until the last execution was finished before it can execute again.

The producer process has to simulate the behavior of an SDF actor. An SDF actor has to wait until all the inputs and outputs are ready, before it can execute. That is the reason of the *peek_wait()* function. When the inputs and outputs are ready it waits for the worst-case execution time as derived in Section 3.3.2. After that it will read the tokens from all the inputs and write tokens to all outputs. At last it also updates the time-stamp on the self edge.

4.2.4 The Consumer

The consumer process is represented by the class *Consumer*. The class is declared in the included file *consumer.h* (shown in Figure 4.15) and its implementation can be found in the file *consumer.cc* (shown in Figure 4.16). The differences between the consumer process and the producer process is that it has an input port instead of an output port, and that data is read from this port instead of being written.

The *steadyState()* function should be in one of the processes. Here it is placed in the consumer process. The *steadyState()* function has to be called every time the actor fires. It can be used as a stop criteria by putting the function in the while statement of the process. As explained in Section 4.1.7 the scheduling of SystemC has to be forced to make sure that all the actors are executed, before calling the *steadyState()* function. This is done by waiting a small delta before calling the *steadyState()* function. Because the MCM of the SDF graph should be correct, we use

```
#include "systemc.h"
#include "state.h"
#include "consumer.h"
#include <assert.h>
#include <iostream>

Consumer::Consumer(const Id& n, In<int>& i) :
    Process(n),
    in( id("in"), i)
{
    self_edge=0;
}

const char* Consumer::type() const
{
    return "Consumer";
}

const double Consumer::state() const
{
    return sc_simulation_time()-self_edge;
}

void Consumer::main()
{
    cout << "Consumer started" << endl;

    sc_time time_delta(1,SC_PS);
    sc_time time_scheduling(4000,SC_PS);
    sc_time time_scheduling_min_delta(3999,SC_PS);
    sc_time time_process(3000,SC_PS);

    wait(time_delta);

    int temp;
    int j=0;
    while ( !steadyState(*pointer_pc) )
    {
        if( peek(in) )
        {
            wait(time_scheduling_min_delta);
            wait(time_process);
        }
        else
        {
            peek_wait(in);
            wait(time_scheduling);
            wait(time_process);
        }
        read(in, temp);
        assert(temp==j);
        self_edge=sc_simulation_time();
        j++;

        wait(time_delta);
    }

    cout << type() << " " << fullName() << ": "
        <<"  Reading="<<j
        <<"  Time="<<sc_simulation_time()<<endl;
}
```

Figure 4.16: Consumer Process Implementation (consumer.cc).

```
Producer started
Consumer started
+-----+
| Cycle time      =      21  |
| N * Repetitionfactor =      2  |
+-----+
Consumer pc.cons:   Reading=21   Time=240.001
```

Figure 4.17: The output of the example.

two different scheduling times. The original scheduling time and a compensated scheduling time (scheduling time minus delta). When the data is ready the compensated scheduling time is used and in the other case the original scheduling time is used, as shown in Figure 4.16. It is very important that the delta is small enough. The delta should be smaller than the granularity of the worst-case execution times of all the actors. In mathematical terms it should be smaller than the *greatest common divider* of all the worst-case execution times.

4.2.5 Running the example

After the code is compiled it can be executed. During execution the output of Figure 4.17 is produced. Examination of the output reveals that both processes are started. The rectangular block is printed by the *steadyState()* function. Inside are the *cycle time* and $N \cdot \text{Repetitionfactor}$. The cycle time is the time of one period, so the time between two identical states. The second number is the number of times the actor (who calls the *steadyState()* function) fires between the identical states. This number is equal to N times the Repetitionfactor of the specific actor. More information about the N and the steady state is given in Section 3.1.3 and in [2]. After the rectangular block (as shown in Figure 4.17) there is some information printed by the consumer process. There is no information from the producer because this process never finishes due to the infinite while loop.

4.3 Implementation

The structure of the HAPI simulator is quite complex. That is why this section gives some information about the structure. Section 4.3.1 explains the implementation of the communication channel in detail. At last Section 4.3.2 provides some info about the implementation of the *steadyState()* function. More information can of course be retrieved from the SystemC code of the HAPI simulator. It can also be useful to look at the YAPI Architecture [6].

The software packages are shown in Figure 4.18. These packages are groups of classes with a similar functionality. In the source code the groups have their own directory. The name of the directory is the same name as the corresponding package. The *API* classes are the interfaces between the application and the HAPI implementation. *Impl*, *Base* and *RTE* packages are quite similar with the YAPI implementation. The real difference between the YAPI and HAPI implementation is the *hrte* package. Appendix C gives a more detailed overview of the relation between the classes.

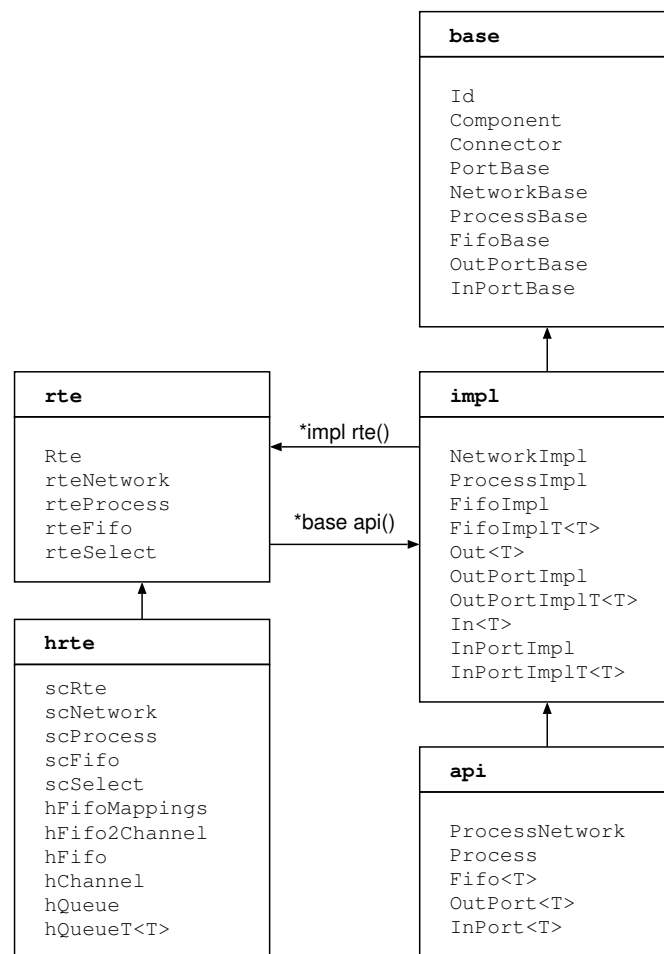


Figure 4.18: The software packages.

```

void hChannel::read(unsigned int* pdata, unsigned int nitems)
{
    assert(nitems <= capacity_mem_read);
    while( not enough packets in mem read queue )
    {
        if( not enough packets in ni read to fire ca read )
        {
            if( not enough packets in ni write to fire ni )
            {
                if( not enough packets in mem write to fire ca write )
                {
                    Wait for credit to be spend;
                }
            }
        }
        if( CA write can fire ) FireCaWrite();
        if( NI can fire ) FireNi();
        if( CA read can fire ) FireCaRead();
    }
    Get packets from mem read queue;
    Wait for packet to arrive at mem read;
    Write credits to the mem read queue;
    notify( credit_send_event );
};

```

Figure 4.19: Pseudo code of the read function in the hChannel implementation.

4.3.1 Communication Channel

When a FIFO is mapped to a communication channel it should have the behavior of a communication channel without adding any processes. Therefore the read function and the write function have to be rewritten in such a way that the FIFO has the behavior of the channel. The advantage of adding no processes is the simplicity and the simulation speed.

The implementation of the communication channel is in the files *hchannel.h* and *hchannel.cc*. The behavior of the communication channel is specified by the SDF model of the communication channel. The SDF model is explained in Section 3.4 and an example of the SDF model is given in Figure 4.9. In Figure 4.9 it is made explicit which SDF actors belong to the communication channel and which actors are modelled by the processes.

The structure of the functions *write()*, *read()*, *peek()* and *peek_wait()* are more or less the same. The main difference is that the reading functions are focussing on packets and the writing functions are focussing on credits. Because the structure of the functions are more or less the same only the function *read* is explained.

An overview of the structure, the parameters and the variables that are used in the implementation is given in Appendix D. Figure 4.19 shows the structure of the read function. The read function wants to take *nitems* of packets out of the memory read queue. If there are not enough packets it will go into the while loop. There it is executing actors (*FireCaWrite*, *FireNi*, *FireCaRead*) until there are enough packets in the memory read queue. If there are not enough packets in the channel to execute any actor then the read function waits until the write function sends a credit. Very important is that none of the functions *FireCaWrite*, *FireNi* and *FireCaRead* are blocking. This means that no

```
void hChannel::FireCaRead(void)
{
    assert( packets inside ni read >= N_ca_read );
    assert( credits inside mem read >= N_ca_read );

    Get packets from NI read queue;
    Get credits out of mem read queue;
    Get token from self edge;

    Fire CA read by calculating the time stamps;
    Put token to self edge;

    Fire CA read 1 by calculating the time stamps;
    Put packets to the mem read queue;
    Put credits to the ni read queue;
}
```

Figure 4.20: Pseudo code of the function `FireCaRead` in the `hChannel` implementation.

`wait` statement may be used. Because if one of the functions `FireCaWrite`, `FireNi` and `FireCaRead` is blocking then deadlock can occur when the producing and consuming process try to execute the same function. That is the reason that every packet and every credit has its own time stamp, so every arrival time can be calculated without a `wait` statement.

The pseudo code of executing the CA at the reading side is shown in Figure 4.20. This corresponds with executing the `FireCaRead` in Appendix D. First the assert statement checks whether there are enough packets and credits to execute. The assert statements are only introduced for debugging purposes and to take sure that no memory segmentation fault occurs. After getting the packets and credits the CA actor will consume one token from its self edge. After consuming all the tokens the arrival times of all the output tokens are calculated and the time stamps are updated. When the output tokens are updated they will be written to the output queues.

The advantage of using time stamps in the communication channel in stead of processes for every actor is that it will not result in events to be handled by the SystemC scheduler. This results in shorter simulation times. The time stamps of the tokens will only be calculated and updated when it is necessary. That is in case of `read()`, `write()`, `peek()` and `peek_wait()` actions of the process. This can be done because the structure of the communication channel, in our case, is always the same.

4.3.2 Steady State

The main program of the steady state is inside the `scNetwork` implementation. The `scNetwork` implementation is one of the class of the `hrte` package. From there it can call the state function from all the processes and all the communication channels. The function will save the state as one long string. The idea is that it can compare the states by comparing the strings. If there are two identical strings the states are also identical. In the current implementation only communication channels and actors are called for their state. Therefore normal FIFOs are not taken into account for the steady state. This is done because the normal FIFOs are used for control information and not for data streams. Communication channels must in the current implementation be used for data streams. The requirement of the level 2 simulator is that it is conservative. In the case that every data connection is a communication channel it is sure that the SDF graph is conservative.

```

void hChannel::state(String& o)
{
    while( CA Read can fire || NI can fire || CA Write can fire )
    {
        if( CA Read can fire ) FireCaRead();
        if( NI can fire ) FireNi();
        if( CA Write can fire ) FireCaWrite();
    }

    Print the waiting times of the credits from Memory write;
    Print the waiting times of the credits from NI write;
    Print the waiting times of the credits from NI read;
    Print the waiting times of the credits from Memory read;

    Print the waiting times of the packets from Memory read;
    Print the waiting times of the packets from NI read;
    Print the waiting times of the packets from NI write;
    Print the waiting times of the packets from Memory write;

    Print the waiting times of the self edge from CA write;
    Print the waiting times of the self edge from NI;
    Print the waiting times of the self edge from CA read;
}

```

Figure 4.21: Pseudo code of the function State in the hChannel implementation.

To calculate the state of a communication channel, as defined in Section 3.1.3, at a certain point in time t is not straight forward. The reason is that the tokens inside the communication channel have a time stamp. The time stamp of a token in the communication channel is defined as the arrival time of the token at the a particular data edge in the SDF graph. Therefore the place of the token in the communication channel does not have to be equal with the place of the token at a certain time t . It would be to expensive to calculate the exact place at a certain point in time for all the tokens in the communication channel. The nice thing is that the state of a token is defined as the waiting time of an actor on a particular data edge at a certain time t . Therefore the state is relative to t which makes it possibly to get negative waiting times of the tokens on a particular data edge in the SDF graph. When storing negative waiting times of tokens on a data edge one has to make sure that it is done always. The strategy that was implemented is to execute all the actors inside the communication channel until none of the actors can execute anymore. Doing it in this way makes sure that the next time the tokens will be at the same place in the communication channel. If the tokens are on the same place in the communication channel and the waiting times of the tokens are equal then the state of the channel is the same. Figure 4.21 shows the structure of the state function. The structure of the state function in a process was already shown by Figure 4.14.

After calculating the state the new state has to be compared with the previous states. The comparison between the new state and the previous states depends on which of the two *steadyState()* functions is applied.

If the *steadyState(pointer_pn)* is applied then the new state will be compared with all the previous states. All the states are stored in a hash table [5]. Searching in a hash table for identical strings is more efficient then searching in a linear list. When two identical states are found it is clear that the simulation is in a steady state. Now the simulator runs another period to calculate the number

```
Initial values:
  in steady state = false;
  fired = 0;
  last state = empty;
  cycle time = 0;

bool scNetwork::steadyState()
{
  if( in steady state )
  {
    fired++;
    Get the new state;
    if( last state == new state )
    {
      Print cycle time;
      Print number of firing between two states;
      fired = 0;
      return true;
    }
  }
  else
  {
    Get the new state;
    if( new state already exist in data structure )
    {
      Get the old state that is identical with the new one;
      cycle time = the time between those two states;
      in steady state = true;
      last state = cur state;
    }
    Save the new state to the data structure;
  }
  return false;
}
```

Figure 4.22: Pseudo code of the function steadyState(pointer_pn).

```
Initial values:
  fired = 0;
  last state = empty;
  cycle time = 0;
  last fire = 0;

bool scNetwork::steadyState(unsigned int n)
{
  fired++;
  if( fired == n )
  {
    fired=0;
    Get the new state;
    if( new state == last state )
    {
      cycle time = simulation time - last fire;
      Print cycle time;
      last state = new state;
      last fire = simulation time;
      return true;
    }
    else
    {
      last state = new state;
      last fire = simulation time;
      return false;
    }
  }
  return false;
}
```

Figure 4.23: Pseudo code of the function `steadyState(pointer_pn, n)`.

of executions of the actor, that called the `steadyState()` functions, before the same state is reached again as shown in Figure 4.22. This number is useful because it can be used to call the function `steadyState(pointer_pn, n)`. The computational complexity of comparing every state with all the previous states is polynomial with the power of two.

If the `steadyState(pointer_pn, n)` is applied then the advantage is that it is not necessary to store all the previous states. The reason is that the number of times the actor executes is known before the same state is reached. When this number is known only one of the n states has to be compared. For example comparing state 0 with state n and comparing state n with state $2 \cdot n$. Only two states are compared at a time therefore only one state has to be stored. This comparison can be performed by an algorithm with a linear computational complexity.

Chapter 5

Communication Assist Design

The CA has been introduced in Chapter 2 of this thesis. The implementation of the CA will be described in this chapter. Section 5.1 describes some limitations of the proposed CA that is implemented. Subsequently the interface of the CA will be assumed in Section 5.2. Section 5.3 explains which arbitration the CA has to perform. Next the CA can be configured which will be explained in Section 5.4. Finally in Section 5.5 the design of the CA is given in more detail.

5.1 Assignment definition

The original graduation assignment was to implement a CA at the cycle true level and to describe it in synthetical VHDL. Because the specification of the CA was not clear it seemed at that time a good idea to start with building a level 2 simulator model of the communication channels which includes the CAs. This took so much time that there was hardly any time left to implement the CA. That is the reason that there are some simplifications in the implementation of the CA. The simplifications are listed below:

- The channel will use registers that initiate the connection. They have to be set-up at the allocation of the channel. Changing of the registers is not included.
- Actors can not share connections! This can only be done if there is a credit path between the CAs.
- Only 2 connections are used in the examples sake of clarity.

5.2 Interface

For now not much effort is put in defining the interfaces of the CA. Therefore some basic interfaces are assumed. See Appendix F for the components that are communicating with the CA.

- The processor will have a *data bus in*, *data bus out*, a *address bus* and a *read/write* signal. Furthermore it has *request* and *grant* handshake signals for communicating with the bus arbiter.
- The memory is a block with a *data bus in*, *data bus out*, a *address bus* and a *read/write* signal.

- The NI works with the Device Transaction Level (DTL) protocol [18]. In this thesis only the connection identifier (connid) and message bus (MSG) are used. When a connection identifier is send, data can be read from the message bus or data can be send to the message bus. The NI has also special signals to send the FIFO status. These signals will notify whether a particular FIFO is full when sending data to the network or empty when reading data from the network.

5.3 Arbitration and scheduling performed by the Communication Assist

The CA is responsible for the arbitration of the switch in the tile. The reason that there is a switch is that both the processor and the CA want to have access to the data memory. Another task of the CA is to handle multiple simultaneous data streams. In other words the CA has to decide for which stream the data is transferred. These decisions are made by a scheduler in the CA.

5.3.1 Arbitration of the Local-bus inside a tile

On the shared bus there are two masters and two slaves. The masters are the processor and the CA. The memory and the NI are the slaves. The CA will control the switch in such a way that both the processor and the CA have an equal amount of time to access the data memory.

In this implementation background memory arbitration [21] is used for the arbitration of the local-bus. In general there is a service cycle of N clock cycles and M cycles are reserved for the processor. It is guaranteed that the processor gets M and the CA gets $N - M$ cycles per period N . In this implementation a service cycle of 2 clock cycles ($N = 2$) was taken and one clock cycle was reserved for the processor ($M = 1$) and one clock cycle for the CA ($N - M = 1$). This is done for the simplicity. But it could be made configurable with registers.

5.3.2 Scheduling between Network Connections

The scheduling between the different connections is done by RR. There is no minimal threshold for transporting data. The maximum number of words that can be transported at once is specified by a register.

Before the choice was made for this scheduling technique TDMA was compared with RR, on both cases with and without a minimum threshold. The maximum throughput for a TDMA wheel is fixed with $\#tokens/T_{TDMA}$. For RR the throughput is at least $\#tokens/T_{RR}$. Here the assumption is made that the *worst case execution time* of an actor in RR is the same as in TDMA.

It is getting more interesting if the latency between RR and TDMA is compared. For this comparison some simple scheduling models were built. These models work with a *stochastic execution time* for the actors. The idea is that the execution time of the actors is not fixed but is according to a distribution.

The models can be used in a simple test application, as shown in Figure 5.1. The generator generates tokens randomly and puts them in FIFO A. The scheduling model will consume the tokens from

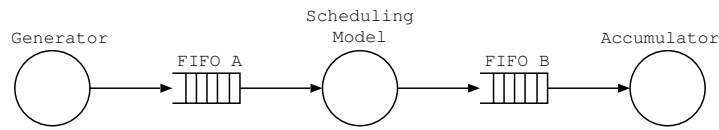


Figure 5.1: Test application for comparing scheduling techniques.

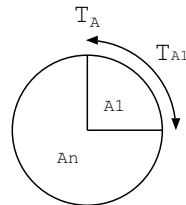


Figure 5.2: TDMA wheel with a cycle time T_A and a actor time T_{A1} .

FIFO A and puts them in FIFO B. The accumulator consumes the tokens from FIFO B and analyzes the latency of the tokens. The latency is defined by the time the token arrives in FIFO B minus the time it arrived in FIFO A ($ar(B, k) - ar(A, k)$). The accumulator analysis all the latencies of all the tokens and puts the latencies into a graph. The graph of all the latencies will give a good idea of the latency distribution of the specific scheduling technique. The FIFO capacities are large enough to assume that they are unbounded.

The scheduling techniques that are compared are:

- TDMA with a threshold of N words
- RR with a threshold of N words
- TDMA with no minimum threshold, but a maximum of N words
- RR with no minimum threshold, but a maximum of N words

The pseudo code for the scheduling models are shown in Appendix E.

TDMA

The TDMA scheduling makes use of a time wheel, as shown in Figure 5.2. In this comparison the time wheel and the arrival times of the tokens are assumed to be independent. Another assumption is that the throughput of the generator is lower than the throughput of the TDMA scheduling. With these assumptions it is possible to predict the distribution of the latency.

The total latency for transport tokens from FIFO A to FIFO B is a concatenation of two latencies. The first latency is the waiting time before actor $A1$ can start firing. This time is depending on the time of one time wheel rotation T_A . Because the arrival time of a token is independent of the time wheel, the first latency is an homogeneous distribution between $(0, T_A]$. The second latency is the time it takes for the actor to transfer the token from FIFO A to FIFO B. In this example a homogeneous distribution between $[0, T_{A1}]$ is used. The concatenation of the two latencies will give the total latency distribution of the TDMA scheduling, as shown in Figure 5.3.

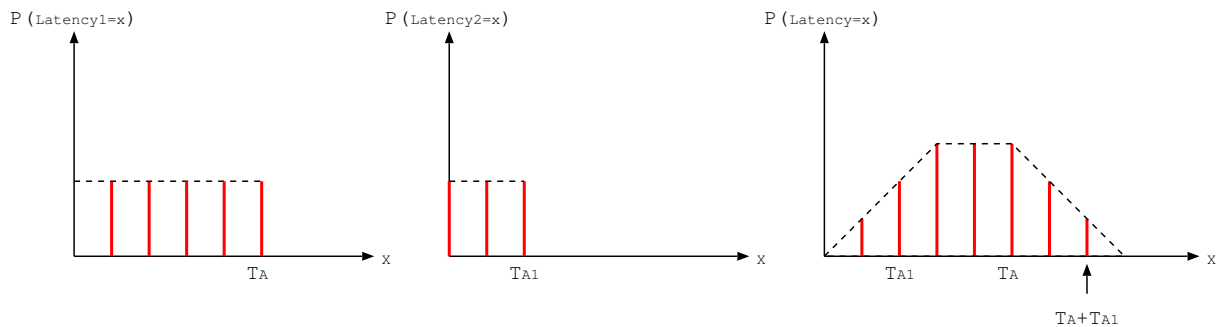


Figure 5.3: The distribution of Latency 1, Latency 2 and the total Latency in case of TDMA.

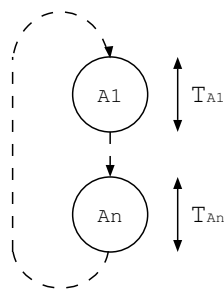


Figure 5.4: Round Robin list with a actor A1 and other actors An.

Round Robin

The next step is to look at a RR list, as shown in Figure 5.4. Also here the assumption is made that the RR scheduling is independent of the arrival times of tokens. The other assumption is that the throughput of the generator is lower than the throughput of the RR scheduling. With these assumptions it is possible to predict the distribution of the latency.

The total latency for transport tokens from FIFO A to FIFO B is again a concatenation of two latencies. The first latency is the waiting time before actor A1 can start firing. This time is depending on the execution time of actor An. But the execution time is not fixed. It is a homogeneous distribution between $[0, T_{An}]$ what is called $T_{schedule}$. Therefore the first latency is a distribution between $(0, T_{schedule}]$. The second latency is the time it takes for the actor to transfer the token from FIFO A to FIFO B. In this example a homogeneous distribution is used between $[0, T_{A1}]$. The concatenation of the two latencies will give the total latency distribution of the RR scheduling. Figure 5.5 shows two cases for the distribution of the total latency. In the first case that $T_{An} > T_{A1}$ and the second case that $T_{An} < T_{A1}$.

Comparison

The outcome of the test application is predicted therefore it is time to compare the prediction with the output of the test application. The setup of the first experiment is TDMA and RR that send one token at a time. The times that are chosen are $T_{A1} = 6$ and $T_{An} = 10$. The average time between two tokens of the generator is $T_{gen} = 50$. For the graph in Figure 5.6 one million tokens are used to get a reliable graph of the latency distribution. When looking at the graph it is clear

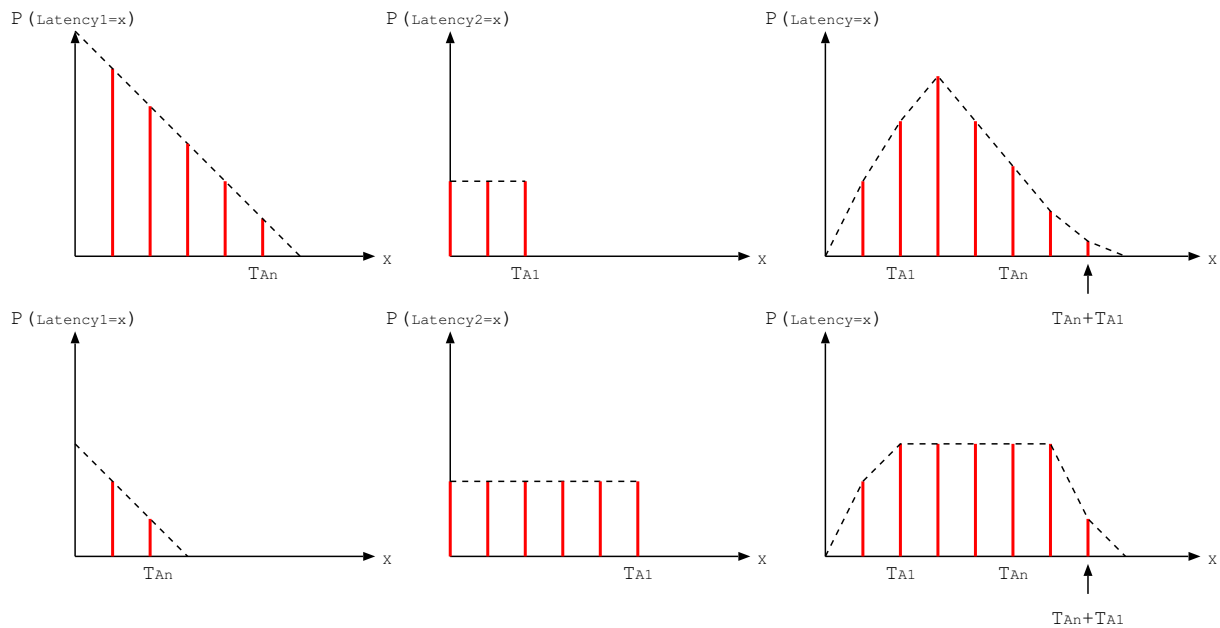


Figure 5.5: The distribution of Latency 1, Latency 2 and the total Latency in case of RR. The first example is the case that $T_{An} > T_{A1}$ and second is that $T_{An} < T_{A1}$.

that the typically latency when RR is applied is lower then the latency of TDMA. The reason is that TDMA has to wait for the end of the time slot in the TDMA wheel before it can go to the next actor.

The next experiment is the comparison between a fixed threshold and no threshold but a maximum of N words that can be transferred. For this experiment the times have not been changed ($T_{A1} = 6$ and $T_{An} = 10$). One RR scheduling uses a threshold of 2 words and the other RR scheduling uses no threshold but a maximum of 2 words. The average time between two tokens of the generator is $T_{gen} = 25$. Figure 5.7 shows the graph of this experiment. From this graph it can be concluded that the option which result in a lower average latency is RR with no threshold and a maximum of N words transferred per period.

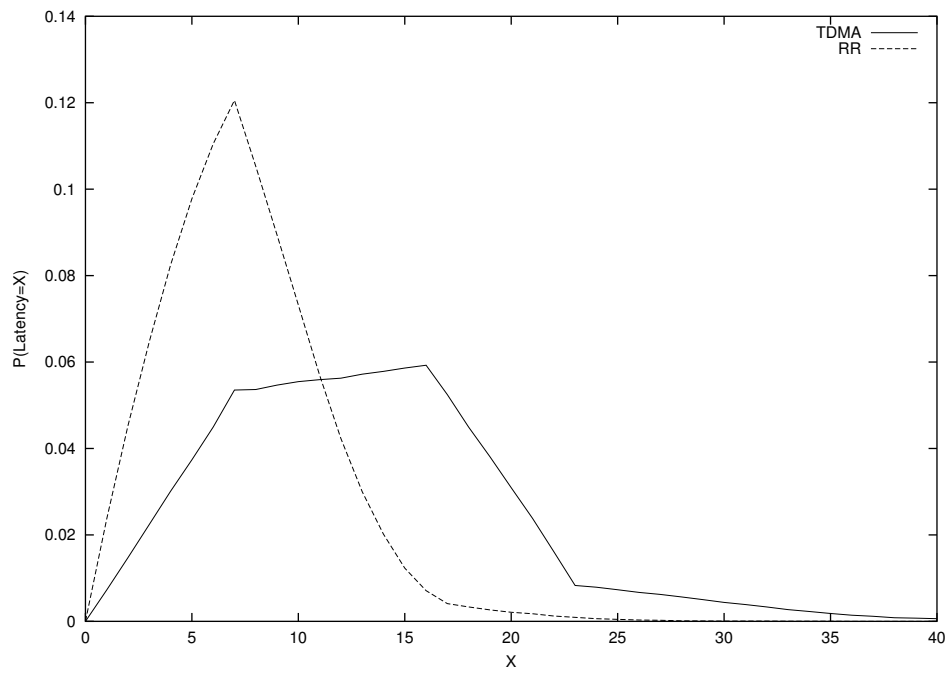


Figure 5.6: Latency distribution of TDMA and RR.

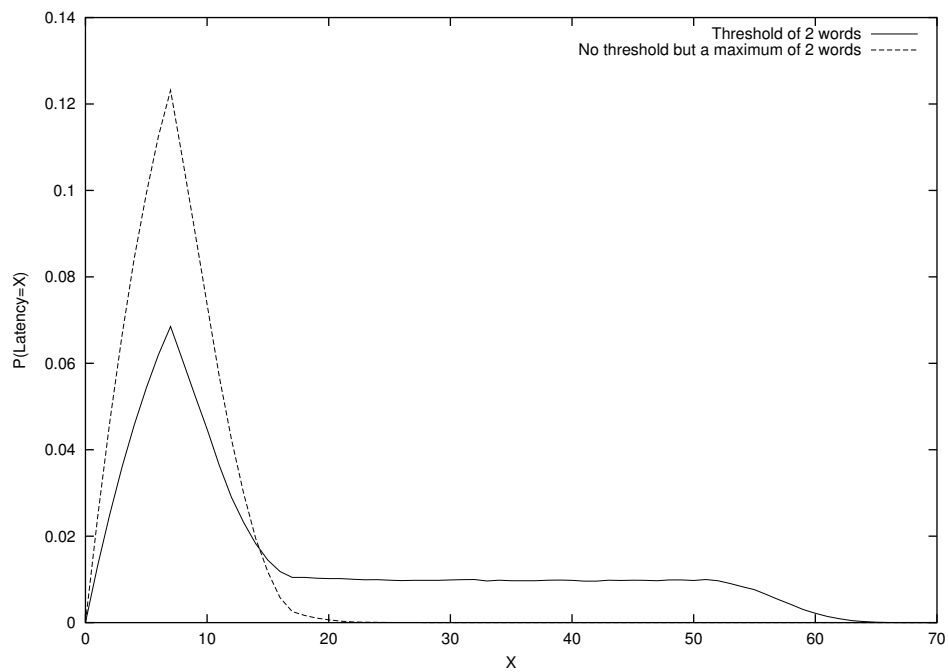


Figure 5.7: Latency distribution of RR with threshold and no threshold but a maximum number of words.

5.4 Configuration

The configuration of the CA can be done by setting registers as can be seen in Appendix F. In this implementation the programming of the registers is not included. This is done for the simplicity. The number of connections in the CA is the same as the number of FIFOs in the network interface. After design time the number is fixed and not reconfigurable anymore.

The registers that are configuring the CA are:

M_i	Maximum numbers of words to be transported at once.
<i>direc</i>	The direction of the channel.
<i>size</i>	The size of the FIFO.
<i>start</i>	The start address of the FIFO.
<i>stop</i>	The stop address of the FIFO.
<i>ID</i>	The connection ID of the network interface.

The *direc* register is necessary to define the direction of the channel. If *direc* = 1 than the data is going from the NI to the data memory and from the data memory to the processor. When *direc* = 0 the processor is writing to the data memory and the CA is transferring the data from the data Memory to the NI.

5.5 Implementation

The implementation of the CA is split up in more functional blocks. See Appendix F for the overview of the architecture. In this section the implementation of all the blocks is explained in more detail.

5.5.1 FIFO Status

By looking at the counter registers (*read* and *write*) the block *FIFO status* can see if the FIFO is full or empty. The block *FIFO status* sends also the status of the FIFO to the processor. Depending on the *direc* variable the status is the amount of data or the amount of space in the FIFO.

The control of the FIFO is done by the C-HEAP protocol [9]. There are two counters, a *read* counter and a *write* counter. The writing process of the FIFO is updating the *write* counter and the reading process is updating the *read* counter. The advantage of C-HEAP is that there are no semaphores needed. The Most Significant Bit (MSB) of the counters is the *wrap* bit. When a counter reaches the end of the FIFO, it will go back to its start position and the *wrap* bit is toggled. See Appendix G for some examples. The pseudo code of the block *FIFO status* is shown in Figure 5.8.

5.5.2 Fire

The *fire* block looks if it can transport data from one FIFO to another. The *fire* block does that by looking at the state of the FIFOs. The *direc* variable defines if the data is going from the NI to the Memory or from the Memory to the NI. The pseudo code of the block *fire* is shown in Figure 5.9.

```
FifoStatus()
{
    empty=false;
    full=false;

    // Look if the FIFO is Full or Empty
    if(readc==writec)
    {
        if(read_wrap==write_wrap) empty=true;
        else full=true;
    }

    if(direc==1)
    {
        // Look for the data that is available
        if(read_wrap==write_wrap) status=writec-readc;
        else status=size-readc+writec;
    }
    else
    {
        // Look for the space that is available
        if(read_wrap!=write_wrap) status=readc-writec;
        else status=size-writec+readc;
    }
}
```

Figure 5.8: Pseudo code for the block FIFO Status.

```
Fire()
{
    Req=false;

    if(direc==1)
    {
        // Transport data from NI to Memory
        if(!Empty_NI && !Full_Mem) Req=true;
    }
    else
    {
        // Transport data from Memory to NI
        if(!Empty_Mem && !Full_NI) Req=true;
        else Req=false;
    }
}
```

Figure 5.9: Pseudo code for the block Fire.

```

INC()
{
    Out_counter=In_counter+1;

    if(Out_counter==stop)
    {
        Out_counter=start;
        if(In_wrap==1) Out_wrap=0;
        else Out_wrap=1;
    }
}

```

Figure 5.10: Pseudo code for the block Increment.

```

Load()
{
    Load_readc=false;
    Load_writec=false;

    if(direc==1)
    {
        // Data from NI to Memory to Processor
        if(S_CA) Load_writec=true;
        if(S_P) Load_readc=true;
    }
    else
    {
        // Data from Processor to Memory to NI
        if(S_CA) Load_readc=true;
        if(S_P) Load_writec=true;
    }
}

```

Figure 5.11: Pseudo code for the block Load.

5.5.3 Increment

The block *increment* will take the counter value and increment it with one word. When counter reaches the stop value, it will go back to the value of the start counter and toggle the *wrap* bit. The pseudo code of the block *increment* is shown in Figure 5.10.

5.5.4 Load

The block *load* is responsible for loading the new counter value into the *read* and *write* counter. The block should load the counter if someone writes to or reads from a FIFO in the data memory. The block is also responsible for controlling the multiplexer such that the correct address is put on the address-bus. The pseudo code of the block *load* is shown in Figure 5.11.

5.5.5 Round Robin

This state machine implements the RR scheduling between the connections. The list will be fixed and is from connection 1 to N . Every connection i can transfer data with a maximum of M_i times. The state machine takes also into account that the processor wants to read from or write to the

Request_P, Req_1, Req_2, Zero / Grant_P, S1, S2, Load, Decrement, Select

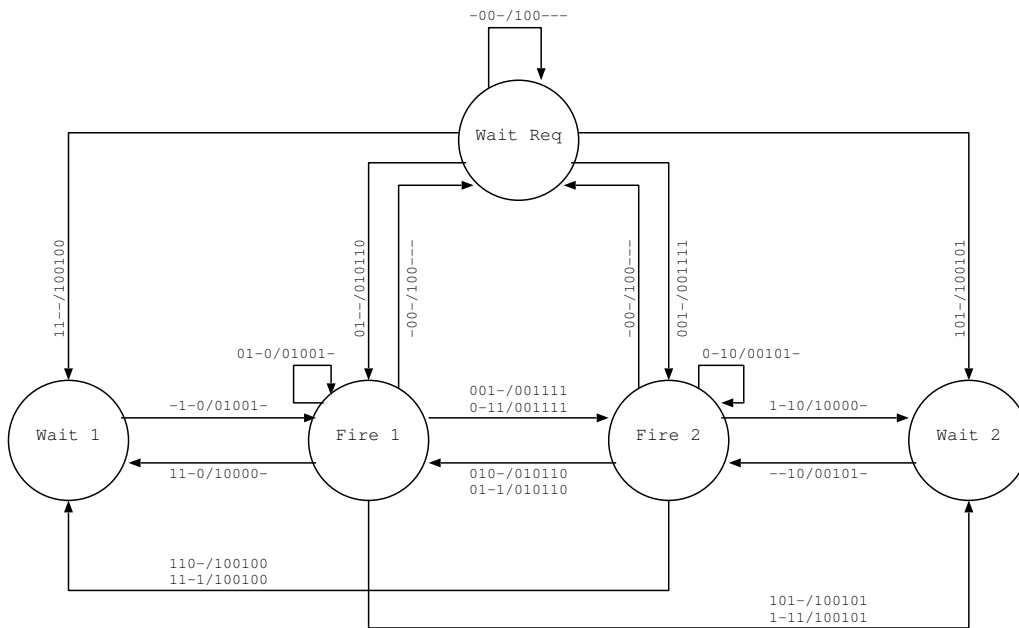


Figure 5.12: Mealy machine of the Round Robin scheduling.

memory. Therefore the state machine takes also care for the control of the switch. The diagram of the state machine is shown in Figure 5.12 and is from the type Mealy Machine [10].

5.5.6 Switch Control

The controller of the switch is responsible for the selection of all the multiplexors. It also decodes the addresses. If the address is from a specific FIFO, the block *switch control* will tell this to the specific connection in the CA. In Figure 5.13 an abstract overview of the switch is given with all the data and address busses and the read/write signals. The state of the multiplexors is depended on the block *switch control*.

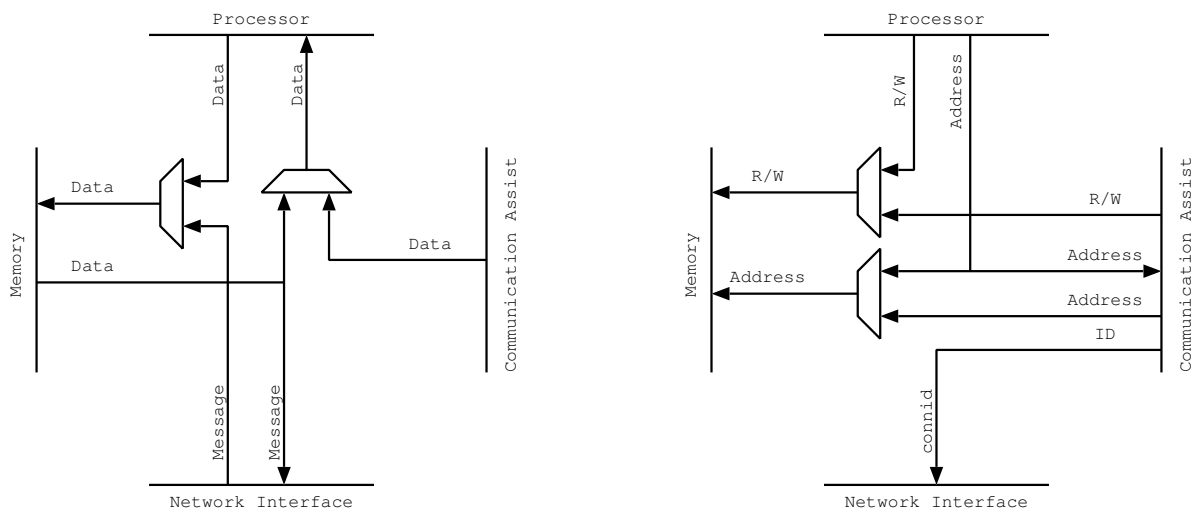


Figure 5.13: Abstract view of the switch. See left for data and right for address and R/W.

Chapter 6

Conclusions and recommendations

Applications that require a computational performance between 10 and 100 giga operations per second can today only be provided by multiprocessor platforms. The current platforms are using central resources like busses and memories. These central resources limit the scalability of the platforms. That is the reason why future multiprocessor systems are using distributed memory and a network on silicon. In hard real-time applications timing deadlines are very important, therefore it is important that the architecture is predictable. To design these complex architectures simulation at three levels of abstraction is assumed. Simulation level 1 is used to verify the behavior of the application. Simulation level 2 is used to verify the functional as well as the temporal behavior. Simulation level 3 is used to verify the cycle true implementation. The main advantages of level 2 simulator compared to level 3 is that the level 2 simulator makes worst-case temporal behavior visible and has a much higher simulation speed.

It is assumed that an application of a multiprocessor consists of multiple tasks. Tasks must have actor semantics such that they can be modeled with an SDF model. Another technique what is used in the architecture is the peek-skip technique. The idea is that every actor in the architecture peeks to look if it can execute. It executes when it is possible and otherwise it will skip executing. This technique makes the applications independent which will increase the predictability. Another result of skipping instead of waiting is that the average performance can increase.

The Æthereal network is assumed as the network in the multiprocessor template. Because of the predictability requirement only the guaranteed throughput streaming channels are used. The guaranteed throughput channels make use of a credit channel between the network interfaces to prevent data loss. In this thesis it is assumed that connections are not shared.

The level 2 simulator executes an SDF model. During this execution tokens will arrive not earlier than in the implementation. In other words the level 2 simulator is conservative. By using conservative SDF models of blocks from the implementation it can be guaranteed that also the composition of these blocks is conservative. In this thesis some basic arbitration and scheduling blocks are modeled and it is proven that these models are conservative. A composition of these blocks is used to model the communication channel of the architecture.

By having an SDF graph of a job it is possible to transform this SDF graph into an HSDF graph. It is shown that the HSDF graph has some interesting properties. For example by examining all simple

cycles in the HSDF graph it is possible to detect deadlock or to derive the minimal throughput. Indirectly it is also possible to determine the minimum FIFO capacities given a required minimal throughput. The reason is that the FIFOs are modeled with a backward edge in which the number of initial tokens on this edge represent the FIFO capacity. By adapting the number of initial tokens the minimal throughput can be affected.

An other property of a strongly connected HSDF graph with fixed actor execution times is that the execution of the HSDF graph will enter a periodic regime. This periodic regime is also called the steady state. After the steady state is detected the behavior becomes periodic and the simulation can be stopped. Detecting the steady state can be done in two different ways. The first way is to observe the starting times of the actors and the second one is to observe the arrival times of the tokens. Observing the arrival times of the tokens can result in a later detection of the steady state than by observing the start times of the actors. The arrival times of the tokens is used in the simulation to detect the steady state because this resulted in a more straightforward implementation.

Every connection in the level 2 simulator is working as a normal FIFO. The difference is in the behavior. When the FIFO is mapped to a communication channel it will set some parameters and has the behavior of a communication channel. The advantages of putting the behavior of the communication channel inside the FIFO is the simplicity of programming and the simulation speed. The behavior of a communication channel can be mapped on a FIFO because there is only one SDF model of a communication channel. Only the parameters and the number of initial tokens of this SDF model are changed. The simulation is fast because the tokens inside a connection are working with time-stamps. Therefore no extra processes have to run because the calculation is done by the methods of the connection. The simulator has two different peek functions. A blocking peek for modeling SDF behavior and a nonblocking peek for control information. The control information can be used for changing the mode of an actor. The detection of the steady state is only done by looking at communication channels and not by looking at FIFOs. In the future it could be interesting to implement a flag in every connection which indicates if the connections should be considered for detecting the steady state. Another problem is that the process, which is calling the steady state function, has to wait a delta cycle before it can call the steady state function. This solution is not very elegant, some future work could be to find the steady state in another way.

The design of the communication assist was based on assumptions of the interfaces from the processing unit, the memory and the network interface. With the design of the communication assist, described in this thesis, it should be possible to arbitrate the local switch with the granularity of 1 clock cycle. This means that the worst-case waiting time for the processor and for the communication assist to access the memory is 1 clock cycle. The best choice for scheduling between connections is round robin with no threshold and a maximum number of words. The reason for that is that this way the maximum average throughput and the lowest average latency will be obtained. When using no threshold it is sufficient to get only full or empty information from the FIFOs, respectively for an output and an input FIFO. The configuration of the communication assist can be done by setting registers. By using registers the blocks in the CA can run in parallel with the result that every clock cycle, that is available, can be used for transferring data. The implementation of the communication assist as shown in this thesis was not verified by cycle true simulation due to lack of time.

List of Figures

2.1	Multiprocessor template.	3
2.2	An application which consists of jobs and jobs which consist of tasks.	4
2.3	Implementation of a guaranteed throughput channel of the Æthereal network.	5
2.4	Abstract view of the communication channel between P1 and P2.	6
2.5	More detailed view of the communication channel between P1 and P2.	6
3.1	An SDF graph.	7
3.2	SDF model of a FIFO with a capacity of 3 tokens between a producing and consuming actor.	8
3.3	An HSDF graph	8
3.4	Plotting for all the actors the difference between successive start times of the specific actor.	10
3.5	The state changes of an HSDF graph from a simple Producer-Consumer example.	13
3.6	Composition of basic blocks and their SDF models.	14
3.7	A implementation of the TDMA wheel with a threshold of N tokens.	14
3.8	SDF model of the TDMA wheel with a threshold of N tokens	16
3.9	A implementation of RR with a threshold of N tokens.	18
3.10	SDF model of RR with a threshold of N tokens.	19
3.11	A implementation of the TDMA wheel with n the number of tokens $1 \leq n \leq N$	21
3.12	SDF model of the TDMA wheel with no threshold but a maximum of N tokens.	22
3.13	A implementation of RR with n the number of tokens $1 \leq n \leq N$	24
3.14	SDF model of RR with no threshold but a maximum of N tokens.	25
3.15	Abstract view of the communication channel with credit paths for fifo capacity.	27
3.16	SDF model of the communication channel.	28
3.17	Simplified SDF model of a communication channel.	28
4.1	Declaration example of a FIFO.	29
4.2	Example of mapping a FIFO to a communication channel.	30
4.3	Syntax of the function write.	30
4.4	Syntax of the function read.	31
4.5	Syntax of the peek function.	31
4.6	Syntax of the peek wait function.	32
4.7	Syntax of the steady state function.	32
4.8	Producer-Consumer example.	34
4.9	SDF Model of the Producer-Consumer example.	34
4.10	The Main Program (main.cc).	35
4.11	Process Network Declaration (pc.h).	35

4.12	Process Network Implementation (pc.cc).	36
4.13	Producer Process Declaration (producer.h).	36
4.14	Producer Process Implementation (producer.cc).	37
4.15	Consumer Process Declaration (consumer.h).	38
4.16	Consumer Process Implementation (consumer.cc).	39
4.17	The output of the example.	40
4.18	The software packages.	41
4.19	Pseudo code of the read function in the hChannel implementation.	42
4.20	Pseudo code of the function FireCaRead in the hChannel implementation.	43
4.21	Pseudo code of the function State in the hChannel implementation.	44
4.22	Pseudo code of the function steadyState(pointer_pn).	45
4.23	Pseudo code of the function steadyState(pointer_pn, n).	46
5.1	Test application for comparing scheduling techniques.	49
5.2	TDMA wheel with a cycle time T_A and a actor time T_{A1} .	49
5.3	The distribution of Latency 1, Latency 2 and the total Latency in case of TDMA.	50
5.4	Round Robin list with a actor $A1$ and other actors A_n .	50
5.5	The distribution of Latency 1, Latency 2 and the total Latency in case of RR. The first example is the case that $T_{An} > T_{A1}$ and second is that $T_{An} < T_{A1}$.	51
5.6	Latency distribution of TDMA and RR.	52
5.7	Latency distribution of RR with threshold and no threshold but a maximum number of words.	52
5.8	Pseudo code for the block FIFO Status.	54
5.9	Pseudo code for the block Fire.	54
5.10	Pseudo code for the block Increment.	55
5.11	Pseudo code for the block Load.	55
5.12	Mealy machine of the Round Robin scheduling.	56
5.13	Abstract view of the switch. See left for data and right for address and R/W.	57
B.1	The Simulation Cycle of SystemC	68
C.1	Main class diagram of HAPI	70
D.1	Overview of the parameters and variables in the channel implementation	72
E.1	Pseudo code for TDMA wheel with a threshold N	74
E.2	Pseudo code for round robin with a threshold N	74
E.3	Pseudo code for TDMA wheel with no minimum threshold, but a maximum of N words	75
E.4	Pseudo code for round robin with no minimum threshold, but a maximum of N words	76
F.1	Abstract view of the architecture in a tile	78
F.2	Architecture of the CA	79
G.1	Examples C-HEAP FIFO protocol	81

List of Tables

1.1	Simulation abstraction levels.	2
3.1	The arbitration/scheduling models that will be explained.	14

Bibliography

- [1] M. Ade, R. Lauwereins, and J.A. Peperstraete. "*Buffer Memory Requirements in DSP Applications*". *Proceedings, Rapid System Prototyping*, pages 108–123, 1994. 66
- [2] F. Bacelli, G. Cohen, G.J. Olsder, and J-P. Quadrat. *Synchronisation and Linearity*. John Wiley & Sons Inc., 1992. 10, 40
- [3] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, J. van Meerbergen, M. Duranton, and L. Steffens. "*Predictable Embedded Multiprocessor System Design*". *Philips DSP-conference*, 2003. 3, 7, 9
- [4] S.S. Bhattacharyya, N. Bambha, M. Khandelia, and V. Kianzad. "*Mapping DSP Applications onto Self-timed Multiprocessors*". *Signals, Systems and Computers*, 1:441–448, 2001. 66
- [5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. ISBN 0-262-03141-8. 44
- [6] S.A. Cristescu. "*YAPI Architecture*". Nat.Lab. Technical Note NL-TN 2001/411, Philips Electronics N.V., 2001. 40
- [7] E. de Kock and G. Essink. *Y-chart Application Programmer's Interface*. Philips Research, Eindhoven, 1.3 edition. 29, 29, 32
- [8] Doulos Ltd. *Fundamentals of SystemC*, 2 edition, 2002. Philips Centre for Technical Training. 33, 67
- [9] O.P. Gangwal, A. Nieuwland, and P. Lippens. "*A Scalable and Flexible Data Synchronisation Scheme for Embedded HW-SW Shared-Memory Systems*". In *ISSS'01: 14th International symposium on system synthesis*, page 1 until 6. The Association for computing Machinery, ACM, Montreal, 30 september - 3 oktober 2001. 53
- [10] A.H. Geerts. *Digitaal Ontwerpen volgens de top-down aanpak*. Academic Service, 1995. ISBN 90-395-0183-1. 56
- [11] S. Goddard and K. Jeffay. "*Managing Memory Requirements in the Synthesis of Real-Time Systems from Processing Graphs*". *Proceedings Fourth IEEE, Real-Time Technology and Applications Symposium*, pages 59–70, 1998. 66
- [12] R.P. Grimaldi. *Discrete and combinatorial mathematics*. Addison Wesley, 3 edition, 1994. ISBN 0-201-60044-7. 15, 15

-
- [13] G. Kahn. "*The Semantics of a Simple Language for Parallel Programming*". In *Proceedings IFIP Congress*, pages 471–475. North-Holland Publishing Co, 1974. 29
- [14] H. Kopetz. "*The Time-Triggered Model of Computation*". *Proc. 19th IEEE Real-Time Systems Symposium*, pages 168–177, 1998. 66
- [15] H. Kopetz and G. Bauer. "*The Time-Triggered Architecture*". *Proceedings of the IEEE*, pages 112–126, 2003. 66
- [16] H. Kopetz and R. Obermaier. "*Temporal Composability*". *Computing & Control Engineering Journal*, pages 156–162, 2002. 66
- [17] E.A. Lee and D.G. Messerschmitt. "*Synchronous data flow*". *Proceedings of the IEEE*, 1987. 7, 9
- [18] A. Radulescu, J. Dielissen, K. Goossens, E. Rijpkema, and P. Wielage. "*An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming*". 2003. Philips Research Laboratories, Eindhoven, The Netherlands. 1, 5, 48
- [19] E. Rijpkema, K.G.W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. "*Trade offs in the design of a router with both guaranteed and best-effort services for network on chip*". *Design, Automation and Test in Europe Conference (DATE)*, pages 350–355, 2003. 1, 5
- [20] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors Scheduling and Synchronization*. Marcel Dekker, Inc., 2000. ISBN 0-8247-9318-8. 7, 8, 9, 33, 66
- [21] A.H. Timmer, F.J. Harmsze, J.A.J. Leijten, M.T.J. Strik, and J.L. Van Meerbergen. "*Guaranteeing on- and off-chip communication in embedded systems*". *Proceedings IEEE Computer Society Workshop On*, pages 93–98, 1999. 48

Appendix A

Related Work

One of the reasons to make a detailed model of the communication between two processes on different processors is to get a better idea about the responsibility of the components inside the communication channel. Another property of the detailed model is the transient state. By using the transient state the behavior of the model is closer to the behavior of the implementation, which can be very useful when simulating soft real-time applications. By analyzing the SDF model of the complete application it is possible to derive the throughput and to detect the critical components in the implementation. The critical components can be computation as well as communication. The detailed model will also be used to derive the minimum capacity of all the individual FIFOs. The presented models of the arbitration and scheduling schemes are the worst-case scenarios of the implementation but they are also exact, which means that it is possible that the worst-case situation in the implementation can occur.

For the literature search models for computation as well as models for communication have been searched. Another objective was to find scheduling and arbitration techniques and their models. The last objective was to find analyzing techniques for example to derive the throughput or the buffer requirements in the system.

SDF is a well-known model of computation that is widely used in the Digital Signal Processing (DSP) domain. Numerous of commercial DSP tools have been developed that incorporate SDF semantics, i.e. COSSAP by Synopsys, SPW by Cadence, ADS by Hewlett-Packard or DSP DesignStation by Mentor Graphics. In the literature the SDF model is mostly used for scheduling actors on the processors. Scheduling strategies for multiprocessor DSP are classified into four types: fully-dynamic, static assignment, self-timed and fully-static. The self-timed strategy is popular and efficient for the DSP domain due to the combination of robustness, predictability and flexibility. In case of self-timed scheduling the assignment of actors to processors and the order of actors on the processor is performed at compile time. Determining when each actor should execute is performed at run-time. In self-timed scheduling each processor executes the actors, assigned to it, in a fixed order that is specified at compile time. Before executing an actor, a processor waits for the data needed by that actor to become available. Therefore this scheduling technique is called blocking. In this thesis another form of self-timed scheduling is used. The assignment of actors and the order of the round robin list is performed at compile time. Determining when each actor should execute is performed at run-time. The actors check for input data and output space before executing. If the actor can execute it will execute else it will skip. Therefore this scheduling technique is called

non-blocking. Skipping of actor executions will result in a different execution order of actors on a processor. The main difference of the scheduling technique in this thesis and the scheduling techniques in the literature is that the scheduling technique in this thesis is non-blocking.

Another form of non-blocking scheduling is used in the Time-Triggered Architecture (TTA) [16] [15]. The TTA is a hard real-time system that makes use of the time-triggered model of computation [14]. In the case of time-triggered the start of a task is triggered by the progression of a global notion of time, i.e. when the global time reaches a specified value. In a time-triggered system every task will periodically observe the state of its environment to determine whether it should execute. Therefore the task will not block the processor. The instants when a task execution start is assumed to be independent of the data involved. It is further assumed that the worst-case execution time of a task is known. The TTA is modeled as a set of nodes that are interconnected by a real-time communication system. A node consists of a communication controller and a host computer. The interface between the communication controller and the host computer is called the communication network interface. The communication in a real-time system must satisfy the temporal constraints. Guarantees for the fulfilment of temporal constraints require maximum load assumptions. Temporal constraints are fulfilled by constructing a static scheduling at compile time. Therefore the time-triggered system satisfies temporal predictability.

In the literature it is not easy to find detailed models of communication between two processors. In some cases the communication costs are incorporated into the graph model by explicitly including communication actors and setting the execution times of these actors to equal the communication costs [4]. In this case the communication latency is considered in the model. When searching for models to model arbitration schemes no results were found.

Another objective was to do analysis on the SDF model. How to analyze the minimal throughput and the FIFO requirements is explained in [20]. There the analyzing is performed on an HSDF model that is derived from the original SDF model. Another way to do analysis is by rules. For example [1] developed rules for SDF graphs to determine the minimal required FIFO capacities that still guarantee a deadlock-free implementation. Another model that is found in the literature is the Processing Graph Method (PGM), which is used for evaluating and managing processor demand, latency and memory usage in distributed systems [11]. The PGM graph is quite similar to the SDF graph. A difference is that an actor distinguishes at the input a threshold amount of input tokens before start executing and a number of tokens that will be consumed during executing. The number of tokens consumed per execution of the actor can be lower or equal to the threshold amount of tokens. In the case of an SDF graph the threshold amount of tokens is equal to the number of tokens that are consumed during the actor execution. A threshold amount greater than the amount of consumed tokens during execution is often used in signal processing filters.

With the SDF models of the scheduling and arbitration techniques, as presented in this thesis, it is possible to combine computation with communication in one SDF graph. Therefore functional as well as temporal behavior for hard real-time applications can be analyzed. Another advantage is that the capacities of all the individual FIFOs of the implementation can be derived by analytical methods.

Appendix B

The Simulation Cycle

SystemC simulation [8] follows the evaluation-update paradigm where all processes that are ready to be executed are executed, and only then are their output signals updated. The SystemC library includes a scheduler that handles all events on signals, and which schedules processes when the appropriate events happen at their inputs.

The scheduler in SystemC executes the following steps during simulation, see also Figure B.1:

1. Initialization phase - All processes (except SC_CTHREADs) execute in an unspecified order until they suspend.
2. Evaluation Phase. Select a process that is ready to run and resume its execution. This may result in immediate event notifications, which may result in additional processes being made ready to run in this same phase.
3. If there are still processes ready to run, go to step 2.
4. Update Phase. Execute any pending updates due to requested updates made in step 2.
5. If there are no more timed notifications (events waiting to happen in the next delta cycle), work out which processes need to run and go to step 2.
6. If there are no more timed notifications, simulation is finished.
7. If simulation has not finished, i.e. there are pending timed notifications (in the future), advance the current simulation time to the earliest pending timed notification.
8. Work out which processes are ready to run due to the events that have notifications at the current time. Go to step 2.

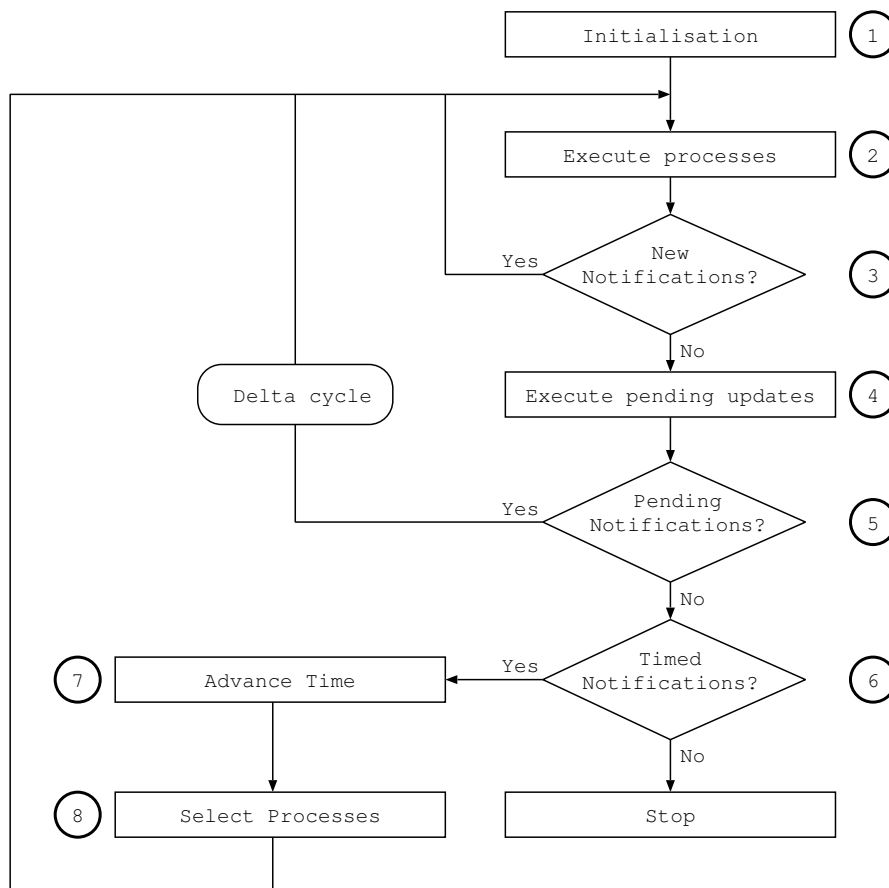


Figure B.1: The Simulation Cycle of SystemC

Appendix C

HAPI main class diagram

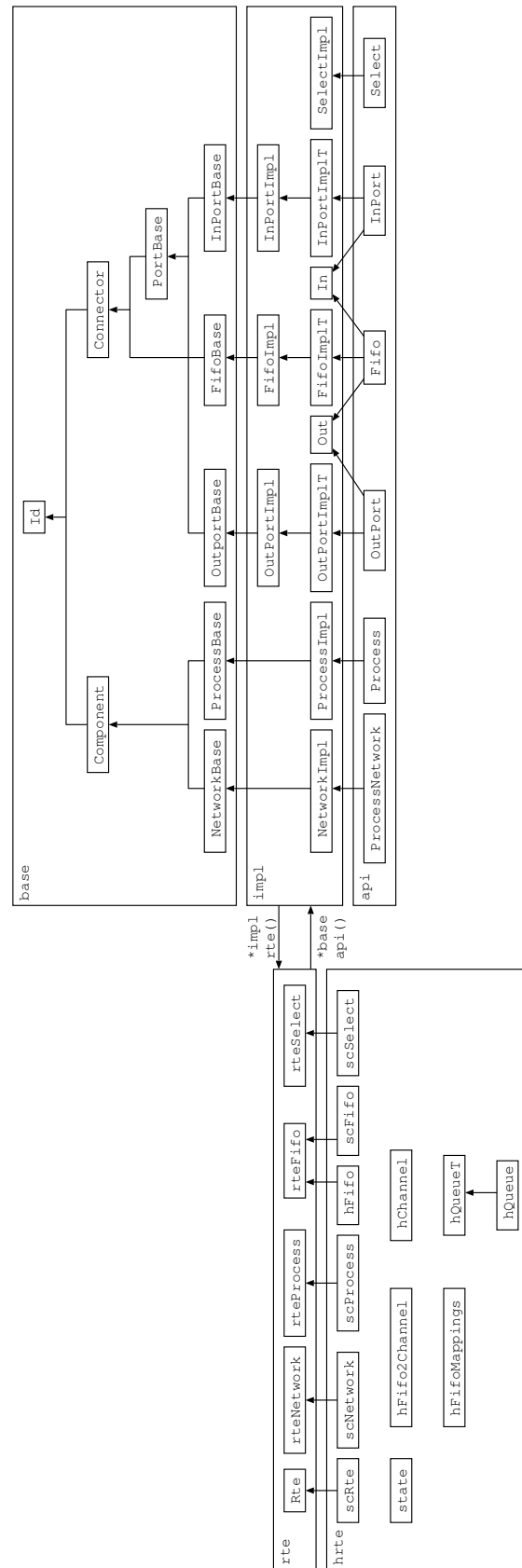


Figure C.1: Main class diagram of HAPI

Appendix D

Parameters and variables in the class hChannel

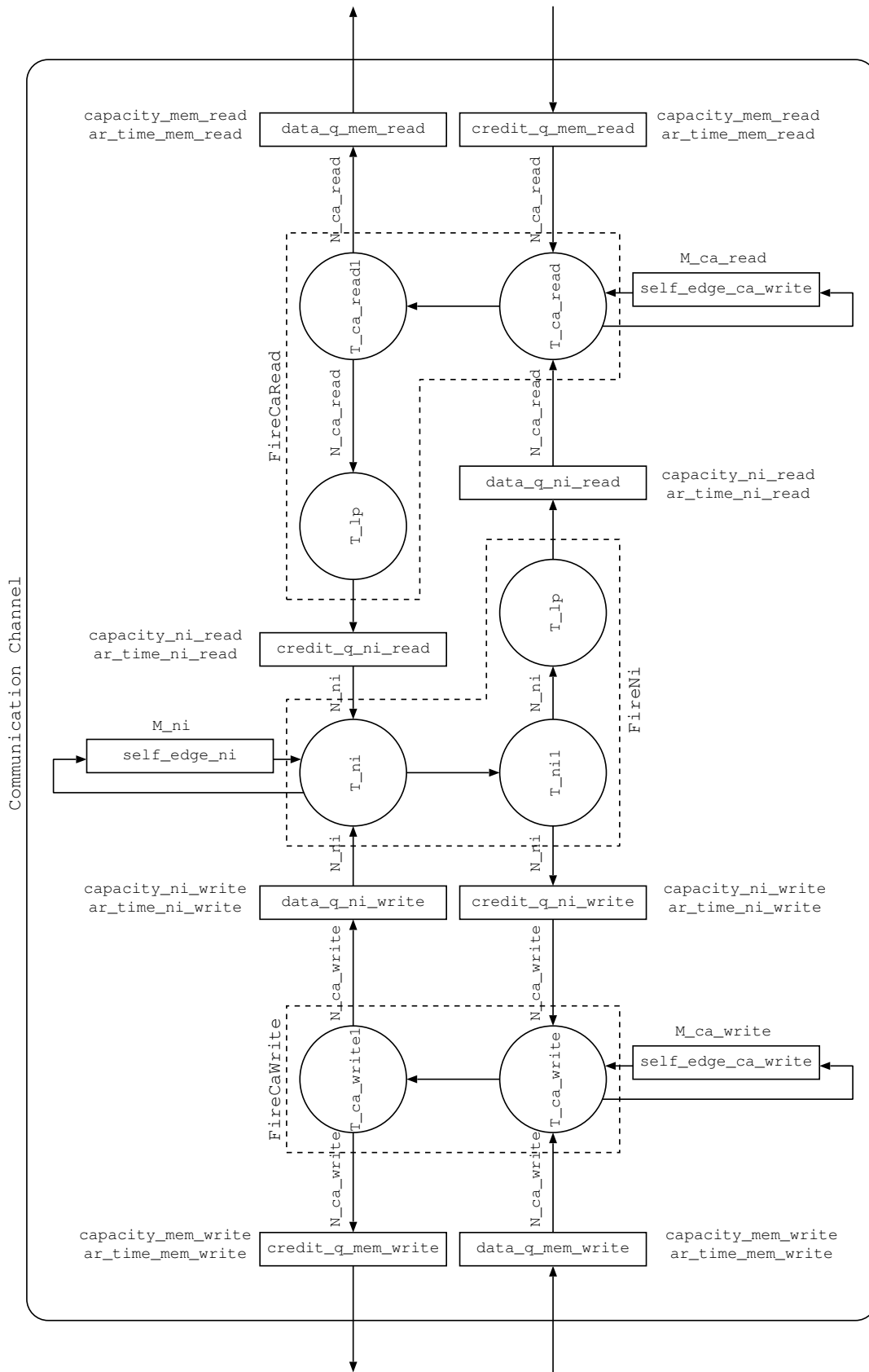


Figure D.1: Overview of the parameters and variables in the channel implementation

Appendix E

Scheduling implementation model

```
TDMA_Equal_N()
{
  time_scheduling=$T_{schedule}$;
  initial_timewheel=Random(0, $T_{timewheel}$);
  wait(initial_timewheel);
  while()
  {
    if( peek(FIFO_A, N) )
    {
      time_process=Random(0, $T_{process}$);
      wait(time_process);
      read(FIFO_A, data_pointer, N);
      write(FIFO_B, data_pointer, N);
      wait(end_of_timeslot);
    }
    else
    {
      wait(timeslot);
    }
    wait(time_scheduling);
  }
}
```

Figure E.1: Pseudo code for TDMA wheel with a threshold N

```
RR_Equal_N()
{
  while ()
  {
    time_scheduling=Random(0, $T_{schedule}$);
    wait(time_scheduling);
    if( peek(FIFO_A, N) )
    {
      time_process=Random(0, $T_{process}$);
      wait(time_process);
      read(FIFO_A, data_pointer, N);
      write(FIFO_B, data_pointer, N);
    }
  }
}
```

Figure E.2: Pseudo code for round robin with a threshold N

```
TDMA_Maximum_N()
{
    time_scheduling=$T_{schedule}$;
    inital_timewheel=Random(0, $T_{timewheel}$);
    wait(inital_timewheel);
    while()
    {
        if( peek(FIFO_A) )
        {
            read(FIFO_A, data_pointer[0]);
            int k=1;
            while( (k < N) && peek(FIFO_A) )
            {
                read(FIFO_A, data_pointer[k]);
                k++;
            }
            time_process=Random(0, $T_{process}$);
            wait(time_process);
            write(FIFO_B, data_pointer, k);
            wait(end_of_timeslot);
        }
        else
        {
            wait(timeslot);
        }
        wait(time_scheduling);
    }
}
```

Figure E.3: Pseudo code for TDMA wheel with no minimum threshold, but a maximum of N words

```
RR_Maximum_N()
{
  while ()
  {
    time_scheduling=Random(0, $T_{schedule}$);
    wait(time_scheduling);
    if( peek(FIFO_A) )
    {
      read(FIFO_A, data_pointer[0]);
      int k=1;
      while( (k < N) && peek(FIFO_A) )
      {
        read(FIFO_A, data_pointer[k]);
        k++;
      }
      time_process=Random(0, $T_{process}$);
      wait(time_process);
      write(FIFO_B, data_pointer, k);
    }
  }
}
```

Figure E.4: Pseudo code for round robin with no minimum threshold, but a maximum of N words

Appendix F

Architecture of the CA

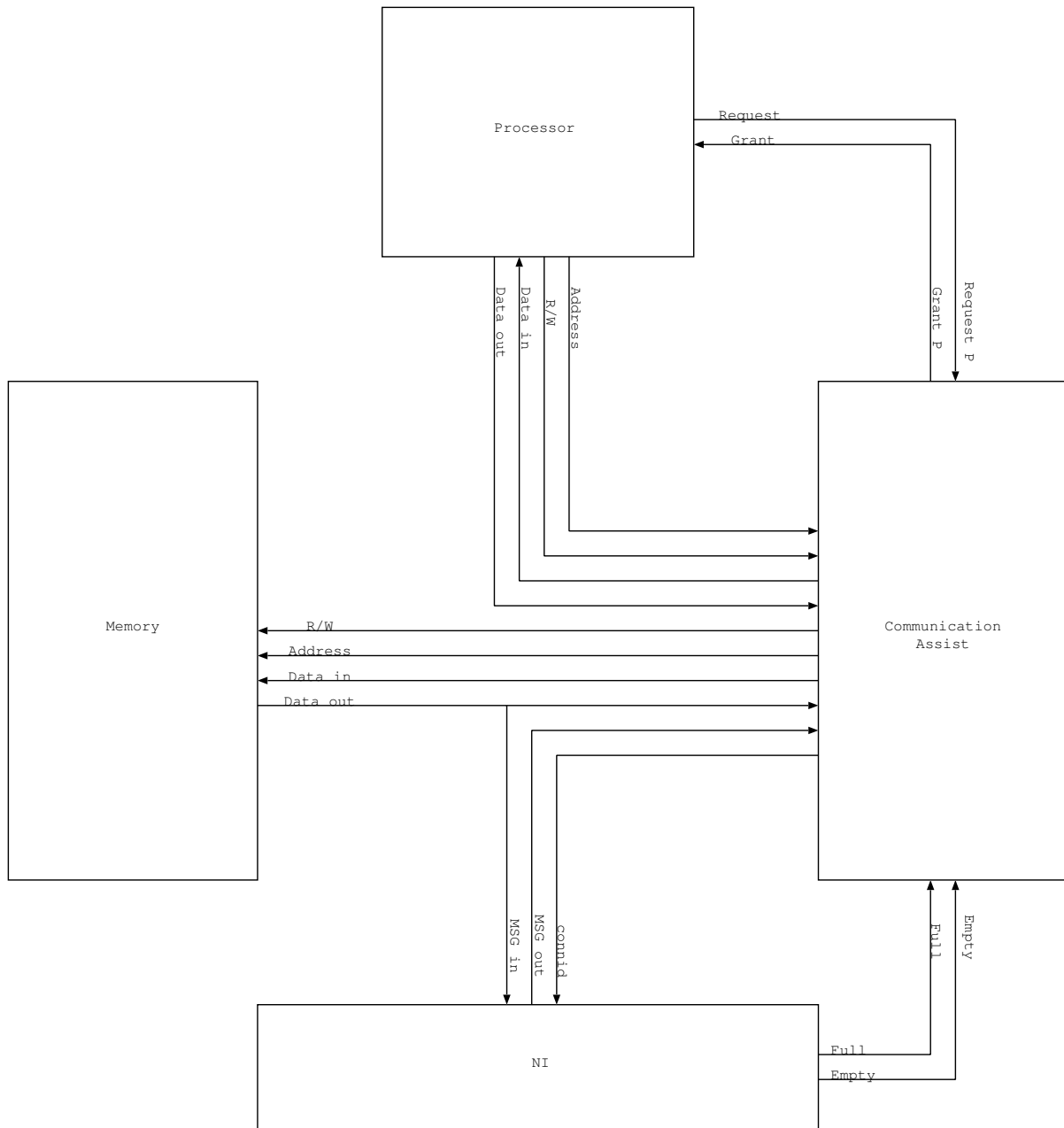


Figure F.1: Abstract view of the architecture in a tile

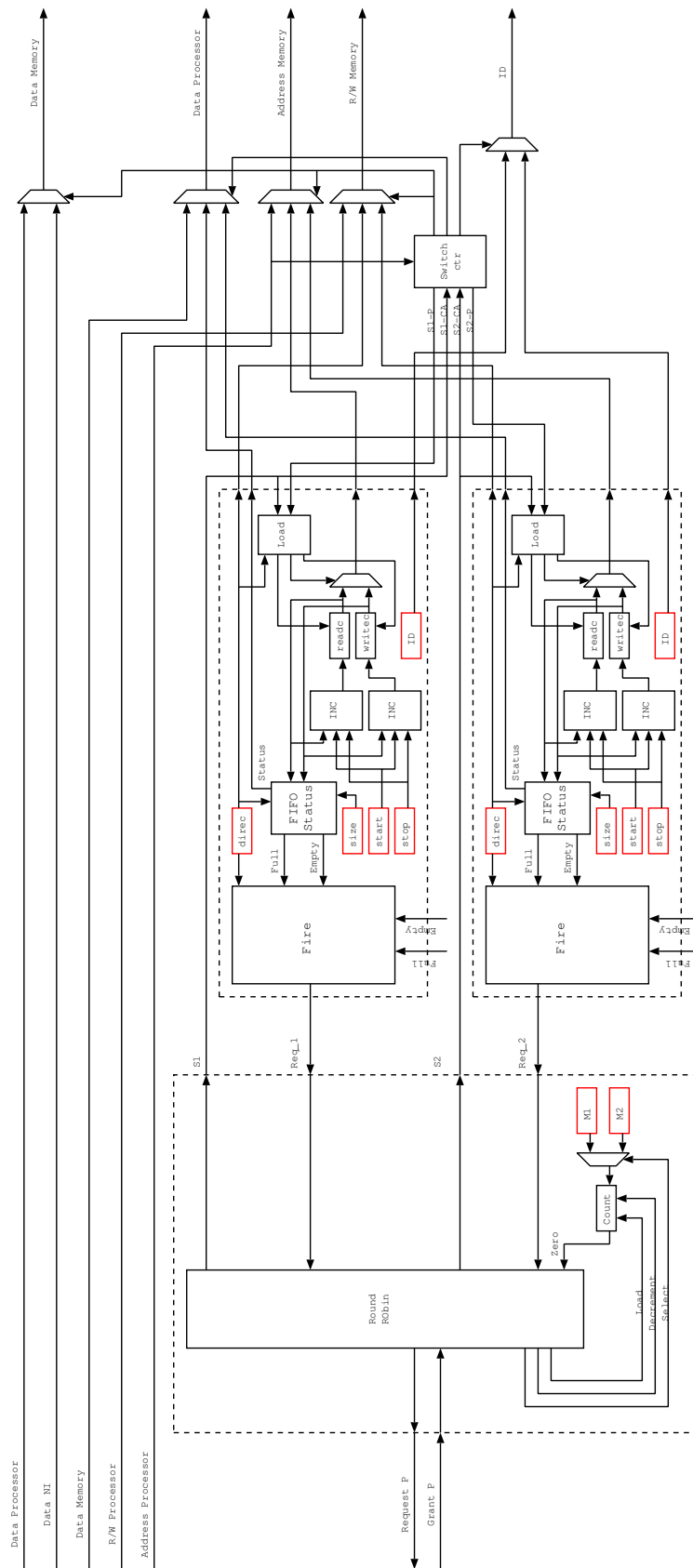


Figure F.2: Architecture of the CA

Appendix G

Examples C-HEAP

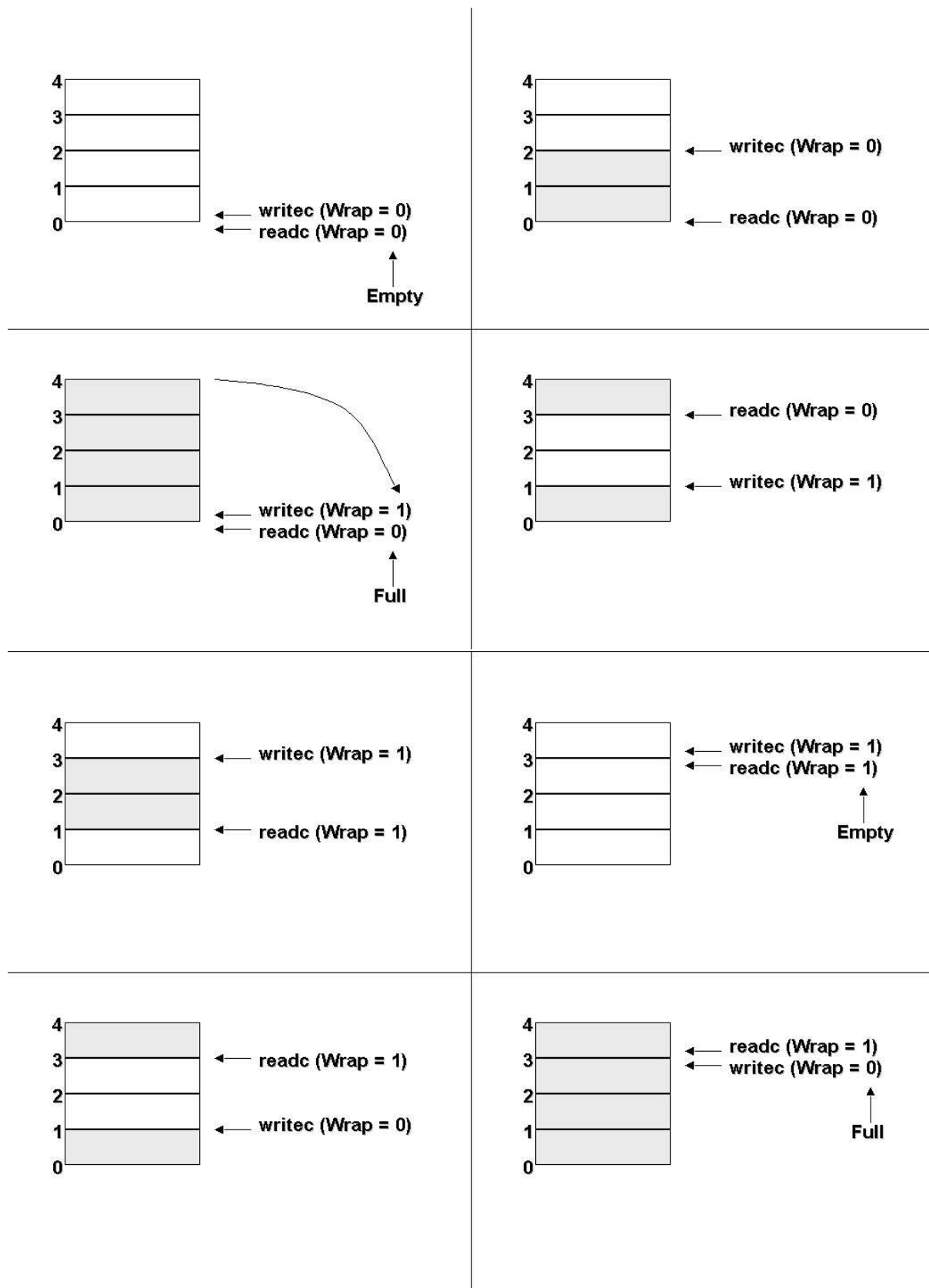


Figure G.1: Examples C-HEAP FIFO protocol

Distribution

Nat.Lab./PI	WB-5	
Department Head:	A. van der Werf	WL-11

Abstract

M. Duranton	Nat.Lab.	WDC3 033
G. Essink	Nat.Lab.	WDC3 031
K. Goossens	Nat.Lab.	WDC3 016
A. Radulescu	Nat.Lab.	WDC3 016

Full report

M.J.G. Bekooij	Nat.Lab.	WDC3 013
J.L. van Meerbergen	Nat.Lab.	WDC3 030
B. Mesman	Nat.Lab.	WDC3 015
O.M. Moreira	Nat.Lab.	WDC _p 036
P. Poplavko	Nat.Lab.	WDC3 041
R. Hoes	Nat.Lab.	WDC3 007
A. van der Werf	Nat.Lab.	WDC3 034