School of Electrical, Electronic & Computer Engineering

Newcastle University

Asynchronous Data Communication Mechanism Models in Matlab and Their Applications

Fei Hao

Technical Report Series

NCL-EECE-MSD-TR-2007-120

July 2007

Contact:
Fei.Hao@newcastle.ac.uk

Supported by EPSRC grants GR/R32666 and GR/R32895

School of Electrical, Electronic and Computer Engineering

University of Newcastle upon Tyne

Asynchronous Data Communication Mechanism Models in MATLAB and Their Applications

A thesis submitted in partial fulfilment

of the requirements for the degree of

Doctor of Philosophy

Fei Hao

May 2007

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgement

I would like to give my sincere thanks to all the persons who have helped me during my PhD research.

I express my complete gratitude to my supervisors, Dr. Graeme Chester and Professor Alex Yakovlev. Because of their enlightened guidance and open minds, I have been able to enter this exciting field of asynchronous system designs. Their advice encouraged me to complete this study.

I express my gratitude to Dr. Fei Xia whose wholehearted support was continuous during the past 5 years. His advice has led me out of difficulties on many occasions. The help from him was not only in academic research but also in many other aspects.

I am also grateful to Dr. Delong Shang for introducing Cadence to me and for many discussions on asynchronous circuits. Without these discussions, the hardware implementation in Chapter 4 would not be completed.

I thank Professor Hugo Simpson, Professor Anthony Davies and all the members involved in the COHERENT project. I benefitted very much from the presentations and discussions at the meetings.

My appreciation also goes to Dr. Danil Sokolov, Dr. Agnes A. Madalinski and Miss Deepali Koppad who helped me in solving several problems on thesis editing in Lyx. Thanks go to Danil and Mr. Xuebo Zhao for sharing their experience of the writing up stage. Mrs. Miao Wang who provided me a link for scientific writing is also thanked.

Many thanks to all the colleagues in the VLSI group for their constant encouragement, and to all my friends for their support and making my life interesting.

I would also like to express my thanks to the EPSRC, School of EECE and Newcastle University for their financial support during my research.

My special thanks go to Mrs. Rong Li and Dr. Xiaojun Meng. It was Rong who recommended and encouraged me to study at Newcastle University. Without the information from Rong and Xiaojun, I would not have had the opportunity to enter this fantastic group.

My great appreciation belongs to my family. They provided me with patience, encouragement and love constantly. Particularly during the writing up stage, my parents and wife took over most of the housework and tried their best to offer me a comfortable working environment.

Abstract

This thesis presents the approaches for creating and testing different asynchronous communication mechanism (ACM) models in MATLAB and investigations on ACM applications in control systems.

To build the control path models of ACMs in MATLAB, two approaches are used. The first one is according to their Petri net specifications. The second one is according to their algorithms.

The first approach is based on the transformation from Petri net to Stateflow - a software package in MATLAB. This transformation is achieved by observing the similarities between the Petri net and Stateflow, and is suitable for all the well-formed Petri nets by which ACMs are mainly described.

The second approach is based on analysing the information exchanges in the algorithms. Each information exchange is represented by a handshake model. ACM control path models are built by connecting these handshake models together.

Complete ACM models are obtained by connecting the control paths and the datapaths which are made up of memory and switch blocks. These models were successfully tested in a test bench which generates reading and writing requests randomly.

A brushless DC motor system was used to investigate the application of ACMs in control systems. Two ACMs were included into the system: one was in the feedback to deal with the asynchrony between the speed sensor and the speed controller, the other was between two controllers to deal with their speed differences. The result shows that the system with ACMs not only works, but also avoids the blocks, which slow down the system response and worsen the overall performance, caused by a traditional buffer solution.

Abbreviations

A/D	Analog to Digital Conversion
ACM	Asynchronous data Communication Mechanism
ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuit
BB	Bounded Buffer
D/A	Digital to Analog Conversion
DAC	Digital to Analog Converter
DC	David Cell
DCS	Distributed Control System
DI	Delay Insensitive
DSP	Digital Signal Processor
e.m.f.	Electromotive Force
EMI	Electro-Magnetic Interference
FM	Fundamental Mode
GALS	Globally Asynchronous Locally Synchronous

GUI	Graphical User Interface
ITRS	International Technology Roadmap for Semiconductors
MATLAB	Matrix Laboratory
Mutex	Mutual Exclusion
NOW	Non Over-write
NRR	Non Re-read
NRZ	Non Return to Zero
OW	Over-write
PID	Proportional, Integral, Derivative
PLC	Programmable Logic Controller
PN	Petri nets
PWM	Pulse Width Modulation
QDI	Quasi Delay Insensitive
RPM	Round per Minute
RR	Re-read
RZ	Return to Zero
SI	Speed Independent
STG	Signal Transition Graph

Chapter 1

Introduction

The real world is asynchronous in nature. Logically, digital systems should be designed in an asynchronous way. However, in the past decades, synchronous systems were dominant because of their easy implementation method.

With the development of modern technology, several inherent problems with synchronous systems became more notable. It is becoming more and more difficult to build a global clock of today's systems. The use of a global clock always leads to high power consumption, which is unacceptable, especially for power-sensitive applications where short battery life is the bane of the users. The EMI (Electro-Magnetic Interference) is a concern in mobile communication applications. Dean [Dea92] proved that systems operated asynchronously have a potential to generate better results. Consequently, asynchronous designs returned to researchers' sight.

Asynchronous data Communication Mechanisms (ACMs) are inter-process communication devices that support the communication of data between writing and reading processes which are unconstrained in, when, and at what rate they can access the mechanism [HP02].

1.1 Motivation

The aim of this thesis is to propose a method of building ACM models at the application level, and to investigate the effect of including ACMs in engineering application systems, particularly in feedback control systems.

1.1.1 Demand for Asynchronous Communication

Inter-process asynchrony is inevitable for computation networks in the future, firstly, because different and diverse functional elements, especially those connecting to analogue domains, tend to have different timing requirements [KEM03, Sim03] and, secondly, because concurrent and distributed system implementations lead to greater asynchrony between components as semiconductor technology advances and the degree of integration increases (the ITRS 2005 "Design" document emphasises multiple clock domains and source-synchronous signalling and predicts networks of self-timed blocks [ITR05]). The size of computation networks is becoming larger, and traffic between the processing elements is increasing. Handling data communication determines performance and other characteristics of such systems.

In truly distributed systems such as sensor networks [KEM03], there is often a desire to have temporal decoupling between digital processes. For instance, parts of a distributed control system may consist of control laws mapped onto hardware embedded into parts of the plant environment, whilst the upper hierarchies of the system may be implemented with software running in general purpose processors which are shared multitasking units. It can be very important to have temporal decoupling between the hardware and the software parts of the control algorithm at the hardware level because of such reasons as:

- avoiding deadlock propagation through the system,
- the desire to have low power characteristics in battery powered units,

- the physical impossibility of keeping everything synchronised in distributed systems,
- different parts of a system requiring radically different processing speeds.

1.1.2 ACM - a Solution for Asynchronous Communications

As a solution for asynchronous communications, ACMs were proposed by Hugo Simpson [Sim90, Sim94] in 1990 and have by now developed into a coherent field including classification, specification, and techniques for implementation, analysis and verification [Sim03, XYCS02].

In ACMs, two independent processes are considered. The one supplying data is called a writer. The one requesting data is called a reader. The memory element between the reader and the writer is known as the datapath. Simpson classified the ACMs into 4 types: *Channel*, *Pool*, *Signal* and *Constant* based on the properties of the reader and the writer, such as "destructive" and "non-destructive". According to the number of memory storage elements, or slots, in the datapath, ACMs can be classified into 2-slot, 3-slot and 4-slot mechanisms. The definitions can be found in Chapter 2. Because the *Constant* type ACM does not perform any communications between the reader and the writer, a new classification based on whether "re-reading (RR)" and (or) "over-writing (OW)" are permitted in the reader and (or) the writer was proposed in [XC02, YXS01]. In this classification, a mechanism called *Message* took the place of the *Constant.* ACM classification was successfully expanded to include types providing more qualitative asynchrony and richer data properties than a traditional FIFO buffer.

The *Channel* type ACM, especially in its FIFO buffer form, has been studied for many years, and its hardware implementations were proposed in [Sut89, HBL+99, LPI01]. Simpson described a fundamental mode (FM) implementation for the *Pool* type ACM [Sim94]. Furthermore, self-timed ACM implementations proposed in

[Sha03, XYS$^+$00, YXS01] were based on their Petri nets specifications [XC00] using a direct translation method [SXY00a].

ACMs are potentially useful in systems with heterogeneous timing as data connectors between processes belonging to different timing domains, which may exist out of necessity. They can also be useful as digital mimics for various types of data connections in analogue systems, with different types of ACMs suiting different data requirements. [XHC$^+$04] This makes it clear that these applications should be investigated.

The study of ACMs so far, though extensive, has not extended to their direct modelling in application-level tools. Previous proposals for modelling ACMs at a higher level, treating them as components in larger systems, employed Petri nets [XC00]. This was suitable for the case where systems containing ACMs were regarded and analysed as general discrete event digital systems. However, in order to study the effect of including ACMs in such engineering application systems as control systems, especially when analogue parts are present, ACM models need to be integrated into popular application-level tools such as MATLAB (Matrix Laboratory) [MATa] and MATRIXx [MATc]. MATLAB was chosen because it was widely used and had more support. In addition, the MATRIXx models could be translated to MATLAB ones [MATb]. MATLAB direct to hardware fast prototyping tools are becoming available [Xil], potentially making it possible to save the step of implementing DSP hardware through the traditional VLSI process. Future developments in this direction could potentially lead to the direct hardware implementation of application systems containing ACMs designed and verified in MATLAB. This provides another motivation for this kind of work.

1.1.3 Application to Control Systems

The control systems nowadays tend to be distributed. This trend may be found in the applications from robotic machining to autonomous vehicle control, etc [YTS00]. The changes are based on the considerations of wiring, flexibility and diagnostics. Considering the wiring, instead of connecting to the central controller, measurement and actuation devices can be connected to a local controller which is connected to other controllers via a communication link. This may reduce the overall wiring cost and complexity with increased reliability [Bur04]. Considering the flexibility, a distributed control system is more scalable, resulting in a more modular system in which additional devices can be added to the system directly. Considering the diagnostic, a distributed control system can take better advantage of the additional information due to the existence of the communication interface. Diagnosing this information makes the preventive maintenance easier or helps to troubleshoot problems more efficiently.

Communication delay has a major effect on system performance and can be a major problem in a system with many nodes and a high frequency. According to Yook [YDS00], an approach to solving this problem is reducing the communications between the distributed processors. That is, the processors should communicate only when necessary instead of as fast as possible. Yook proposed a State Estimator Framework [YDS00] to deal with the problem. In this approach, an estimated value is used instead of the real sampled one. The communication only occurs when the difference between the estimated value and the actual value exceeds the preset threshold. As a trade-off, the computation load at each node is increased due to the additional computation related to the estimator. This approach was based on synchronous design. The computation has to be carried out in each clock cycle. The computational load will be extremely high at high frequency. An asynchronous approach could be an alternative to reduce both the computation load at each node and the communications, and consequently, reduce power consumption.

ACMs are well positioned for delivering data between the asynchronous processes in control systems. When an analogue value is presented, it is converted into a digital one through an analogue to digital converter (ADC). A class of ADCs, named as “asynchronous”, has been described in [KYG00]. With the level-crossing scheme [SSV96] and an asynchronous design, a new class of asynchronous A/D converters based on time quantisation is proposed in [ASFR03]. An asynchronous A/D converter combined with an asynchronous controller will decrease both the communications and the computational load. For overall performance, information may need to be passed among the controllers in different time domains. The communications among the controllers will have synchronisation problems. To deal with the synchronisation problem, ACMs can be added.

For example, in a brushless DC motor control system, the speed controller works out a desired current according to the error between the reference and the actual speeds. The torque controller generates the amount of drive based on the error between the desired and the actual currents. The torque controller requires considerably faster control actions than the speed controller. However, the desired current has to be passed from the slow speed controller to the fast torque one. A direct digital link between the two controllers could potentially propagate hazards from one to the other. As a safety-critical process, the torque controller cannot tolerate such hazards, because they may burn out the motor or power electronics. With an ACM inserted between them, the two controllers could be executed independently of each other. The hazard propagation is avoided.

1.2 Publications and Contribution

The main contributions of this thesis include: implementation of a 3-slot *Signal* [HYC+03], Petri net to Stateflow conversion, building ACM models in the MAT-

LAB environment [HC03], investigations on buffered ACM [HXC+04b, XHC+06] and ACM applications in control systems [HXCY04, HXC+04a].

1.2.1 Petri Net to Stateflow Conversion

ACMs are often described as Petri nets. Looking for an approach for conversion from Petri net to Stateflow, whose models could be embedded into the MATLAB environment, is a straightforward way to build such kinds of ACM models in MATLAB. The basic connections in Petri net, such as linear connection, fork, join, merge and choice, were successfully converted. Based on this, a step by step conversion approach is proposed.

1.2.2 Building ACM Models in MATLAB Environment

To investigate applications, the MATLAB models of ACMs were built. The control paths were modelled by two approaches. One is to translate from their Petri net specifications [HC03] using their Petri net to Stateflow conversion, the other is to use 4-phase handshake models to build models according to the ACMs algorithms.

The former one is closer to hardware implementations, which concern the internal structures more. A 3-slot *Signal* is used as an example showing the similarities between the MATLAB model and the hardware circuits.

The models built by the latter approach are used in the investigation of ACM applications. There are two reasons for doing this: first, they are simpler and easier to be understood compared to the ones by the former approach, but they indicate the same properties. And second, the internal structures are not important in the application level models.

1.2.3 Investigations on Buffered ACM

Buffered ACM refers to an ACM containing a buffer, which is used to transfer the data which consists of a stream of items of the same type. Because of the existence of the buffers, the latency becomes an important property to be investigated. After modelling an RR-BB ACM, the relationship among the number of cells, the speed of the reader and the writer, the properties of the freshness and latency are discussed.

1.2.4 ACM Applications in Control Systems

The controllers execute at their own speeds. To pass data among the different time domains, ACMs have been used. A brushless DC motor system is used as an example to investigate the effect when different type ACMs are included. Two ACMs were inserted into the system. One was in the feedback path and the other was between the two controllers. The system performed differently when varying the two ACMs.

1.3 Organisation of Thesis

The thesis is organised as follows:

Chapter 1 Introduction briefly outlines the aims, the contributions and the organisation of this thesis.

Chapter 2 Background presents the background information of this thesis. Firstly, fundamental knowledge on asynchronous designs such as circuit signalling protocols and data representations is introduced. Then ACMs, their properties and classifications are presented. After the brief introduction on control systems, the modelling tools, especially Stateflow [Sta], software embedded in MATLAB, are described.

Chapter 3 Petri Net to Stateflow Conversion presents a direct translation approach from a well-formed Petri net to a Stateflow model.

Chapter 4 Model ACMs in MATLAB first presents the MATLAB model of a 3-slot *Signal* directly translated from its Petri net specification. Second, 3 ACM function blocks are modelled, a 2-slot *Channel*, a 3-slot *Signal* and a 4-slot *Pool*, in MATLAB. With these models, an approach of building ACMs directly from their algorithms is presented. Compared to the models built from the Petri net, these function blocks described in this chapter are much easier to build and understand. Third, this chapter presents the investigations on buffered ACMs. RR-BB is used as example to illustrate how the global view and the modular design blocks are modelled. The properties of RR-BB ACM are also discussed. Finally, hardware implementation of a 3-slot *Signal* is presented to show the similarities between the MATLAB model and the hardware circuits.

Chapter 5 Applications in Control Systems investigates the performance of control systems when ACMs are plugged in. An ACM is included in between the speed controller and the torque controller and another in the feedback path in a brushless DC motor system. The system performance varies when changing the two ACMs. Detailed discussions on the performance were carried out.

Chapter 6 Conclusions and Future Work summarises the contributions of this thesis and the future work.

Chapter 2

Background

2.1 Introduction

This chapter provides background knowledge, including the introduction of asynchronous designs, especially the Asynchronous data Communication Mechanisms, control systems, including feedback control systems, stability criterion, PID controller and brushless DC motor control system, and the tools for modelling and implementations, such as Petri net and MATLAB for the readers so that they are prepared for understanding the subsequent chapters.

2.2 Asynchronous Designs

2.2.1 A Brief History of Asynchronous Designs

Asynchronous circuits were first studied by analysing the nature of input restrictions on sequential circuits fifty years ago. A brief histroy of asynchronous designs could be found in [DN95].

Huffman [Huf64] postulated that there must be a minimum time between input changes in order for a sequential circuit to be able to recognise them as being distinct.

This work was extended by the fundamental contributions of Unger [Ung69, Ung70, Ung71] and McCluskey [McC65]. This led to a class of circuits known as Huffman circuits, also known as fundamental mode circuits.

Muller [MB59, Mul62, Mul67] proposed a different class of circuits by using a ready signal, which is more closely related to modern asynchronous circuits. In Muller circuits, input signals were only permitted when the ready signal was asserted. When the circuit is not ready to accept additional inputs, it holds its acknowledge to indicate that no further requests can be tolerated.

The seminal work by Stephen Unger [Ung69] provided a detailed method for synthesising single-input change asynchronous sequential switching circuits in 1969. Much practical work which followed in the next decade was significantly influenced by him.

The Macromodule project was another noteworthy effort which was conducted at Washington University in St. Louis in 1967. This project provided an early demonstration of the compositional benefits of asynchronous circuit modules. It provided a sound foundation for the numerous macromodular synthesis approaches being investigated today [BS88, BS89, Ebe91].

Charles Seitz was another notable pioneer, who introduced a Petri net like formalism in his Ph.D. thesis [Sei71] which was extremely useful in asynchronous circuits design and analysis. His influence directly resulted in the asynchronous implementation of the first operational dataflow computer [Dav78] and the first commercial graphics system.

Another significant pioneer in the asynchronous world is Victor Varshavsky. He found that the conceptual part of the problems associated with the class of live and safe Petri nets was very close to problems in self-timing and showed a way of translating a Petri net to an asynchronous circuit avoiding complex state-encoding procedures [Var73]. The direct mapping technique [BY02] is based on his work.

Most of the circuits produced by industry are synchronous. An idea of building systems is to compose systems from these predesigned components. A promising method called Globally Asynchronous Locally Synchronous (GALS) architecture is proposed by Chapiro in his PhD thesis [Cha84].

To make the asynchronous design automatically, many synthesis tools were produced, such as Petrify [CKK$^+$96], Balsa [BE00], Tangram [BKR$^+$91], Minimalist [FNT$^+$99], 3D [YDN92], etc.

2.2.2 Classification of Asynchronous Circuits

At the gate level, asynchronous circuits are classified into self-timed, delay-insensitive and speed-independent, depending on the delay assumptions [SF01].

Speed-independent (SI) circuits are the circuits which operate correctly assuming positive, bounded but unknown delays in gates and ideal zero-delay wires, i.e. arbitrary d_A, d_B and d_C, but $d_1 = d_2 = d_3 = 0$ in Figure 2.1 [SF01]. SI definitions based on the different purposes that can be found in [YLSV96]. Its theoretical definition was made by [BM91] in 1991. SI circuits are one of the most broadly used asynchronous design styles.

Delay-insensitive (DI) circuits are the circuits that operate correctly with positive, bounded but unknown delays in wires as well as in gates, i.e. arbitrary d_A, d_B, d_C, d_1, d_2 and d_3 =0 in Figure 2.1. In other words, DI circuits' operations are independent of the delays in both the gates and the wires. In fact, the class of DI circuits is rather small. [Mar90a] showed that only circuits composed of C-elements and inverters can be delay-insensitive.

Circuits that are delay-insensitive with the exception of some carefully identified wire forks where $d_2 = d_3$ in Figure 2.1 are called quasi-delay-insensitive (QDI). Such wire forks, where signal transitions occur at the same time at all end-points, are called *isochronic* [Mar90b].

The correct operations of the self-timed circuits rely on more elaborate and

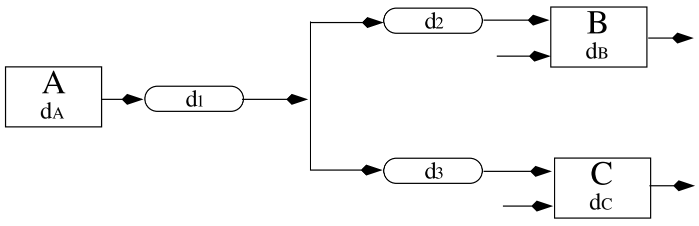


Figure 2.1: A Circuit Fragment with Gate and Wire Delays

engineering timing assumptions.

2.2.3 Handshake Protocols

As stated in [Bai00]: in asynchronous designs, the information is transferred between the sender and the receivers using handshake protocols. The sender generates a request when it is ready to transfer, and the receiver sends an acknowledgement when it is ready to receive. These may occur on dedicated signalling wires, or may be implicit in the data encoding [Mye01]. In either case, the first event indicates information validity, and the second one indicates the acceptance and readiness on receiving information.

The request and acknowledgement may be passed using one of the following two handshake protocols: 2-phase handshake protocol, also called non return to zero scheme (NRZ), and 4-phase handshake protocol, also known as return to zero scheme (RZ).

- 2-Phase Handshake

In the 2-phase handshake protocol, as shown in Figure 2.2, the polarity of a transition is not important. The two transitions are regarded as signalling events. If information is ready, the sender changes the level of the request signal (1 to 0, or 0 to 1). On finishing receiving the information, the receiver performs a transition of the acknowledgement signal. The information is kept valid during the period between request and acknowledgement to ensure it can be obtained correctly by the receiver.

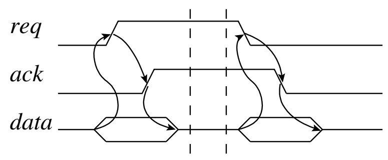


Figure 2.2: 2-Phase Handshake Protocol

Proponents of the 2-phase handshake protocol try to use the lack of a return to zero phase to achieve higher performance and lower power circuits.

- 4-Phase Handshake Protocol

Signal levels are important in the 4-phase handshake protocol because they are used to indicate the validation and the acceptance of data. Therefore, this protocol is also called level signalling. In this protocol, the request and acknowledgement signals have to be returned to zero so that they can stay in the same state after information has been delivered. When information is ready for delivery, the sender generates a request signal (which can be either high or low, depending on the designer). After obtaining the information, the receiver sends an acknowledgement to the sender. On receiving the acknowledgement, the request is reset by the sender. Finally, the receiver withdraws the acknowledgement. This is illustrated in Figure 2.3.

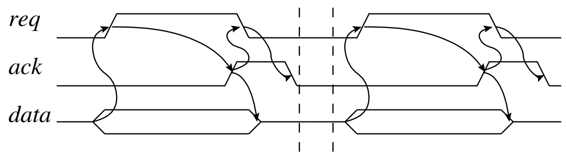


Figure 2.3: 4-Phase Handshake Protocol

4-phase protocol and 2-phase protocol can be converted to each other. The conversions are well documented in Liu's Ph.D. thesis [Liu97].

4-phase control circuits are often simpler than 2-phase ones because the signalling lines can be used to drive level-controlled latches. Therefore, the handshake protocols mentioned in this thesis use the 4-phase protocols.

2.2.4 Data representation

In asynchronous designs, the data representation includes single-rail, dual-rail, 1-hot (1-of-n) and m-of-n schemes.

The single-rail [Pee96] data uses normal Boolean levels to encode the data, and the timing signals are passed from separate request and acknowledge wires. It is the same encoding method as that in conventional synchronous designs. Single-rail is also known as bundled-data protocol. The term "bundled-data" hints at the timing relationship between the data signals and the handshake signals, while the term "single-rail" hints at the use of one wire to carry one bit of data.

The dual-rail [Ver88] data use two wires to encode the information for one bit. Table 2.1 illustrates the dual-rail encoding scheme. *d.t* and *d.f* represent the two wires. In the transfer, only one wire has activity, that is, only one of the two wires can be 1, the other is 0. An n bits dual-rail data item is encoded by 2*n signal wires. The request signal of a dual-rail data item is also implicit in the code, by which it is possible to determine when the entire data word is valid or withdrawn.

	d.t	d.f
Empty("E")	0	0
Valid "0"	0	1
Valid "1"	1	0
Not in use	1	1

Table 2.1: Dual-Rail Encoding

Dual-rail encoding can be seen as a 1-hot encoding of that bit and often it is useful to extend to 1-of-n encodings in control logic and higher-radix data encodings. In 1-hot encoding, each state s_i is represent by a vector y, where $y_i=1$ and $y_j=0$ for $i \neq j$. Examples can be found in [YVMS95].

If the focus is on communication rather than computation, then m-of-n encodings may be of relevance. In m-of-n encodings, activities occur on more than one wire to indicate one possible code. 1-of-n and dual-rail are both special cases of m-of-n encodings.

2.2.5 Asynchronous Components

In the past decade, researchers proposed many useful components for asynchronous designs. A brief introduction is given below for the components used in this thesis. These components include a David Cell (DC), a mutual exclusion (Mutex), an SYNC, a dual-rail D latch and a dual-rail multiplexer.

- David Cell

DC was firstly introduced by Rene David. He proposed the simplest DC in his work [Dav77]. This kind of DC is built essentially around SR flip-flops as shown in Figure 2.4.

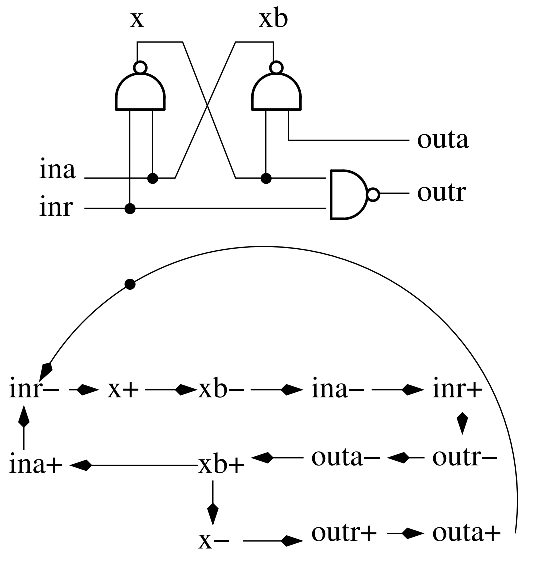


Figure 2.4: Basic David Cell

Each DC has a pair of complementary stable states, which are used for representing the presence and absence of a token in a corresponding 1-safe PN place. As shown in Figure 2.4, the internal state (1, 0) in the flip flop (x, xb) is associated with the presence of a token in the place. The opposite state (0, 1) in the flip flop represents the absence of a token in the place. This DC is a negative active component, as a result, all the four signals, *ina, inr, outa* and *outr,* stay at a high level normally, and the active level is low.

A DC works as follows: the internal state is (0, 1), absence of a token, initially, therefore, *ina* and *outr* are both 1. Once *inr* is activated, x is set to 1 and because of *outa* remaining inactive, 1, xb changes to 0. The internal state has now changed to (1, 0), which means that the token has been transferred to the current place. When xb becomes 1, the preceding DC withdraws the request signal *inr*, which enables the current DC to control a further event. Because both x and *inr* are high, the *outr* is activated to give occasion to fire the events controlled by this DC, and then send a request to the subsequent DC. Once the *outa* signal is received from a succeeding DC, the internal state becomes (0, 1) again, and then the *outr* changes to 1. More details can be found in [VM96b, YK98].

- Mutex

Mutex [Sei80] is the abbreviation for the Mutual exclusion element. It was first introduced as a programming technique that ensures that only one program or routine at a time can access some resource, such as a memory location, an I/O port, or a file, often through the use of semaphores, which are flags used in programs to coordinate the activities of more than one program or routine.

The Mutex controls the exclusive access of two or more independent processes to a shared resource. The required behaviour is:

1. processes should be granted access in the order of their requests,
2. simultaneous requests are served nondeterministically [WB00], and
3. only one process is granted at a time

In hardware design, the Mutex is a circuit used to provide a delay for a signal when metastability occurs. A Mutex has the same number of inputs and outputs. It grants the earlier request and holds the other request until the resource is released. The Mutex is composed of a flip-flop and a metastability resolver, as shown in Figure 2.5. If two requests come to the Mutex very close in time, the output of the flip-flop could become metastable. When metastability occurs, the metastability resolver will not change its outputs until the metastability is settled. The Mutex in this thesis is high level active.

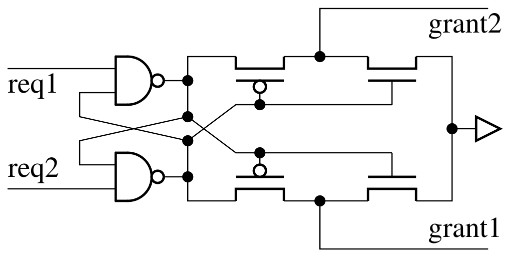


Figure 2.5: Mutex

- SYNC

A SYNC [YXS01, XC02], also known as a sampler, is a synchroniser for two or more signals. It is always used in handling self-timed to synchronous interfaces and vice-versa. It is shown in Figure 2.6.

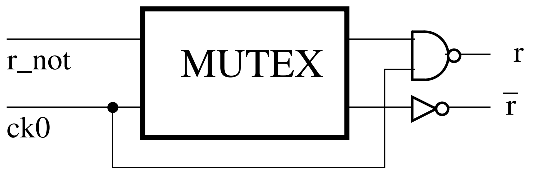


Figure 2.6: SYNC

This circuit is made up of a Mutex, a NAND gate and an inverter. Initially, *r_not* and *ck0* are both inactive (in low level). Once the *ck0* is activated, the output $\overline{r}$ is reset as active. On releasing the *ck0*, $\overline{r}$ becomes inactive again. When *r_not* is activated first, it will not affect the output until *ck0* arrives to reset *r*. Once the *ck0* is withdrawn, *r* is set accordingly.

- Asynchronous D Latch

Asynchronous D latches are often used in the datapath to store data items when a request comes. One asynchronous D latch is able to store one bit with dual rail outputs. Figure 2.7 shows the circuit of an asynchronous D latch [VMS95].

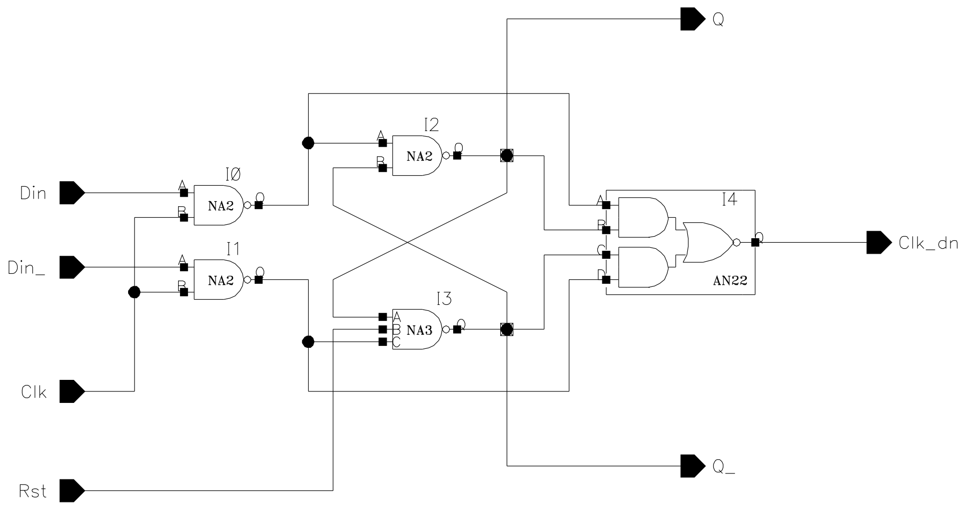


Figure 2.7: Asynchronous D Latch

The first stage of the asynchronous D latch is a clocked RS latch with an *Rst* input to initialise *Q* to 0. The inputs *Din* and *Din_* are dual rail bits, i.e., they are always complementary. They can only be stored in the latch when the *Clk* becomes high. The *Clk* input is connected to the request signal. The second stage is a complex gate to generate a completion signal. When the bits are stored into the latch, the inputs A and B of the complex gate are complementary, so are the inputs C and D. Consequently, the output shows a high. Once the *Clk* is released, the outputs of *both I0* and *I1* become high. This will not affect the outputs of *I2* and *I3*, in other words, they are still complementary. Because only one input of *I4* is low, it outputs a low level.

One D latch is able to store only one bit with dual rail outputs. One byte of data requires an asynchronous D latch set, including 8 asynchronous D latches connected together. The completion signal will not be sent until all the 8 bits are successfully stored. One asynchronous D latch set represents one slot in the datapath.

- Asynchronous Multiplexer

An asynchronous multiplexer is a device that selects one signal among a number of input signals according to the control inputs. The asynchronous multiplexer set in this design is to select one data byte out of three. Similar to the asynchronous D latch, each byte must be dealt with bit by bit. Therefore, an asynchronous multiplexer set is made up of 8 identical asynchronous multiplexers. Figure 2.8 shows the circuit for the asynchronous multiplexer, which is modified from the dual-rail asynchronous multiplexer shown in [PFF96].

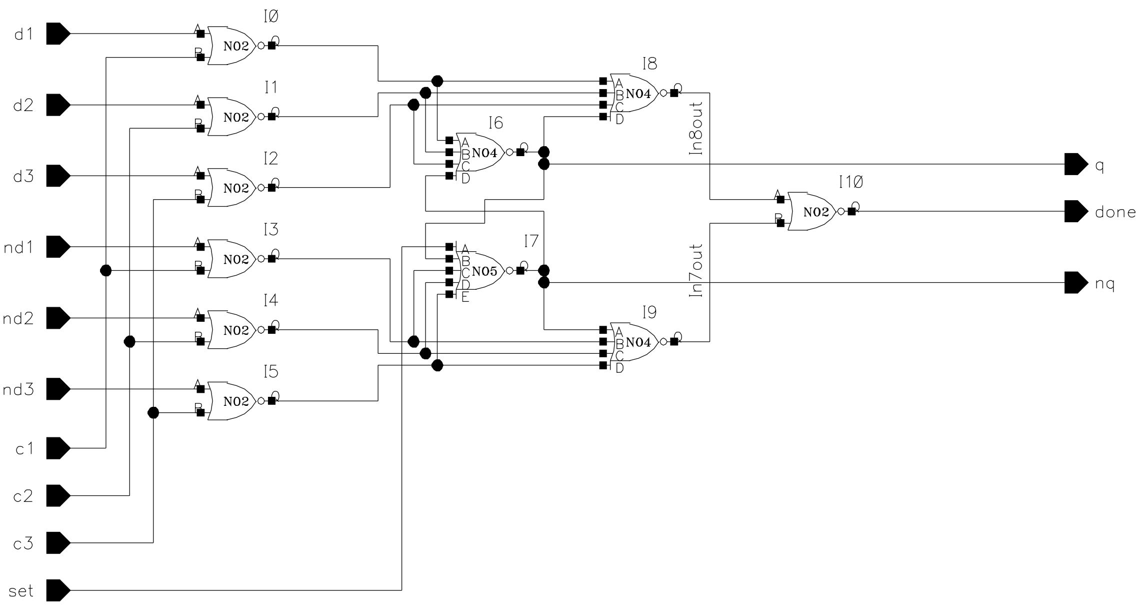


Figure 2.8: Asynchronous Multiplexer

The inputs include 3 pairs of dual rail bits, 3 control bits and a set bit. The asynchronous multiplexer has three stages. The first stage chooses a pair of dual rail bits according to the valid control port. The second stage is an extended NOR RS flip-flop which is used to hold the chosen dual rail bits. The third stage is to generate a completion signal after the flip-flop is set or reset.

The set signal is active high. It sets output *q* and resets *nq*. The control inputs *c1-3* are one-hot active low signals. When none of them are active, the outputs of the first stage are all low. The flip-flop will maintain the current outputs. Once one of them is activated, the selected input pair feeds to the second stage after inversion. They will be stored in the flip-flop and delivered to the outputs. For both *I8* and *I9*, one high input, from either the first stage output or that of the second stage, changes its output to low, and then forces output *done* to high until the control signal is released.

2.3 Asynchronous Communication

Asynchronous communication refers to digital communication (such as between computers) in which there is no timing requirement for transmission.

As stated in [XYCS02], in heterogeneously timed systems, data interfaces may need to be maintained between subsystems not belonging to the same timing domain. The minimal form of this problem is unidirectional data transmission between two single-thread processes.

When the two communicating processes are not synchronised, it is often necessary to pass the data through some intermediate data repository, such as an ACM.

Since Hugo Simpson proposed the concept of Asynchronous data Communication Mechanisms (ACM) [Sim90], it has been studied for decades and has by now developed into a coherent field including classification, specification, and techniques for implementation, analysis and verification. ACMs are potentially useful in systems with heterogeneous timing as data connectors between processes belonging to different timing domains, which may exist either out of necessity or desirability. They can also be useful as digital mimics for various types of data connections in analogue systems, with different types of ACMs suiting different data requirements. The expansion of ACM classification to include several types of ACMs providing more qualitative asynchrony and richer data properties than the traditional FIFO buffer made clear that these applications can be envisaged, and the successful work in synthesis and verification of implementations made them practical.

2.3.1 ACMs and Their Properties

An ACM is used to deliver data items between two processes executing in different timing domains. The provider of the data is called the writer, while the data receiver is known as the reader.

The general scheme of these kinds of data communication mechanisms is shown in Figure 2.9.

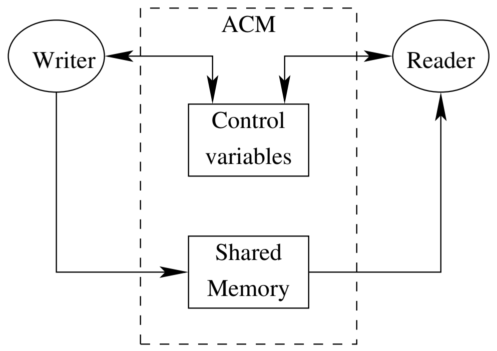


Figure 2.9: ACM with Shared Memory and Control Variables

Most ACM implementations tend to include shared memory, accessible to both writer and reader, for the data being transferred, and control variables, which may be set by one side and read by the other.

A data area capable of holding a single item within shared memory is known as a slot. The different memory designs are named according to the number of slots they contain. This kind of memory mechanism is known as slot-type mechanism.

The most significant properties of ACMs are listed below [XYS+00]:

1. Asynchrony: An ACM should not require the reader and writer processes to be synchronised to each other permanently. If an ACM provides a complete

temporal divide between the reader and writer processes, so that the reader and writer processes are entirely temporally independent from each other, it is said to be fully asynchronous or synchronisation free.

2. Data coherence: Data, in the form of a record with several fields, must always be passed as a coherent set, i.e. interleaved access to any data record by any process is not permitted. Data coherence is violated if at any time both writer and reader access the same data storage location in the shared memory. In the slot-type mechanisms, this means that the writer and the reader should not access the same slot simultaneously. A writer cycle may include a data coherence violation if the writer access conflicts with a reader access at the same slot. The same may be said of a reader cycle. Data coherence is completely maintained if the writer and reader do not access the same slot simultaneously at all.

3. Data freshness: Data freshness describes how up to date any item of data that the reader obtains from the ACM is. In slot-type mechanisms, the latest data item is always found in the slot which the writer accessed during its last cycle. Data freshness is normally checked just before a reader access or at the beginning of a reader cycle. Data freshness is violated if the latest data item written by the writer during its last cycle is not read when the reader accesses the memory.

4. Data sequencing: The reader should obtain data items in the same order to that in which they were written into the ACM by the writer. This property is known as data sequencing. Data sequencing may be violated if the reader obtains the data items in an order other than that written by the writer. This could happen when the reader or the writer accesses the wrong slot.

5. Data loss: Data loss occurs when some items written into the ACM by the

writer are not eventually obtained by the reader. When the writer writes often and the reader reads only occasionally, data loss may happen.

2.3.2 Taxonomies of ACMs

- Lamport's Classification

Lamport [Lam86] classified ACMs into *Safe*, *Regular* and *Atomic* registers.

Safe registers return the value most recently written when a read is not concurrent with a write; when a read is concurrent with a write, they can return any value.

Regular registers return the value most recently written when a read is not concurrent with a write; when a read is concurrent with a write, or series of writes, the read may return the value prior to the writes, the final value written, or any of the values written by the intermediate writes.

Atomic registers return the value most recently written when a read is not concurrent with a write; when a read is concurrent with a write, or series of writes, the behaviour is consistent with them occurring in some serial order.

Lamport also tried to assemble the more useful *Regular* and *Atomic* registers out of directly realisable *Safe* types. Exploiting the fact that the registers transmitting bit type data items can implement to *Atomic* type directly, many ACM protocols have been proposed.[Tro89, KKV87, Sim90]

- Simpson's Classification

Simpson proposed a more general classification system according to the qualitative asynchrony specifications. He divided all ACMs into four types [Sim94]: *Channel, Pool, Signal* and *Constant.*

1. The Destructive Writing mentioned in Table 2.2 means that the writing process can never be held up. The data item in a *Pool* is always destroyed by the

	Destructive Reading	Non-destructive Reading
Destructive Writing	*Signal*	*Pool*
Non-destructive Writing	*Channel*	*Constant*

Table 2.2: Simpson's Classification of ACMs

writer, however, it is not generally the case for a *Signal* where the data item may have already been destroyed by the reader.

2. A Non Destructive Writing means that the writing process may be held up if there is no space for putting new data items.

3. A Destructive Reading means that the reading process may be held up if there is no data in the route waiting to be read.

4. A Non Destructive Reading means that the reading process can never be held up.

The names in Table 2.2 are given by Simpson for demonstrating the reading and writing rules. They are defined as follows [Sim94]:

A *Pool* is characterised by non destructive reading and destructive writing. It allows reference data to be passed from one process to another. The reference data (a single coherent record in conventional programming terms) is retained within the *Pool* where it can be consulted at any time by the reader and refreshed at any time by the writer. Special techniques can be used to maintain the coherence of the data whilst ensuring that there is no temporal interference between writer and reader when the *Pool* is implemented in shared memory. An initial (valid) value, such as valid "0" or valid "1" (refer to Table 2.1), should be loaded in a *Pool* at build time to cater for the situation where the *Pool* is first accessed by the reader before any value has been inserted by the writer. Data-loss property is generally unimportant and cannot be assured in a *Pool*, because of the existence of the destructive writing.

A *Signal* is characterised by destructive reading and destructive writing. It allows event data to be passed from one process to another. Event data (a single coherent record) can be overwritten at any time by the writer, but the data can only be accessed once by the reader. It follows that the data may not be accessed at all if the reader is too slow or if the writer "changes its mind" before the event data has been read. A *Signal* should be initialised to empty (refer to Table 2.1) at build time. The *Signal* is an important communication mechanism in real time systems, as it avoids back propagation of temporal interaction effects (i.e. the actions or inactions of the reader have no direct effect on the timing of the writer). Because of the same reason as a *Pool*, data-loss property is unimportant in a *Signal* either.

A *Channel* is characterised by destructive reading and non destructive writing. It allows message data to be passed from one process to another. Whereas the *Pool* and the *Signal* notionally hold a single coherent record value (from the functional point view), the *Channel* has a capacity and can be used to retain a number of values between processes. Thus complete characterisation must include the capacity of the *Channel*. It is now possible that, at the time of a destructive read, several items will be available for removal. Thus an additional constraint is needed which defines that items are removed from a *Channel* in the order in which they are inserted. A *Channel* should be initialised to empty at build time. Data-loss is a serious failure in a *Channel*.

A *Constant* can be regarded as configuration data. It essentially provides a "write once" capability. Generally the value of a *Constant* is established (written) at build time, and we would not expect to see any real time networks which show a process writing to a *Constant*; hence the use of a restricted form of symbol which indicates no means of connection for a writer.

- New Classification

Simpson's classification has its limitations. For example, there is symmetry between

Pool and *Channel* but no symmetry between *Signal* and *Constant,* and there is not any communication between reader and writer if a *Constant* is used.

To overcome these limitations, a new classification [YXS01, XYCS02] was proposed based on whether Re-reading and Over-writing are permitted or not.

Here, Over-writing means a new data item superseding a previous one when no item in an ACM has been read, while Re-reading means reading a previously read item when no newer one is available. If Re-reading is permitted, the reader will not be held up and if Over-writing is permitted, there will not be holding up in the writer. In contrast, if Re-reading is not allowed, the reader must wait until a newer data item is available. If Over-writing is not allowed, the writer will not proceed until the reader has read the old data item.

	NRR	RR
NOW	*Channel*	*Message*
OW	*Signal*	*Pool*

Table 2.3: New Classification of ACMs

In Table 2.3 RR and NRR stand for Re-reading and Non Re-reading, OW and NOW mean Over-writing and Non Over-writing. Also the names of *Channel, Signal* and *Pool* are inherited from Simpson's Classification in the previous subsection. In this new classification, *Message* is the dual of *Signal* which is defined as follows:

A *Message* is characterised by re-reading and non-overwriting. It allows event data to be passed from one process to another. Event data can be re-read at any time by the reader, but the data is not allowed to be overwritten before it is read. However, rather than the "write once" capability in *Constant*, rewriting is allowed in *Message* when the data has been read by the reader.

2.3.3 Slot-Type Mechanisms

Reader and writer should not access the same slot simultaneously. One, two and three slot mechanisms are developed for conditionally asynchronous communication within a single processor, and the four slot mechanism is for fully asynchronous communication.

- 1-Slot Mechanism: Only one data item is held within the memory. The reader and the writer execute asynchronously until one process receives a request for reading or writing while the other one is accessing the slot.
- 2-Slot Mechanism: Two slots are used so that the writer/reader alternates between the two slots, and the reader/writer accesses whichever of them is not the target of the current write/read (or of the next write/read, if there is none currently active). When re-reading / over-writing are permitted, the problem is that if a read/write overlaps multiple writes/reads, the second write/read will use the slot already being used by the reader/writer. It seems that additional slots are necessary. This problem will not have an effect on a *Channel* because in *Channel*, neither re-reading nor over-writing is permitted.
- 3-Slot Mechanism: If one read overlaps multiple writes, the reader uses one slot, and the writer alternates between the remaining two slots. On the other hand, if one write overlaps multiple reads, the writer occupies one slot, and the reader will wait or re-read the newer data item in the two slots. The problem is that if the statement of updating control variables in the reader cannot be regarded as atomic relative to the combination of the statements of updating control variables in the writer, the read index and the write index may be the same value, which will lead to the reader and the writer accessing the same slot simultaneously, in other words, it violates the coherent property [XC99]. In this case, a Mutex could be used in implementations.

- 4-Slot Mechanism: The 4-slot mechanism arranges the slots as two pairs; the pairs can be called left and right, and within each pair referred to as the top and bottom slots. The writer operates on one pair, alternating between the top and bottom slot. A control bit *latest* indicates to which pair the writer most recently wrote, and another bit *index* associated with the pair indicates which slot of the pair was most recently written; both these control bits are written only by the writer. The reader uses a control bit *reading* to indicate from which pair it is reading; it sets this equal to *latest* when it begins a read; conversely the writer chooses the opposite pair when it starts a write operation. A 4-slot mechanism guarantees the fully asynchronous property even if the access of control variables is not regarded as atomic. For more details please refer to [Sim90].

2.3.4 Some ACM Algorithms

In this subsection, algorithms of 3 type ACMs are introduced. All the three types will be modelled in the following chapters.

2.3.4.1 2-Slot *Channel*

A 2-slot *Channel* is defined as a data buffer which can be used to retain two values between processes. A *Channel* can be empty (with no new item in the memory), half-full (with one new item in the memory) and full (with two new items in the memory). It allows neither re-reading nor overwriting. When the *Channel* is full the writer can not access it and the writer is held up until the *Channel* status is changed. The reader can not perform an access to it when the *Channel* is empty. Only when the *Channel* is half-full may both the reader and the writer access it, but not the same slot simultaneously. At system start-up time, a *Channel* should be initialised to empty.

The algorithm for a 2-slot *Channel* can be synthesised from a Petri nets specification as demonstrated in [YX01], as shown below:

Algorithm 1 Algorithm for a 2-slot *Channel*

```
Writer                                    Reader
wr: wait until $w$!=$r$ then write slot $w$;    r0: wait until $w$==$r$ then $r$=!$r$;
w0: $w$=!$w$                              rd: read slot $r$
```

The statements of *wr* and *rd* are the data accesses, and the others are control variable statements used to determine which slot is to be accessed by the reader and the writer.

2.3.4.2 3-Slot *Signal*

A 3-slot *Signal* is a *Signal* with the capability of holding 3 data items in the memory at a time. The writer points to the slot which is the one neither occupied last by the reader nor accessed by the previous write access. As a result, if the reader occupies one slot for a long time, the writer will access the remaining two slots alternately. This guarantees that the reader always reads the latest available data item in the memory.

The algorithm for a 3-slot *Signal* is also synthesised from a Petri nets specification which is demonstrated in [YX01] as shown below:

Algorithm 2 Algorithm for a 3-Slot *Signal*

```
Writer:                     Reader:
wr: write slot w;           r0: wait until (r!=l) r:=l;
w0: l:=w;                   rd: read slot r;
w1: w=neither (l, r);
```

From the definition of *Signal*, it does not allow re-reading, that is, when there are no new data items in the shared memory, the reader will keep waiting until a new one is available. The *r0* statement plays this role. For the writer, once the new data is stored, *l* is updated in statement *w0*, and *w* is assigned with the index of the slot which is neither being read (r) nor just written (l). This is performed by the *'neither'* function in statement *w1*.

If the statements are considered non-atomic, the 3-slot *Signal* fails in the data coherence and data freshness properties only when at least a consecutive pair of statements *w0* and *w1* are executed between the start and the finish of statement *r0* [XC99]. One way to solve it is to ensure that the *w0* and *w1* pair does not execute simultaneously with *r0*.

Where w is the writer index; r is the reader index; l is the index for the most recent written slot. All these three control variables have the same definition domain with three values. In this case, it can be {0, 1, 2}. The *neither* in *w1* is a function to choose a value from the definition domain other than the two arguments. The function follows the Table 2.4.

l\r	0	1	2
0	1	2	1
1	2	2	0
2	1	0	0

Table 2.4: Table for *neither* Function

2.3.4.3 4-Slot *Pool*

Both the 2-slot and the 3-slot mechanisms have failure modes so far as data coherence is concerned. However, a 4-slot has not under fundamental mode assumptions [Sim90, Cla00]. A 4-slot ACM has 2 pairs of slots in the memory. The pairs can be called left and right, and within each pair referred to as the top and bottom slots. The writer operates on one pair, alternating between the top and bottom slot. A control bit *latest* indicates to which pair the writer most recently wrote, and another bit index associated with the pair indicates which slot of the pair was most recently written; both these control bits are written only by the writer. The reader uses a control bit *reading* to indicate from which pair it is reading; it sets this equal to *latest* when it begins a read; conversely the writer chooses the opposite pair when it starts a write operation. A 4-slot *Pool* is also known as a fully asynchronous communication mechanism because it maintains the coherence of the data whilst ensuring that there is no temporal interference between writer and reader when the *Pool* is implemented in shared memory.

One algorithm for the 4-slot *Pool* has been developed by Simpson [Sim90] nearly 20 years ago. It is shown in Algorithm 3.

Algorithm 3 Algorithm for 4-slot *Pool*

```
Writer:                             Reader:
wr: write slot $d[n,!s[n]]$         r0: $r := l$
w0: $s[n] :=!s[n]$                  r1: $v := s$
w1: $l := n \parallel n :=!r$       rd: read slot $d[r, v[r]]$
```

In this algorithm, all the variables are binary; the use of "$\|$" is for concurrent assignments. The array d indicates two pairs of data storage elements, they are the left pair $d[0,\ 0]$, $d[0,\ 1]$ and the right pair $d[1,\ 0]$, $d[1,\ 1]$. Variable n is used to choose which pair is to be written and r is used to choose the pair for reading. l indicates the most recently written pair. s is an array with two members, $s[0]$ and $s[1]$. Similarly to s, v also has two members $v[0]$ and $v[1]$. Variable $s[n]$ is used to indicate the most recently written slot in the pair (top or bottom), and $v[r]$ is the slot in the pair to be read.

The first writer statement *wr* is the slot access statement. In this statement, the data is written into the slot $!s[n]$ in pair n. The second writer statement *w0* updates $s[n]$, not the whole array s, to its complement which means the row which has been written in *wr*. In the last statement for writer *w1,* the most recently written pair l is updated to n, and then n is assigned a value complement to r. Table 2.5 is the truth table for the control variables before and after statement *w1*.

In the first reader statement, the reader obtains the information about which pair the latest data is in. The second statement tells the reader which slot in the

Before *w1*			After *w1*		
l	*n*	*r*	*l*	*n*	*r*
0	0	0	0	1	0
0	0	1	0	0	1
0	1	0	1	1	0
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	0	0	1
1	1	0	1	1	0
1	1	1	1	0	1

Table 2.5: The Truth Table for *w1*

pair was most recently updated. Combining the information, the reader reads the latest data item.

In this algorithm, the *w1* statement guarantees that the next data item will be written to a different pair from the one which is currently occupied by reader. The *w0* and *r1* statements ensure that even if both the reader and the writer access the same pair, they will occupy different slots.

2.4 Control Systems

A control system is a device or set of devices that manages the behaviour of other devices. It is an interconnection of components connected or related in such a manner as to command, direct, or regulate itself or another system [FPEN06].

The control systems can be classified in different ways. Based on whether persons are involved or not, they can be classified into **manual control** systems and **automatic control** systems. According to whether the system tracks a reference signal, they can be **tracking** systems (or **servos**) or **regulators.** According to the information used in the controlling action, the system can be **open-loop control** (which does not use measures of the system output in the control action), and **close-loop control**, also called **feedback control** (whose output is fed back for use in the control computation). According to other system properties, they can also be classified into **continuous** (the controller is in the continuous time domain) vs. **discrete** (the controller is in the discrete time domain), **linear** (subject to the principle of superposition) vs. **nonlinear** (not subject to the principle of superposition), **time invariant** (whose output does not depend explicitly on time) vs. **time variant** (whose output does depend explicitly on time), and **causal** (depends only on the current and previous inputs) vs. **noncausal** (does not only depend on the current and previous inputs) [FPEN06].

The discrete timed systems include the systems with a single sampling rate and those with multi sampling rates. There is also a distinction for the latter between the systems for which the multi-rate are synchronised and those for which there is no simple relationship between the rates. The first case can be simply implemented with the traditional synchronous manner. Therefore, the communications between processes are managed synchronously. Because there is not synchronisation in it, the second case has to be dealt with by an asynchronous approach - and this is the category to which this thesis is concerned.

2.4.1 Feedback Control System

The block diagram of a feedback control system is shown in Figure 2.10. The process, whose output is to be controlled, is the central component of the system. The actuator is the device that can influence the controlled variable of the process. The central issue with the actuator is its ability to move the process output with adequate speed and range. Generally, the actuator and the process are intimately connected, and this combination is called the plant. The controller is the component that actually computes the desired control signal. Because the controller works on electrical signals, the input reference and the output must be in electrical forms. The input filter plays the role for converting of the reference signal, while the sensor measures and converts the output into electrical forms. The Comparator is to compute the difference between the reference signal and the sensor output to give the controller a measure of the system error.

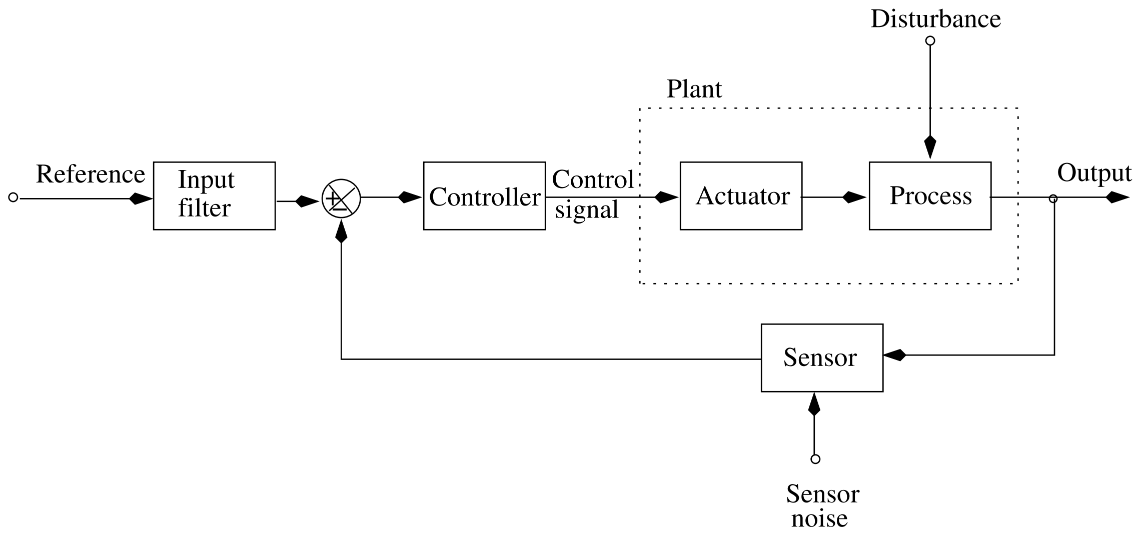


Figure 2.10: Block Diagram of an Elementary Feedback Control System

2.4.2 Dynamic Response

Dynamic response illustrates the performance of a system. It can be studied within the frequency and the time domains.

In the frequency domain, the response (called frequency response) is the measure of any system's response at the output to a signal of varying frequency (but constant amplitude) at its input. The frequency response is typically characterised by the magnitude of the plots of the system's response, measured in dB, and the phase, measured in radians, versus frequency. With taking the logarithm of the radian frequency as abscissa, together these two plots, constitutes a **Bode Plot** [FPEN06].

Time response, as shown in Figure 2.11, the dynamic response in time domain, includes transient response and steady state response.

Transient response is the time response to an initial condition or sudden applied signal, normally a step signal or an impulse signal. If the input signal is a step,

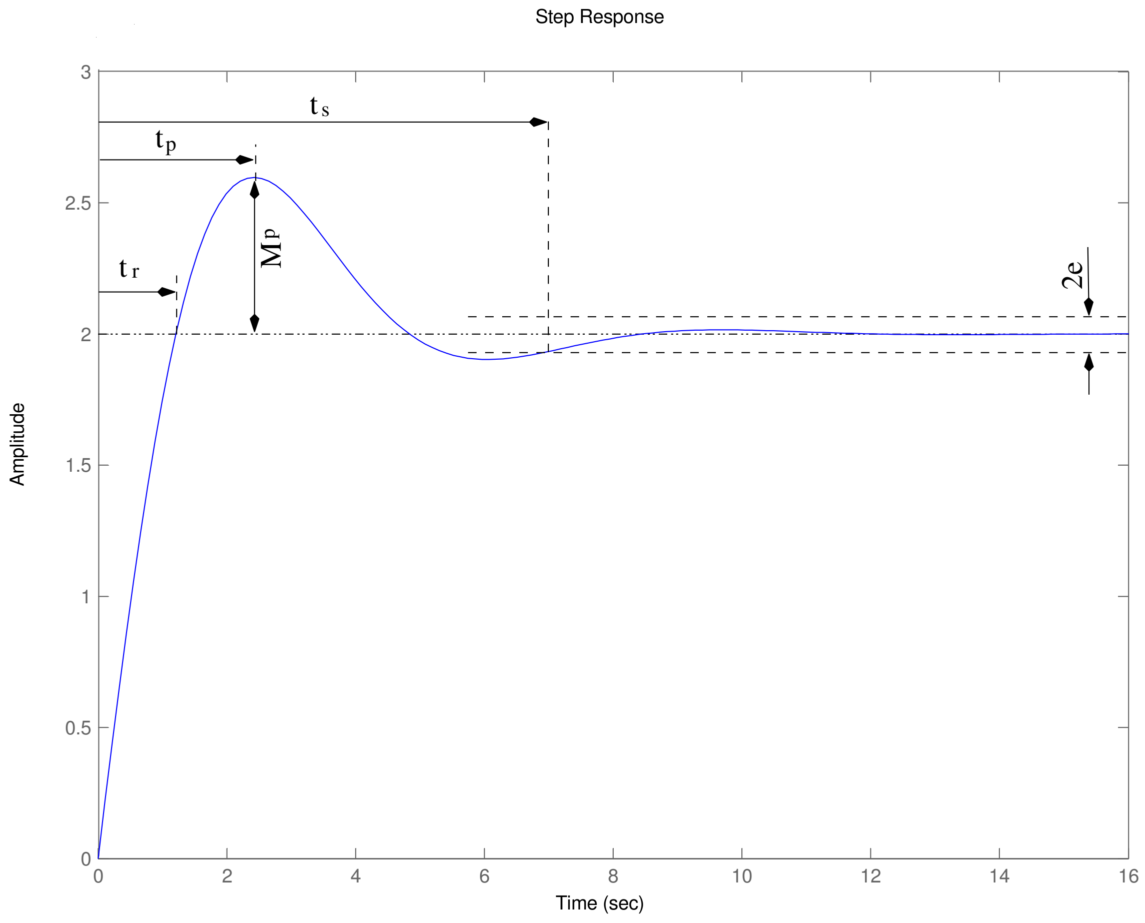


Figure 2.11: Time Response

the system response is called step response. If the input signal is an impulse the response is called impulse response.

The step response is characterised by the rise time t_r (the time it takes the system to reach the vicinity of its new set point), the settling time t_s (the time it takes the system transients to decay), the overshoot M_p (the maximum amount the system overshoots its final value divided by its final value) and the peak time t_p (the time it takes the system to reach the maximum overshoot point).

The steady-state response is the response of a system at equilibrium. The steady-state response does not necessarily mean the response is a fixed value. Steady-state error e (the difference between the desired value and output of the system) is the steady-state response for a step input signal.

2.4.3 Stability Analysis

The study of stability of nonlinear and time-varying systems is complex and often difficult. This section will only introduce the stability analysis of linear time-invariant systems.

A linear time-invariant system is stable if all the roots of the transfer function denominator polynomial have negative real parts and it is unstable otherwise [FPEN06].

2.4.3.1 Neutral Stability

The closed-loop stability can be determined by evaluating the frequency response of the open-loop transfer function and then performing a test on that response.

Suppose the transfer function of a system is:

$$H(s) = \frac{KG(s)}{1 + KG(s)}. \tag{2.4.1}$$

$KG(s)$ is the open-loop transfer function.

A Bode plot of a system that is neutrally stable will satisfy the conditions:

$$|KG(j\omega)| = 1 \tag{2.4.2}$$

and

$$\angle G(j\omega) = 180^o. \tag{2.4.3}$$

The most common situation is: increasing gain leads to instability and $|KG(j\omega)|$ crosses the magnitude $= 1$ once. Therefore, the stability criterion for this case is: $|KG(j\omega)| < 1$ at $\angle G(j\omega) = -180^o$.

However, there are systems for which an increasing gain can lead from instability to stability; in this case, the stability condition is: $|KG(j\omega)| > 1$ at $\angle G(j\omega) =$

-180^o.

2.4.3.2 Routh's Stability Criterion

To avoid the complex computation of the high order polynomial roots, E. J. Routh proposed a criterion for testing the stability of a system, known as Routh's Stability Criterion, in his essay. In this criterion, a Routh array is created from the characteristic equation of the system.

A system is stable if and only if all the elements in the first column of the Routh array are positive.

2.4.3.3 Nyquist Stability Criterion

The Nyquist Stability Criterion is another method for determining stability of a closed loop system. A closed loop system is stable if all of the closed loop poles are in the left half of the s-plane. That is a basic fact about a system. Harry Nyquist showed that it is possible to get information about closed loop pole location by plotting open loop frequency response data.

In order to understand the Nyquist stability criterion it is necessary to understand what a Nyquist plot is. A Nyquist plot is a graph (in complex plane) in which the magnitude and phase of a frequency response are plotted. Where the phase is the angle and the magnitude is the distance from the origin. This plot combines the two types of Bode plot - magnitude and phase - on a single graph, with frequency as a parameter along the curve.

The Nyquist Stability Criterion can be stated as:

- If the open-loop transfer function F(s) is stable, the closed-loop system is unstable for any encirclement of the point -1 in its Nyquist plot.
- If the open-loop transfer function F(s) is unstable, in its Nyquist plot, there must be one clock-wise counter encirclement of -1 for each pole of F(s) in the

right-half of the complex plane.

- The number of surplus encirclements (greater than N+P) is exactly the number of unstable poles of the closed-loop system.

2.4.3.4 Stability Margins

There are two commonly used quantities that measure the stability margin: gain margin and phase margin.

The gain margin is the factor by which the gain can be raised before instability results, while the phase margin is the amount by which the phase of $G(j\omega)$ exceeds -180^o when $|KG(j\omega)| = 1$.

2.4.4 PID Control

PID controller is widely used in the process and robotics industries as the control of steady-state error. A classical PID controller contains three terms: proportional (**P** term), integral (**I** term) and differential (**D** term) control. The control equation is:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)dt + K_d \frac{de(t)}{dt} \tag{2.4.4}$$

where $e(t)$ is the system error.

The **P** term means that the feedback control signal is linearly proportional to the system error. It can be used to control the constant term and the natural frequency. The increase of the gain K_p will reduce the rise time and steady state error, but the damping is also reduced which may be too low for satisfactory transient response and may result in oscillation.

The **I** term is proportional to the integral of the error, which has a major influence on the steady state error. The integral term will not stop changing until its input is zero, and therefore if the system reaches a stable steady state, the input signal to the integrator will of necessity be zero.

The **D** term is proportional to the derivative of the system error, which affects suddenly changing signals and compensates the reduction of the damping caused by the P term.

2.4.5 Brushless DC Motor Control System

Brushes are used to conduct current between stationary wires and moving parts, most commonly in a rotating shaft in a DC motor. As the brushes are slowly abraded the life of a DC motor using brushes could be significantly shorter than one without brushes. A brushless DC Motor is a DC motor that does not contain brushes [Bru02]. Figure 2.12 shows the basic structure of this kind of control system found in [KKAJ90]. This system consists of a mechanical part (the motor), an electrical part (the motor's drive sub-system) and an electronic part (the integrated circuit controller) Cascade control is usually used for this drive, with an inner loop controlling motor current or torque and an outer loop controlling motor speed [Bla96]. Both the speed and current/torque controllers were integrated into the same ASIC in [KKAJ90]. The speed and current control laws are implemented digitally as PID controllers.

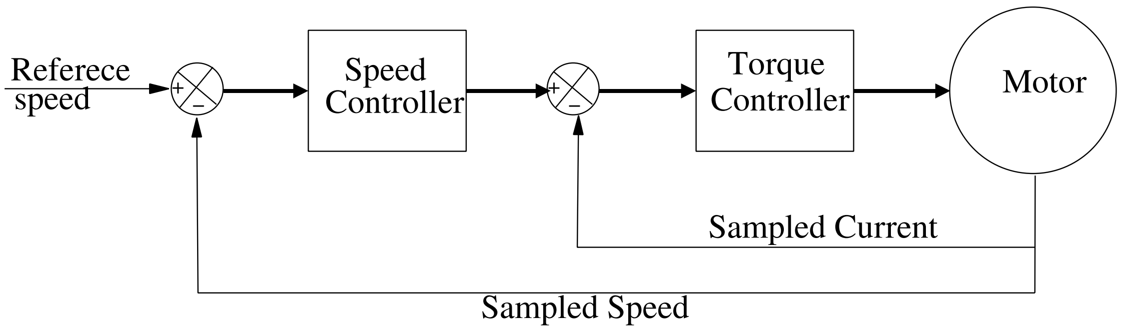


Figure 2.12: Schematic of Motor Control System

This is how a brushless DC motor works. A set of sensors on the motor's shaft sense the position when the motor is rotating. According to the position information, the actual speed is calculated. By comparing to the reference speed, a speed error is worked out, which feeds to the speed controller. A reference current is produced by the speed controller based on the speed error. According to the error between the actual and the reference current, the amount of drive required is calculated by the torque controller. Applying the calculated amount to a Pulse Width Modulator (PWM), the drive voltage is worked out [KAJ91]. Pulse Width Modulation is a technique employed to regulate the output power by changing the pulse width. In a digital system, the integral number of steps related to the clock frequency is used as the pulse width.

2.5 Model and Implementation Tools

2.5.1 Petri Net (PN)

Petri nets could be used to describe asynchronous systems [VM96a]. According to the Petri net specifications, hardware could be directly implemented by translation [VM96b, YK98, YKKL94, WB99, SBY03, Sha03]. As a useful modelling tool for asynchronous systems, Petri nets will be briefly introduced in this section.

The Petri nets were firstly introduced by Carl Petri in the 1960s [Pet62, Pet73]. They provide a simple graphical description of a state-transition system with an easy representation of concurrent events or a choice between alternative events. Furthermore, the set of reachable states can be obtained from a PN using a straightforward algorithm.

2.5.1.1 Definition of Petri Nets

The basic elements of a classical Petri net include places, transitions, tokens (or markings) and arcs. The graphical representations for these elements are illustrated in Table 2.6.

Place here refers to a container that one could put data into. Transition refers to

Elements	Representations
Place	○
Transition	|
Token	•
Arc	↘

Table 2.6: Basic Elements in Petri Nets

an entity that modifies the system's state. Token is a virtual object that is passed between places to communicate. Only the device with the token may communicate, to avoid clashing with other devices. Arc is assigned a direction which shows the way the tokens pass.

The formal definition can be found in [DJ01, Pet81, Sok06]:

Definition 2.1. A Petri net (PN) is formally defined as a tuple PN = $<P$, T, F, $M_0>$, where

P is a set of places.

T is a set of transitions.

F is a set of arcs known as a flow relation. It is subject to the constraint that no arc connects two places or two transitions, or more formally: $F \subseteq (P \times T) \cup (T \times P)$.

$M_0 : S \rightarrow \mathbb{N}$ known as an initial marking, where for each place $s \in S$, there are $n \in N$ tokens.

There is an arc between $x \in P \cup T$ and $y \in P \cup T$ iff $(x, y) \in F$. An arc from a place to a transition is called consuming arc, and that from a transition to a place is known as producing arc. The preset of a node $x \in P \cup T$ is defined as $\bullet x = \{y \mid (y, x) \in F\}$, and the postset as $x\bullet = \{y \mid (x, y) \in F\}$.

The dynamic behaviour of a PN is defined as a token game. It changes markings according to the enabling and firing rules of its transitions. A marking is a mapping $M : P \rightarrow \mathbb{N}$ denoting the number of tokens in each place, $\mathbb{N}$={0,1} for 1-safe PNs. A transition t is enabled iff $M(p) > 0$, $\forall p \in \bullet t$. The evolution of a PN is possible by firing the enabled transitions. Firing of a transition t results in a new marking M' such that $\forall p \in P : M'(p) = \begin{cases} M(p) - 1 & if \quad p \in \bullet t, \\ M(p) + 1 & if \quad p \in t\bullet, \\ M(p) & otherwise \end{cases}$, that is, for an enabled transition t one token is removed from each preset place and one token is produced to each postset place.

There are three important properties of a PN: safeness, liveness and deadlock-freeness. A PN is said to be k-bounded if the number of tokens in every place of a reachable marking does not exceed a finite number k. A 1-bounded PN is also called 1-safe. A PN is deadlock-free if, no matter what marking has been reached, it is possible to fire at least one transition of the net. A PN is live if for every reachable marking M and every transition t it is possible to reach a marking M' that enables t.

2.5.1.2 Basic Connections in Petri nets

There are five basic connections in Petri nets. They are linear, join, fork, merge (also called exclusive join) and choice (exclusive fork) shown in Figure 2.13.

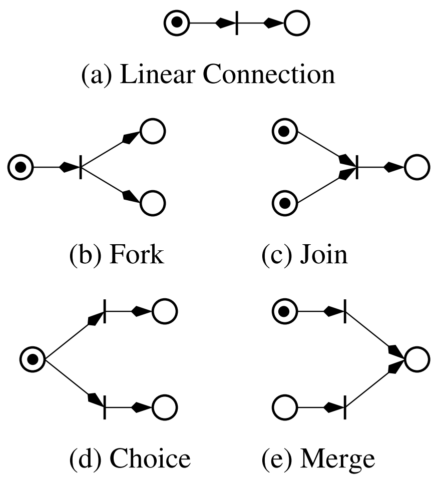


Figure 2.13: Basic Connections in Petri net

A linear connection, case (a) in Figure 2.13, means that a place has only one output transition whose only input place is this place, or a transition has only one output place whose only input transition is this transition. Linear connection is the most commonly used connection in a Petri net. It indicates that a satisfied condition triggers an action, or an event leads to one state. All other connections can be called non-linear connections.

A fork connection, (b) in Figure 2.13, is when a transition has two or more output places and one input place. Once the transition in the fork is fired, it puts tokens in all of its output places. This implies that the event leads to two or more places.

A join connection, (c) in Figure 2.13, is when a transition has two or more input places and one output place. Only when all the input places hold a token does the transition fire. A join connection implies that it needs to satisfy several conditions to trigger one action.

A choice connection, (d) in Figure 2.13, is when a place has two or more output transitions and one input transition. When the place in the choice holds a token, only one of its output transitions can be fired. Sometimes the output transitions of a choice may have read arcs (will be defined later) or other input places attached. The choice represents the fact that one condition is able to trigger several actions, but only one is triggered according to other conditions.

A merge connection, (e) in Figure 2.13, is when a place has two or more input transitions and one output transition. Firing any of the input transitions can add a token in the place. It follows a first come first served rule. However, in a 1-safe net, it must ensure that only one transition fires at a time.

Fork and join imply that at least two branches are executed at the same time. Choice and merge imply the existence of choice because only one branch is executed at a time.

Besides the five basic connections stated in Figure 2.13, there is another useful connection called "read arc". A read arc is the arc between a transition and one of its input places, the token in which will not be consumed when the transition is

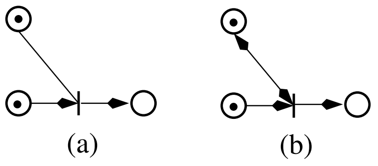


Figure 2.14: Read Arc

fired. It has two different notations. It can be a line without arrows, as shown in Figure 2.14 (a). It can also be a line with arrows in both sides, as shown in Figure 2.14 (b), which means firing the transition will consume the token in the place, but a token will be put back to the place afterwards.

2.5.2 MATLAB

MATLAB is a highly versatile language for technical computing. The name stands for Matrix Laboratory. It integrates computation, visualisation, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. It is an interactive system whose basic data element is an array that does not require dimensioning. This allows the user to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar non-interactive language such as C or FORTRAN.

MATLAB, as software in a synchronous platform, is not able to represent and simulate the truly asynchronous behaviours. However, as the effective sampling intervals are short enough, it is assumed that the events can happen at any time. It will only fail to represent the real world when events happen near simultaneously, which will cause metastability in real life. However, on one hand, the possibility of the occurrence of this occasion is very low, so the simulation results from MATLAB are reliable most of the time. On the other hand, the reason for using MATLAB is to study the asynchronous behaviour in an application system level instead of trying to see what happens if there is metastability in the ACMs at this level of analysis.

2.5.2.1 Simulink

Simulink is a software package for modelling, simulating, and analysing dynamic systems. It supports linear and nonlinear systems, modelled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

For modelling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors.

After a model is defined, it can be simulated, using a choice of integration methods, either from the Simulink menus or by entering commands in the MATLAB Command Window. Using scopes and other display blocks, the simulation results can be seen while the simulation is running. The simulation results can be also put in the MATLAB workspace for postprocessing and visualisation.

Useful Simulink Blocks in Simulink

A **memory block**, Figure 2.15, outputs its input from the previous time step, applying a one integration step sample-and-hold to its input signal.

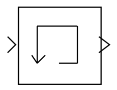


Figure 2.15: Memory Block

In the modelling, the memory block also plays a role of preventing ambiguous execution order in a loop.

A **switch block**, shown in Figure 2.16, passes through the first (top) input or the third (bottom) input based on the value of the second (middle) input. The first and third inputs are called data inputs. The second input is called the control input.

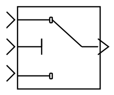
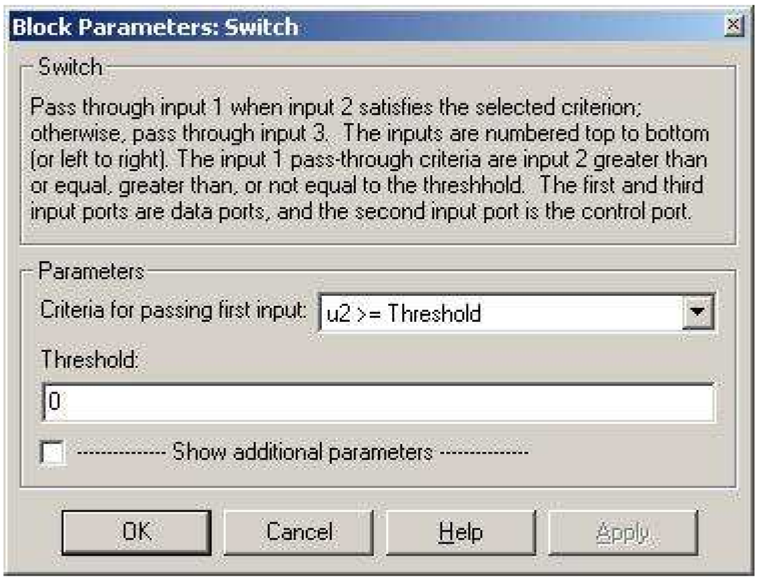


Figure 2.16: Normal Switch Block



Switch

Pass through input 1 when input 2 satisfies the selected criterion; otherwise, pass through input 3. The inputs are numbered top to bottom (or left to right). The input 1 pass-through criteria are input 2 greater than or equal, greater than, or not equal to the threshhold. The first and third input ports are data ports, and the second input port is the control port.

Parameters

Criteria for passing first input: u2 >= Threshold

Threshold:

0

------------- Show additional parameters -------------

Figure 2.17: Parameter Box for Switch

The conditions are defined under which the first input is passed with the **Criteria for passing first input** parameter. The block can be made to check whether the control input is greater than or equal to the threshold value, purely greater than the threshold value, or nonzero. If the control input meets the condition set in the **Criteria for passing first input** parameter, then the first input is passed. Otherwise, the third input is passed. (See Figure 2.17) The threshold value is fed into the control input.

A **normal switch block** only contains two data inputs. In many cases, there is a need for providing a data link among more than two data inputs. A multi-port switch block is designed for these models. Figure 2.18 shows a Multi-Port Switch block. It chooses from a number of inputs. The first (top) input is called the control input, while the rest of the inputs are called data inputs. The value of the control input determines which data input is passed through to the output port. If the control input is an integer value, then the specified data input is passed through to the output. If the control input is 1, then the first data input is passed through to the output. If the control input is 2, then the second data input is passed through to the output, and so on. If the control input is not an integer value, the block first truncates the value to an integer by rounding down. If the truncated control input is less than 1 or greater than the number of input ports, an out-of-bounds error is returned.

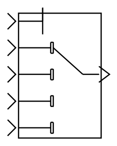


Figure 2.18: Multi-Port Switch Block

The number of data inputs can be specified with the **Number of Input Ports** parameter.

In modelling the data path, the switches are used to choose routes or hold data items. The switches are atomic so the memories are used to simulate the delays within the data path.

The normal switch can be used with binary control input, and the multi-port switch block may accept a natural number control input.

2.5.2.2 Stateflow

Stateflow is a software package in MATLAB. It is a graphical design and development tool for control and supervisory logic used in conjunction with Simulink. It supports statecharts formalism which was introduced in 1987 by Harel [Har87]. It can be used to:

1. Visually model and simulate complex reactive systems based on finite state machine theory

2. Design and develop deterministic, supervisory control systems

3. Easily modify the design, evaluate the results, and verify the system's behaviour at any stage of the design

4. Automatically generate integer, floating-point, or fixed-point code directly from the design

5. Take advantage of the integration with the MATLAB and Simulink environments to model, simulate, and analyse the system

Stateflow provides clear, concise descriptions of complex system behaviour using finite state machine theory, flow diagram notations, and state-transition diagrams all in the same Stateflow diagram. Stateflow brings system specification and design closer together. It is easy to create designs, consider various scenarios, and iterate until the Stateflow diagram models the desired behaviour.

Flow diagram notation creates decision-making logic such as for loops and if-then-else constructs without the use of states. In some cases, using flow diagram notation provides a closer representation of the required system logic that avoids the use of unnecessary states.

2.5.2.3 Notation of Stateflow

A Stateflow diagram is composed with the symbolic objects of Stateflow notation. There are 3 different types of available Stateflow objects: graphical objects, non-graphical objects and data dictionary.

Graphical objects, drawn in the Stateflow diagram editor, are shown in Table 2.7

Nongraphical objects, including event and data objects, do not have graphical representations in the Stateflow diagram editor. However, they can be seen in the Stateflow Explorer.

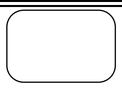
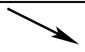
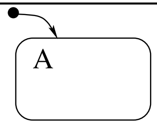
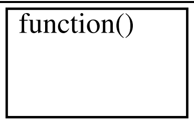
Name	Notation
State	
Transition	
Default transition	
Connective junction	
Graphical function	

Table 2.7: Notation of Graphical Objects in Stateflow

Data dictionary is a database containing all the information about the graphical and nongraphical objects. Data dictionary entries for graphical objects are created automatically as the objects are added and labelled. Nongraphical objects are explicitly defined in the data dictionary by using the Explorer. The parser evaluates entries and relationships between entries in the data dictionary to verify that the notation is correct.

1. State

   State describes a mode of a reactive Stateflow chart. When a state is active, the chart takes on that mode. When a state is inactive, the chart is not in that mode. The activity or inactivity of a chart's states dynamically changes based on events and conditions. The occurrence of events drives the execution of the Stateflow diagram by making states become active or inactive. At any point in the execution of a Stateflow diagram, there is a combination of active and inactive states.

   The label for a state appears on the top left corner of the state rectangle with the following general format:

```
name/

entry(en):entry action

during(du):during action

exit(ex):exit action

on event_name:on event_name action
```

A state label starts with the name of the state followed by optional / character and additional lines with keyword prefixes. Each keyword prefix identifies a different type of action associated with the state, and is optional and positionally independent.

Every state (and chart) has a decomposition that indicates what kind of substates it can contain. All substates of a superstate must be of the same type as the superstate's decomposition. Decomposition for a state can be exclusive (OR) or parallel (AND).

Exclusive (OR) state decomposition for a superstate (or chart) is indicated when its substates have solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time. The children of exclusive (OR) decomposition parents are OR states.

The children of parallel (AND) decomposition parents are parallel (AND) states. Parallel (AND) state decomposition for a superstate (or chart) is indicated when its substates have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are always active at the same time.

2. Transition

Transition is a curved line with an arrowhead that links one graphical object to another. In most cases, a transition represents the passage of the system from one mode (state) object to another. A transition is attached to a source and a destination object. The source object is where the transition begins and the destination object is where the transition ends.

A transition is characterised by its label. The label can consist of an event, a condition, a condition action, and/or a transition action. Transition labels have the following general format:

event[condition]{condition_action}/transition_action

Event: the specified event is what causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. Absence of an event indicates that the transition is taken upon the occurrence of any event.

Condition: a condition is a Boolean expression to specify that a transition occurs given that the specified expression is true. The condition is enclosed in square brackets ([]).

Condition Action: a condition action follows the condition for a transition and is enclosed in curly braces ({}). It is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid. If no condition is specified, an implied condition evaluates to true and the condition action is executed.

Transition Action: the transition action is executed after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined to be valid. The transition action is preceded with a forward slash.

3. Default Transition

Default transitions are primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighbouring exclusive (OR) states. They have a destination but no source object.

In some circumstances, a label can be marked on default transitions as other transitions. For example, it can be specified that one state or another should become active depending upon the event that has occurred. In another situation, specific actions take place depending upon the destination of the transition.

4. Connective Junction

   A connective junction represents a decision point between alternate transition paths taken for a single transition. Connective junctions are used to help represent the following:

   - Variations of an if-then-else decision construct, by specifying conditions on some or all of the outgoing transitions from the connective junction
   - A self-loop transition back to the source state if none of the outgoing transitions is valid
   - Variations of a for loop construct, by having a self-loop transition from the connective junction back to itself
   - Transitions from a common source to multiple destinations
   - Transitions from multiple sources to a common destination
   - Transitions from a source to a destination based on common events

5. Graphical Functions

   A graphical function is a function defined graphically by a flow graph that provides convenience and power to Stateflow action language. The following example in Figure 2.19 shows a graphical function side by side in a Stateflow diagram with the transition that calls it:

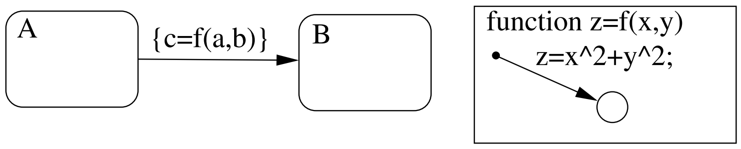


Figure 2.19: Graphical Function Example

In this example the function z = f(x,y) is called in the condition action of the transition from state A to state B. The function is defined using symbols that are valid only within the function itself. The function is called using data objects available to states A and B.

Graphical functions are similar to textual functions such as MATLAB and C functions. Like C and MATLAB functions, graphical functions can accept arguments and return results. Like C and MATLAB functions, graphical functions can be invoked in transition and state actions. Unlike C and MATLAB functions, however, graphical functions are full-fledged Stateflow graphical objects.

6. Event

   An event is a Stateflow object that can trigger a whole Stateflow chart or individual actions in a chart. Because Stateflow charts execute by reacting to events, events are specified and programmed into the charts to control their execution. Events can be broadcast to every object in the scope of the object sending the event, or can be sent to a specific object.

7. Data

   A Stateflow chart stores and retrieves data that it uses to control its execution. Stateflow data resides in its own workspace, but data residing externally in the Simulink model or application that embeds the Stateflow machine can be also accessed. When creating a Stateflow model, any internal or external data used in the action language of a Stateflow chart must be defined.

A data object has the following properties:

Name: The name of the data item.

Scope: Scope specifies where it resides in memory relative to its parent. It can be:

- Local: it resides and is accessible only in a machine, chart, or state.
- Input from Simulink: it is accessible in a Simulink Chart block but resides in another Simulink block that might or might not be a Chart block. The receiving Chart block reads the value of the data item from an input port associated with the data item.
- Output to Simulink: it resides in a Chart block and is accessible in another block that might or might not be a Chart block. The Chart block outputs the value of the data to an output port associated with the data item.

Type: Data type of the data, including double, single, int32, boolean etc.

Port: Index of the port associated with the data item. This control applies only to input and output data.

Limit Range: This control group specifies values used by a Stateflow machine to check the validity of this data item. It includes the next two controls.

- Min. Minimum value that this data item can have during execution or simulation of the Stateflow machine it belongs to.
- Max. Maximum value that this data item can have during execution or simulation of the Stateflow machine it belongs to.

8. Stateflow Explorer

The Stateflow Explorer displays any defined event or data objects within an object hierarchy where machines, charts, states, boxes, and graphic functions are potential parents. The Stateflow Explorer can be used to create, modify, and delete event and data objects.

2.5.2.4 Semantics of Stateflow

1. Executing a Chart

   A Stateflow chart executes when it is triggered by an event from Simulink. A chart is inactive when it is first triggered by an event from the Simulink model and has no active states within it. After the chart executes and completely processes its initial trigger event from the Simulink model, it exits to the model and goes to sleep, but still remains active. A sleeping chart has active states within it, but no events to process. When Simulink triggers the chart the next time, it is an active but sleeping chart.

2. Executing a State

   A state is entered (becomes active) in one of the following ways:

   (a) Its boundaries are crossed by an incoming executed transition.

   (b) Its boundary terminates the arrow end of an incoming transition.

   (c) It is the parallel state child of an activated state.

   State entry action, if specified, is performed when it becomes active. The state is marked active before its entry action is executed and completed.

   Active states that receive an event that does not result in an exit from that state, execute a *during* action to completion if a *during* action is specified for that state. An *on event_name* action executes to completion when the event specified, event_name, occurs and that state is active. An active state executes its during action and on event_name action before processing any of its children's valid transitions. During and on event_name actions are processed based on their order of appearance in the state label.

A state is exited (becomes inactive) in one of the following ways:

(a) Its boundary is the origin of an outgoing executed transition.

(b) Its boundary is crossed by an outgoing executed transition.

(c) It is a parallel state child of an activated state.

A state performs its exit action, if specified, before it becomes inactive. The state is marked inactive after the exit action has executed and completed.

3. Executing a Transition

   Transitions have sources and destinations. A transition is only triggered when all the following conditions are satisfied:

   (a) The source state is active;

   (b) The transition events, if specified, have happened;

   (c) The specified conditions are true.

   The transition action is executed after the transition destination has been determined to be valid.

2.6 Conclusions

In this chapter, a brief introduction was given on asynchronous designs, asynchronous communications, especially ACMs, control systems and some modelling tools, such as Petri nets and Stateflow in MATLAB.

In this thesis, asynchronous communications are the core, Stateflow is the major tool, and control systems are the applications. The knowledge of asynchronous designs is the fundamental requirement of a 3-slot *Signal* implementation in Chapter 3. Some ACMs are specified in Petri nets. With the knowledge of Petri nets, the reader will understand how an ACM works.

Chapter 3

Petri Net to Stateflow Conversion

3.1 Introduction

As stated before, according to the Petri net specifications, hardware could be directly implemented by translation [VM96b, YK98, YKKL94, WB99, SBY03, Sha03]. To model the asynchronous systems specified in Petri nets in the MATLAB environment, a technique similar to direct translation can be studied. Fortunately, the software package Stateflow in MATLAB works quite similarly to Petri nets. It is the straightforward way to convert a Petri net model to a Stateflow one. It should be notice that a similar work was proposed in [Esh05] about transforming a Petri net model to Statechart. In this work, every place is simply converted into a basic state. This may cause problems when joins exist (will be discussed in next section).

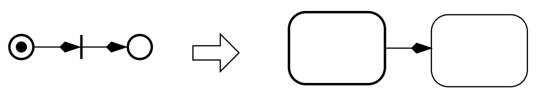
	Petri net	Stateflow
State	○	▢
Transition	|	↘

Table 3.1: Basic Components in Petri net and Stateflow

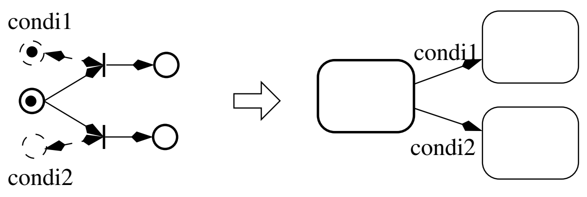
Petri nets and Stateflow models have many similarities. Both models represent transitions between states. They have similar basic components as seen in Table 3.1.

3.2 Conversion for the Basic Connections

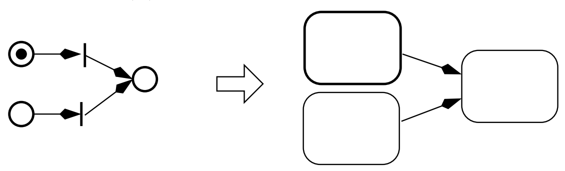
Converting a complex Petri net to a Stateflow model is based on the conversions for PN connections stated above. The basic idea of the conversion is to replace the places and transitions in a Petri net with the states and transitions in Stateflow. This idea works perfectly with the linear and exclusive connection as illustrated in Figure 3.1. As stated before, the choice may not exist alone, other places may be attached in the output transitions. The dashed lines pointing to dashed circles in merge are two other places which represent the conditions for firing the corresponding transitions. The conditions are transferred to the transition conditions in the Stateflow diagram.



(a) Linear Connection



(b) Choice Connection



(c) Merge Connection

Figure 3.1: Basic Conversions: linear and Exclusive Connections

Here, the states and transitions in the Stateflow are equivalent to the places and the transitions in the PN. A place holding a token can be represented by an active state, it appears as a rounded rectangle with thicker border. The token moving from one place to the other is represented by the active state being transferred from one state to the other. Executing a transition in Stateflow can represent firing a transition in PN.

Figure 3.1 (a) shows the translation of the linear connection. The first place holds a token, which enables the transition. After the transition is fired, the token is transferred into the second place. The corresponding Stateflow model represents that the first state is active initially, which leads the execution of the transition. After the transition is executed, the active state moves from the first state to the second one. Here, an active state represents a place holding a token.

Figure 3.1 (b) shows the translation of a conditional choice connection, that is, the choice depending on the tokens in other places in the system. If the place *condi1* (*condi2*) holds a token, in other words, if the condition *condi1* (*condi2*) is satisfied, the upper (lower) transition will be fired. Therefore, condition transitions of Stateflow are used to represent the transitions in PN choice. It has to be mentioned that if the place *condi1* and *condi2* hold a token at the same time or do not exist, it leads to a conflict. Both of the transitions are enabled, but only one can be fired. The resulting conflict may be resolved in a purely non-deterministic way or in a probabilistic way, by assigning appropriate probabilities to the conflicting transitions. All these could be programmed in the transition conditions.

Figure 3.1 (c) shows the translation of a merge connection. In the PN model, when the upper (lower) place holds a token, the upper (lower) transition is fired. As a 1-safe net, upper and lower places could not hold token at the same time. The Stateflow model works in the same way. When the upper (lower) state is active, the upper (lower) transition is executed. Because of the One active state rule (discussed in next section), upper and lower states could not be active at the same time.

Forks and Joins cannot be converted by direct replacement. The reasons are explained as follow. The default setting for the Stateflow is Exclusive (OR). According to the **One Active State Rule** (will be explained in next section), only one state can be active in an Exclusive chart or state. Forks and Joins always imply that there are more than one places holding token, which means that more than one states are active in Stateflow models. Obviously, this breaks the **One Active State Rule**. To dealing with this problem, a state with parallel decomposition is added. Figure 3.2 shows a Fork model in Stateflow. The transition activates the AND state which contains two or more parallel states. The composition setting for these parallel states is Exclusive, because the states inside are still active in series. The default transitions point to the first states, and would be activated at once when the parallel states are activated.

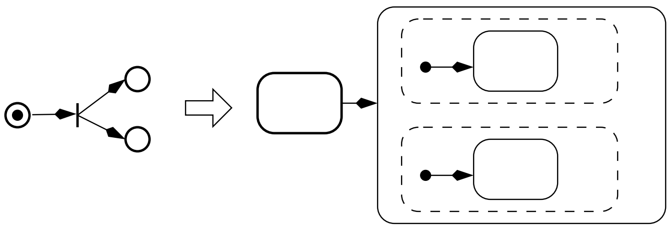


Figure 3.2: Basic Conversions: Fork Connection

A join model in Stateflow is illustrated in Figure 3.3. The places in the join are converted to one state in each parallel state. The transition starts from one of these states (for example *a* in the Figure) and ends at a state outside the parallel states, and the transition condition is that all the other states are active. Whether one state is active (or not) can be checked by a flag variable which is set in the entry action and reset in the exit action.

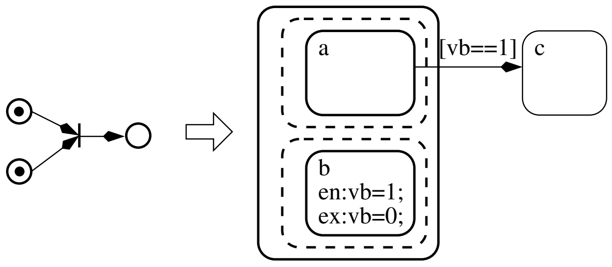


Figure 3.3: Basic Conversions: Join Connection

According to the **One Active State Rule**, Figure 3.3 implies that once the transition is executed, both *a* and *b* are inactive. As a result, all the tokens in current marking should be consumed and one token should be put in the successive place. However, in many models, this is not always the case. Such as in Figure 3.4, firing *t2* only consumes the tokens in p1 and *p2*, but not that in *p3*. That is, firing *t2* dose not result in consuming all the tokens in the marking. Therefore, Figure 3.3 cannot handle this case. It must be refined.

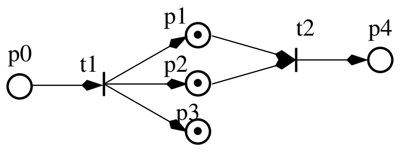


Figure 3.4: An Example of a Transition Not Consuming All the Tokens

The simplest way is to move state *c* in Figure 3.2 into the parallel state where state *a* resides. According to the Petri net principle, once a transition is fired, the tokens in all its input places are removed, that is, all the input states are deactivated. However, the active state *c* in the parallel state is not able to deactivate state *b* in the other parallel state. To deactivate state *b*, a complementary state *c_ comp* is added, as shown in Figure 3.5. A flag variable is also added in state *c*. When *c* is active, the transition between *b* and *c_ comp* is executed, and state *b* is deactivated. In [Esh05], as shown in Figure 3.6, because no complementary states are added, the state *b* remains active after the transition from state *a* to *c* is executed. Reflecting back to the Petri net model, this means that the token in one place has not been consumed after the transition was fired. If this place is referenced by any other nets, it will produce a fault.

Figure 3.7 is the reachabilities for the Petri nets model, Eshuis' method and the method in this thesis. The three or four digits represent state (place) *a*, *b*, *c* (, *c_ comp*). It can be seen that Eshuis' join is not equivalent to the Petri nets model. Leaving the redundant state *c_ comp* out of consideration, the model in the thesis

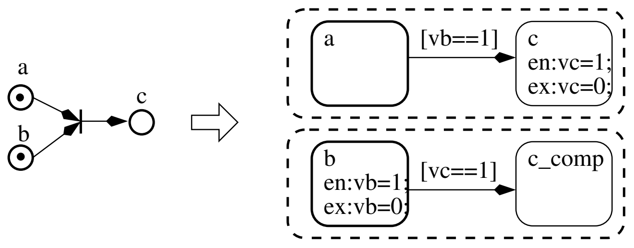


Figure 3.5: Basic Conversions: Join Connection Refinement

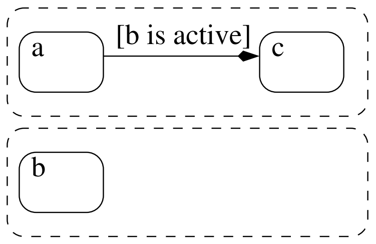


Figure 3.6: Join Connection According to Eshuis

is equivalent to the PN model.

Petri nets	110	$\longrightarrow$	001
Eshuis' Method	110	$\longrightarrow$	011
Method in this Thesis	1100	$\longrightarrow$	0011

Figure 3.7: Reachabilities for the Join

Based on Figure 3.5, the net in Figure 3.4 is modelled in Figure 3.8. Figure 3.9 shows the reachabilities for Figure 3.4 and Figure 3.8. The digits represent *p0* to *p4* and *p4_comp*. Leaving the redundant state *p4_comp* out of consideration, the Stateflow model is equivalent to the PN model.

Figure 3.3 is suitable for the case that a join transition consumes all the tokens in the marking, and Figure 3.5 is appropriate for the case that a join transition consumes some of the tokens in the marking.

The transformation of a read arc is similar to that of a join. The only difference is that after firing, the token in the place will not be consumed. In the Stateflow model, the state keeps active. Because concurrency is still maintained and the read

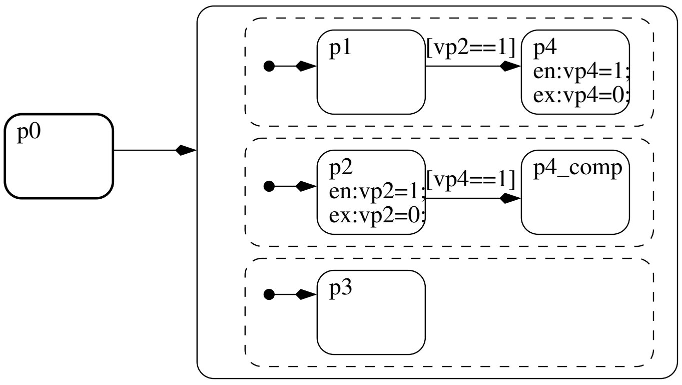


Figure 3.8: Stateflow Model for Figure 3.4

$$\begin{tikzcd}
\text{Petri nets} & 10000 \arrow[r] & 01110 \arrow[r] & 00011 \\
\text{Stateflow} & 100000 \arrow[r] & 011100 \arrow[r] & 000111
\end{tikzcd}$$

Figure 3.9: Reachabilities for Figure 3.4 and Figure 3.8

state is still active, the model of the read arc can be modified from Figure 3.3 by moving state c into the parallel state, as illustrated in Figure 3.10.

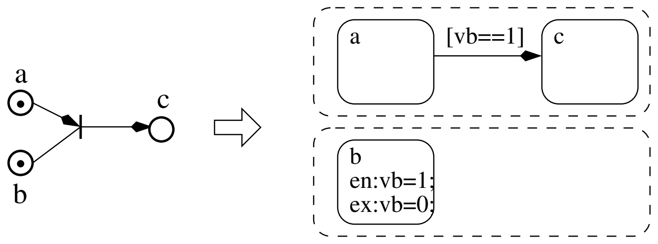


Figure 3.10: Basic Conversions: Read Arc

3.3 One Active State Rule

The basic connections can be modelled perfectly. However, some complex connections may not be modelled correctly. This section will show such kinds of examples.

Case 1: A fork followed by a merge as shown in the left of Figure 3.11 can not be modelled as the Stateflow diagram shown in the right in the figure. A token in place a will lead two tokens in place d. This is not a 1-safe net any more. In the Stateflow diagram, any exit from descendant states in *P* to state *d* will result in an exit from state *P*, which will deactivate all the descendant states in state *P*. This is not equivalent to the Petri nets shown in the figure.

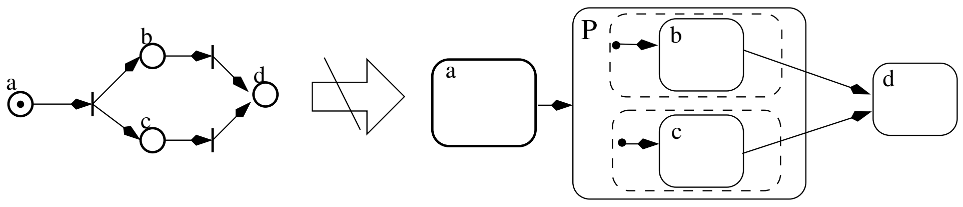


Figure 3.11: A Fork Followed by a Merge

Case 2: A choice followed by a join as shown in the left of Figure 3.12 can not be modelled as the diagram shown in the right of the figure either. Only when both places *b* and *c* hold tokens, transition t1 is enabled. Referring to the Stateflow diagram, only when both states *b* and *c* are active, will state *d* be activated. However, it is not allowed for two or more states to be active at the same time in the sequential domain. Consequently, the diagram in the figure is not a valid Stateflow chart.

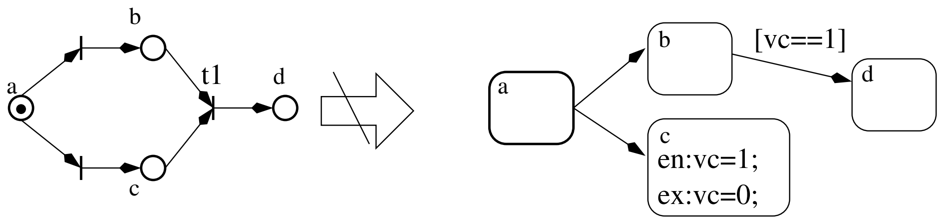


Figure 3.12: A Choice Followed by a Join

Case 3: The net shown in Figure 3.13 cannot be modelled into Stateflow either. To enable *t1*, both places *a* and *c* should hold a token. This means that in the Stateflow model, state *d* will not be activated until both state *a* and *c* are active. Because state *c* is a descendant of state *P*, an active *c* will lead to an active *P*. As a result, both the states *a* and *P* are active simultaneously in the sequential domain, which is not allowed. Therefore, the Stateflow chart in this figure is also invalid.

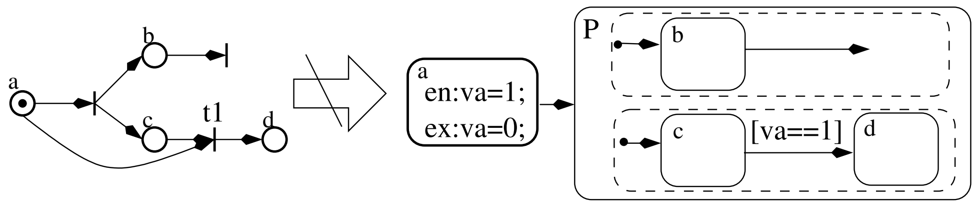


Figure 3.13: An Choice Followed by a Fork

Case 4: The Stateflow can not model the net shown in Figure 3.14 either. In this net, if *t2* is fired, it may lead to one token in place *c* and one in place *d*. This will map to the Stateflow case that both the states *c* and *d* are active, which is not allowed in Stateflow. In the diagram itself, the arc from state a to state *c* will deactivate the state *P* and all its descendants. Therefore, this Stateflow diagram is neither equivalent nor valid.

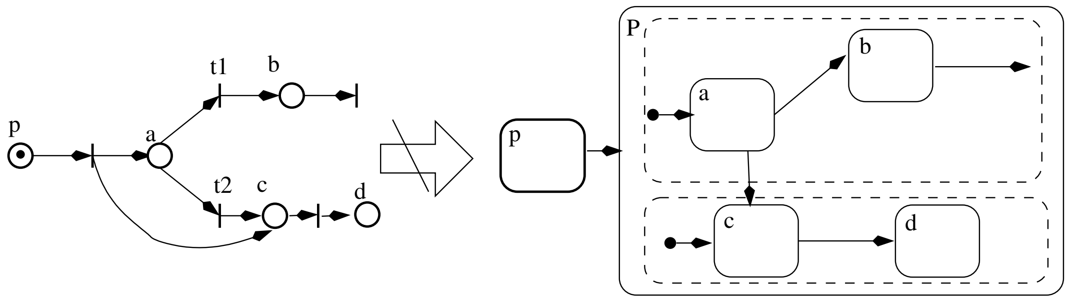


Figure 3.14: A Fork Followed by a Choice

In an active Stateflow chart, there is one and only one active state within each sequential domain. This can be called a **One Active State Rule**. These four cases are non-safe nets. All the non-safe nets will result in a place holding more than one token. In Stateflow charts, it means an active state is activated by other states in the same sequential domain or several active states in the same sequential domain are trying to active one state simultaneously. Both of the scenarios break the **One Active State Rule**. Therefore, non-safe nets are not suitable for this kind of conversion.

3.4 An Example for Conversion

In this section, Figure 3.15 will be taken as an example to illustrate the conversion from well-formed Petri nets [DE95] model to a Stateflow. In this example, all the basic connections are included.

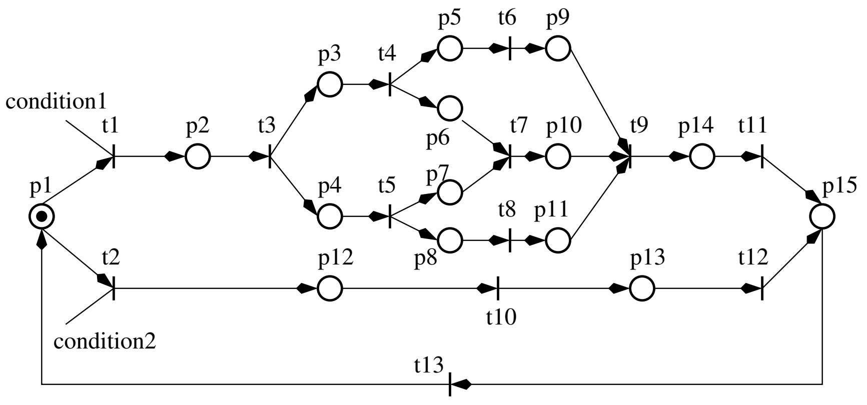


Figure 3.15: Petri net Example

The first step of converting a Petri net is to select a suitable starting place. This place must be the only place holding a token in a reachable marking. This implies a limitation that the Petri net model must have at least one reachable marking in which only one place holds a token. Due to this limitation, a pipeline model has to be converted with other approaches, which will be proposed in next section. In the example in Figure 3.15, *p1* was chosen as the starting place.

Starting from the chosen place, all the linear connections, choices and merges are converted by replacing them with the corresponding models in Figure 3.1. The part, which starts with forks and all their branches end at joins, are treated as single states firstly. Figure 3.15 was converted to Figure 3.16 following these statements. State *C1* represented the part starting from *t3* and ending at *t9* in the net. At this stage, the Stateflow graph does not contain any concurrency.

The branches of the fork are modelled as parallel states within state *C1*. If there are more forks in the branches, they can be modelled in the same way. In the ex-

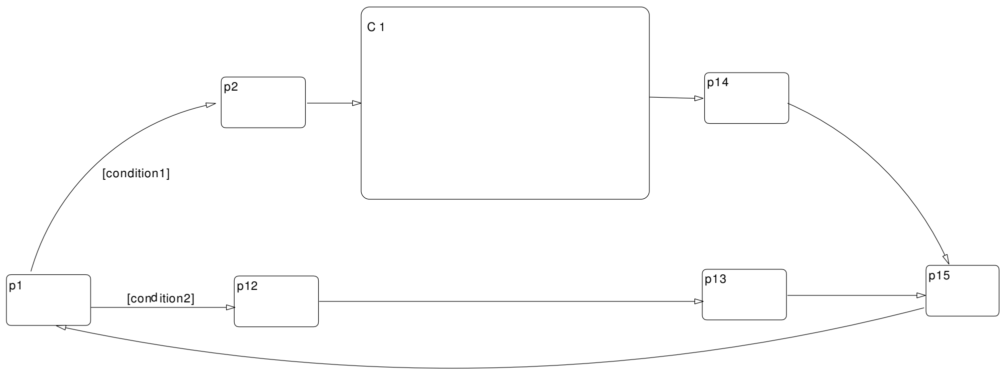


Figure 3.16: Petri net Conversion - Step1

ample, after the fork from *t3*, there are two concurrent branches. Each branch is modelled within one of two parallel states. The forks from *t4* and *t5* can also be modelled in the same way. Because *t4* and *t5* belong to parallel states themselves, they have to be converted as parallel substates within their parents' ones as illustrated in Figure 3.17. In this figure, state *C11* and *C12* contained the two branches after transition *t3*. As place *p3* and *p4* were marked after *t3* was fired, state *p3* and *p4* would be the first active states in their branches. Thus, the default transitions in the branches were directed to them. States *C2* and *C3* contain the concurrent parts after transitions *t4* and *t5*. Their concurrent branches were in states *C21*, *C22*, *C31* and *C32*. If any state within its parent state is active, the parent state is also active; if any state within its parent state transits to an outside state, all the states in its parent are deactivated. For example, if state *p9* is active, state *C21*, *C2*, *C11* and *C1* are all active. If any of the states inside *C1* transits to state *p14*, *C1* and all the states inside are deactivated.

In the Stateflow, only the states which are able to be active concurrently may be joined. If after the join, the concurrency still exists, the join is converted as Figure 3.5, that is, the joined states are put into one of the concurrent branches, and a compensating state is added in all the other branches leading to the join. If there is no concurrency after the join, it is modelled as a transition with conditions as

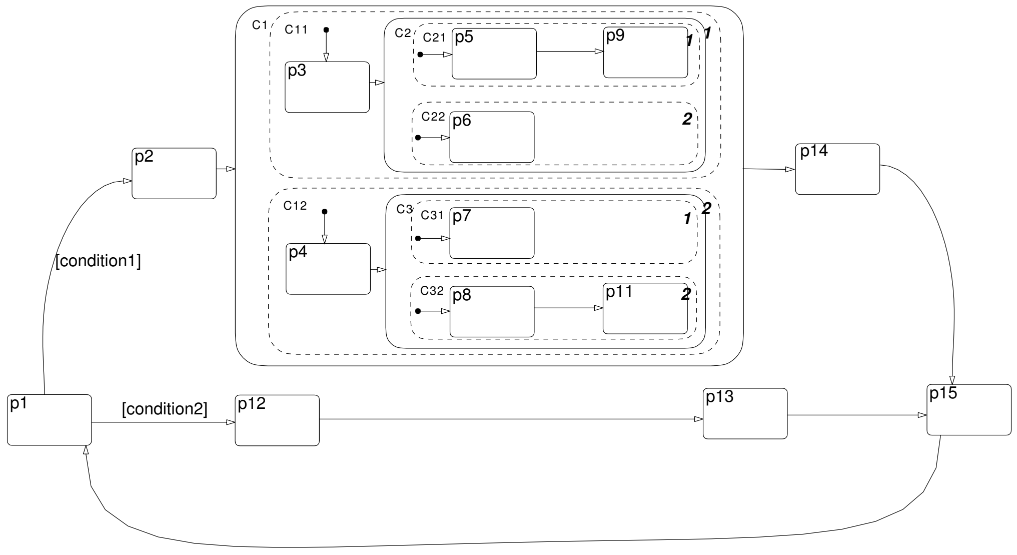


Figure 3.17: Petri net Conversion - Step2

Figure 3.3. Figure 3.18 illustrates the conversion that *p6* and *p7* were joined at *t7* and output to *p10* in Figure 3.15. *t7* was modelled as a transition between the states *p6* and *p10*. Because the states in *C21* and *C32* were still concurrent with *p10*, *p10* was put into *C22* and a compensating state was added in *C31*. As after the join *t9*, there was no more concurrency, the join was modelled as a transition from *p9* to *p14* with the condition that *p10* and *p11* were active.

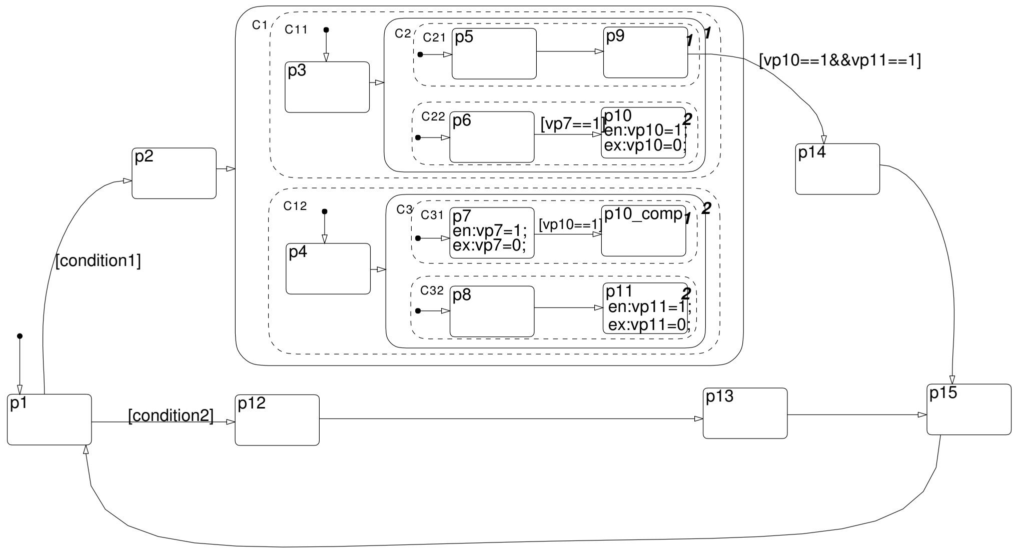


Figure 3.18: Petri net Conversion - Step3

The final step is modelling initial markings. They can be modelled as a state with a default transition attached. One and only one state in the sequence domain is active at a time, therefore, there must be one and only one unconditional default transition in each sequential domain. Consequently only the initial markings which maintain the **One Active State Rule** can be modelled in Stateflow. In addition, if a state in the parallel state is initially active, all its compensating states and parent states must be marked as active states as well. If the initial active states in the parallel branches are not the same as the ones which the default transitions point to, they have to be attached with conditional default transitions. In this example, the initial marked place *p1* maintained the rule. However, if one of the initial markings is *p9*, it makes *C2* active, and to maintain the rule, one of the states in *C22* must be active initially. For the same reason, one of the states in *C12* must be active initially. Figure 3.19 illustrates an example for the case that *p9* is marked initially.

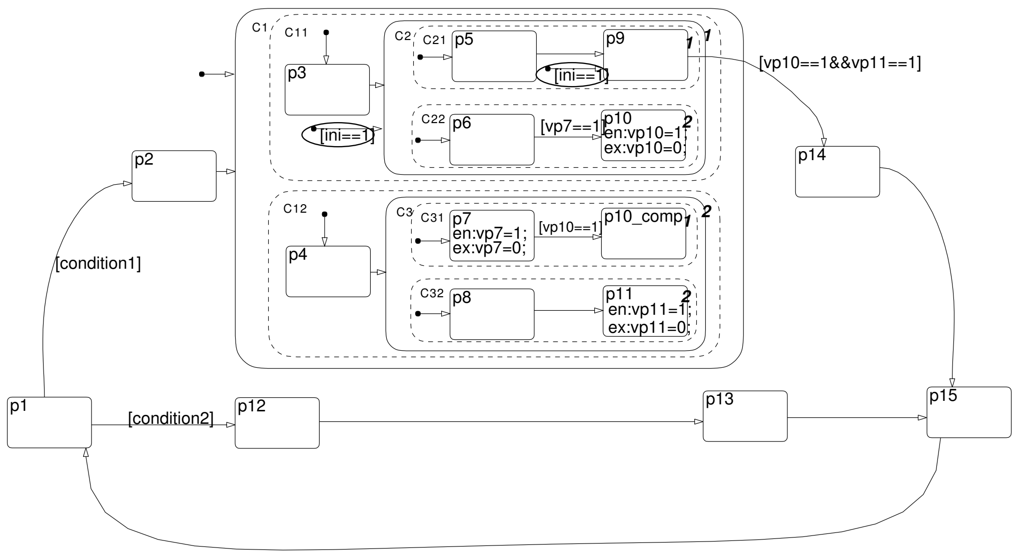


Figure 3.19: If p9 is marked initially

The Petri nets shown in Figure 3.15 are typical well-formed nets containing most of the possible connections that may appear in well-formed nets. A successful transformation of the nets shown in Figure 3.15 implies that this method could be used in the transformation of well-formed nets.

3.5 Conversion of a Pipeline

In the previous section, the way of conversion of a simple case of well-formed PN models is introduced. However, the method is not suitable for a pipeline conversion. The reason could be discovered by analysing a PN model of a pipeline as shown in Figure 3.20. Figure 3.20 is a PN model of a four-stage pipeline. Each stage contains two places, which represent "full" and "empty" states, and two transitions, the input (left hand) and the output (right hand). The adjacent stages share one transition. The two places on the left hand side of the pipeline represent the input handshake with the environment, and the two places on the right indicate the output handshake with the environment. Obviously, there must be one and only one place holding a token in each stage, that is, at any time, the marking of a four-stage pipeline model contains four places that hold tokens. It is impossible to find such a starting place as stated in previous section. Therefore, a new method should be investigated for converting a pipeline model.

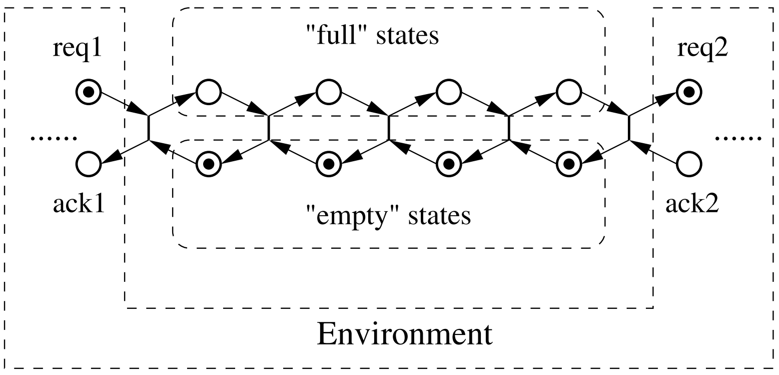


Figure 3.20: PN Model of a Pipeline

It is necessary to fully understand how the PN model works before starting the conversion. This is how it works. If the first stage of the pipeline is empty and there is an input request, the input transition is fired. As a result, this stage becomes full and sends an acknowledgement back to finish the handshake. If the last stage of the pipeline is full and there is an output acknowledgement, the output transition is fired. As a result, this stage becomes empty and sends a request back to finish the handshake. The stages in the middle work in the same manner with treating the previous stage as the input and the following as the output.

As each stage of a pipeline holds one and only one token, an n-stage pipeline contains n places which hold tokens. Because of the **One Active State Rule**, the n stages are converted into n concurrent states. The conversion is started with a single stage.

According to the basic conversions, a single stage of the pipeline is converted in Figure 3.21.

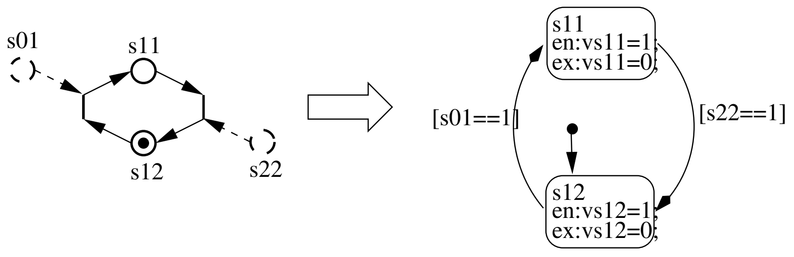


Figure 3.21: Conversion of a Single Stage

If only one stage is considered, anything outside that stage is treated as the environment. The discussions here only focus on how the environment affects the stage. The effect on the environment can be regarded as that within other stages. Therefore, Figure 3.21 only contains the input request *s01* and output acknowledgement *s21*. Places *s12* and *s01* join at *s11*, and places *s11* and *s22* join at *s12*. These two joins are converted using the basic join conversion. Because the initial place in the PN model is *s12*, the default transition in the Stateflow model points to the state *s12*.

The basic idea of building an n-stage pipeline is putting n stages together. Figure

3.22 shows the scenario of combining two stages. If we name the right hand transition of the first stage as *t12*, and the left hand transition of the second stage as *t21*, it could be seen that *t12* and *t21* represent the same transition in the PN model. They should be executed simultaneously. However, in Figure 3.22, *t12* and *t21* will never be executed at the same time. Furthermore, one of them could not be executed, because the execution of the other one will fail the transition condition.

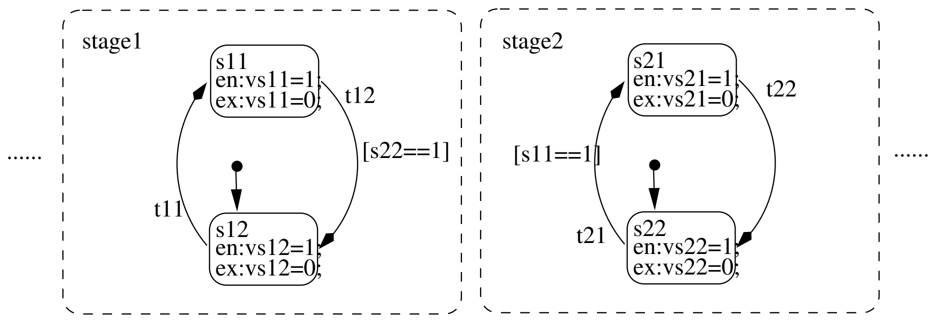


Figure 3.22: Combining Two Stages

Coming back to the PN model, it can be seen that: to fire a transition, it must be enabled: the two input places hold tokens. After the transition is fired, it becomes disabled: the two output places hold tokens. That is, firing occurs between the times when the transition is enabled and disabled.

Enabled and disabled could be treated as two states of the transition in the PN model. If it can be ensured that both *t12* and *t21* are completed between these two states, and the intermediate states would not affect other executions, the executions of *t12* and *t21* could be treated as being simultaneous. Although some internal behaviour was added to the Stateflow model, it is still observation-equivalent (or weakly bisimilar) [Mil89] to the PN one.

Figure 3.23 gives a solution of stages with a refined transition. The initial state of the transition is disabled. Only when both the states *s11* and *s22* are active, *trans1* becomes enabled, and only when both the states *s12* and *s21* are active, *trans1* becomes disabled. The transition condition of *t12* and *t21* changed to *[vt1==1]*

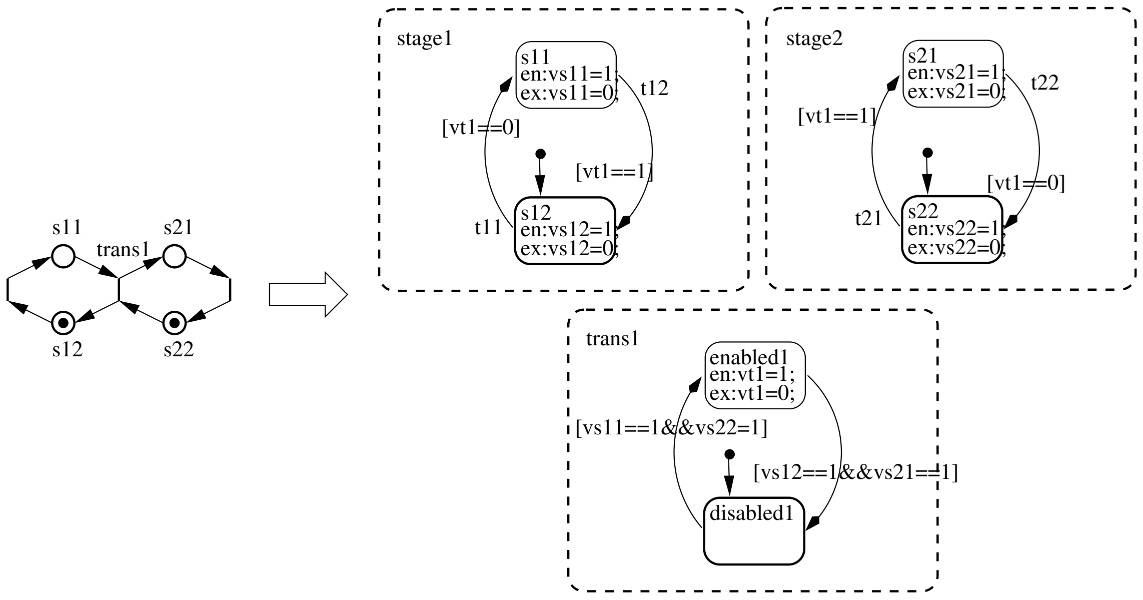


Figure 3.23: Stages with Refined Transition

which means that *trans1* only fires when it is enabled. This guarantees both *t12* and *t21* occur between enabled and disabled states. The transition conditions of *t11* and *t22* become *[vt1==0]* which means that the stages would not respond until the trans1 is disabled. This ensures that the intermediate states will not affect other executions.

The 4-stage pipeline in Figure 3.20 is converted into the Stateflow model as shown in Figure 3.24 by combining the model in Figure 3.23 together. In the figure, the transitions in the *stage* states have two conditions rather than one. The reason for this is that a stage is not only adjacent with one side but also with the other side. The effects from both sides should be counted at the same time.

3.6 Conclusions

ACMs are normally described by Petri nets. The successful conversion from Petri nets to Stateflow makes it possible to build ACM models in MATLAB. The conversion is based on the similarities between Petri nets and Stateflow, such as places and states, transitions, place holding a token and active state. With these similarities, the well-formed Petri net models can be easily translated into Stateflow.

There are also differences between Well-formed Petri nets and Stateflow, such as in PNs several places may hold a token but in Stateflow there is a **One Active State Rule**. Because of this, the conversion of a pipeline becomes difficult. In this case, the transitions in PN have to be modelled as two states, one of which represents the transitions which are enabled and the other indicates when they are disabled.

Although the conversion was still manual, the success of converting linear connection automatically in [Zho03] gives us hope of fully automatic conversions.

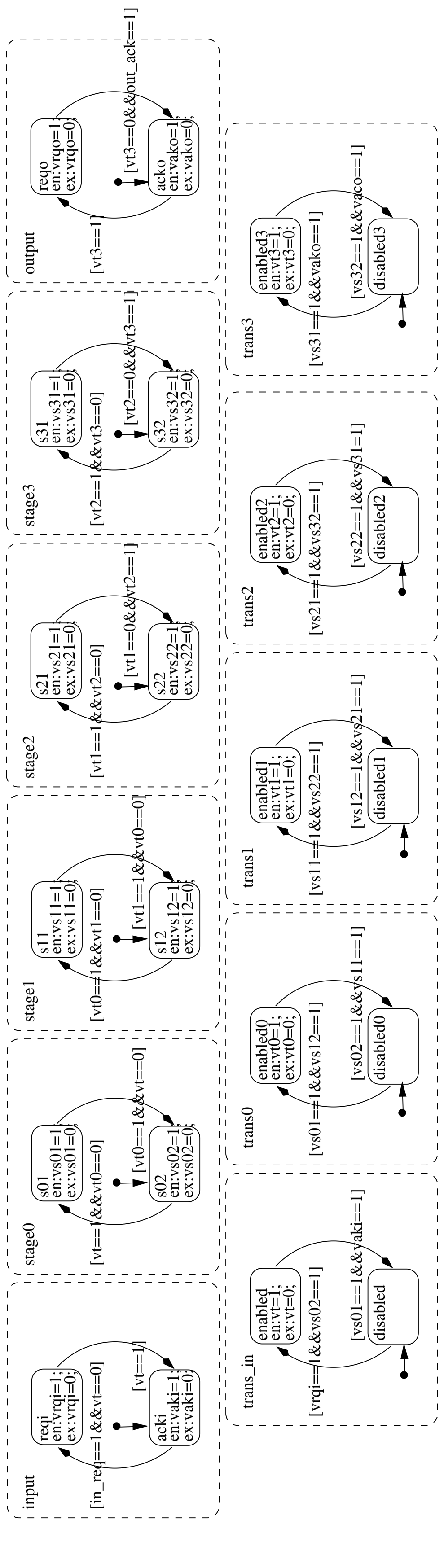


Figure 3.24: Stateflow Model of a 4-Stage Pipeline

Chapter 4

ACM Models in MATLAB

4.1 Introduction

In this chapter, MATLAB models of ACMs will be investigated. As one of the most important protocols in asynchronous designs, a handshake model is firstly built so that it could be used in ACM modelling.

Behaviour of ACMs can be specified by Petri nets. Because there is a method, described in the previous chapter, to translate Petri nets models into Stateflow charts, it is a straightforward idea to build ACM from its Petri nets specification. A 3-slot Signal will be used as an example to show how an ACM is modelled from its Petri nets specification.

The Stateflow model of ACM translated from Petri nets representation is more like a structure model, which is not necessary when investigating the applications. Function blocks showing ACM properties can be used instead. These blocks can be modelled directly from their algorithms, which make the models much simpler and easier to be understood. 2-slot *Channel*, 3-slot *Signal* and 4-slot *Pool* are used as examples to show how the different types of control paths (*Channel*, *Signal* and *Pool*) and data paths (2-slot, 3 or more slots) are modelled. As a *Message* is regarded as the duel of a *Signal*, it is not discussed in this chapter.

In general, the data being transferred consists of a stream of items, instead of only one, of the same type, hence sometimes buffering in the ACM is useful. This chapter also describes an investigation of ACMs which contain buffers. The writer and the reader processes are also assumed to be single-thread loops, during each cycle of which a single item of data is transferred to or from the ACM.

Hardware implementation of a 3-slot *Signal* is carried out to show that the models have corresponding circuits, which have the same properties.

4.2 Handshake Model

As mentioned in Section 2.2.3, handshake protocols are important protocols in asynchronous designs, and so are they in ACM modelling. Therefore, handshake models in MATLAB need to be investigated. In this section, a 4-phase handshake protocol is modelled so that it could be applied to ACM modelling.

A handshake is the interface protocol between two processes. As a result, it is modelled in both the sender and the receiver. The sender is the active component in the handshake, because it is sending the requests, and, therefore, the receiver response to the request is the passive component. Taking the four phase handshake as an example, in the sender, when a piece of information is ready, the protocol follows this order: sending a request, waiting for the acknowledgement sent from the other side, releasing the request. The sender is active during the period from the information being ready till the request being released. When triggered by the completion signal of the receiving process, the receiver becomes active. The handshake protocol on this side follows this order: sending an acknowledgement when receiving is completed, waiting for the release of the request, resetting the acknowledgement. After this cycle, the receiver becomes inactive.

No matter whether it is sender or receiver, the order of the process is: triggered by an event, performing the protocol, becoming inactive. The order of the protocol is: sending a signal, waiting for the response from the other side, releasing the signal.

A handshake can be well represented in Stateflow by a state which will become active when triggered by an event, and the protocol can be perfectly modelled by actions and conditions.

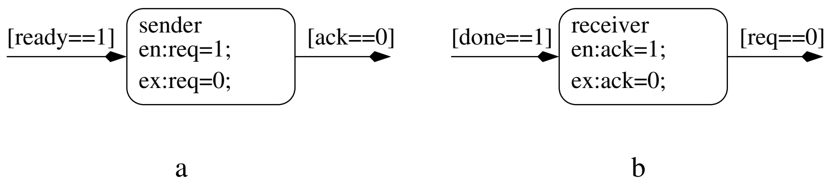


Figure 4.1: Handshake Protocol in Stateflow (a: Sender Side, b:Receiver Side)

Figure 4.1 shows the handshake protocol in Stateflow. Here, *req* and *ack* stand for request and acknowledgement.

One handshake cycle in the sender is represented in the following way: when transition condition *ready* is satisfied (information is ready for delivery), the sender state becomes active; at the same time, a request is generated in the state entry actions; the state itself represents waiting for the acknowledgement in the transition conditions, which lead to the exit from the state (end of waiting) and execution of the transitions; on exiting a state, the request is released; the state comes back to inactive.

One handshake cycle on the receiver side is represented as: when the transition condition *done* is satisfied, i.e. the requested operation has been done, the receiver state becomes active; at the same time, an acknowledgement is generated in the state entry action; the state itself represents waiting for the release of the request in the transition conditions, which leads to the exit from the state (end of waiting) and execution of the transitions; on exiting a state, the acknowledgement is released; the state comes back to inactive.

Please note that the request is not delivered to the receiver state directly but to the receiving process instead.

4.3 MATLAB Model for a 3-Slot *Signal*

ACM can be specified with Petri nets. The MATLAB model of ACM in MATLAB could be translated from the Petri nets specifications by using the transformation method stated in Chapter 3. The Petri nets representation of a 3-slot *Signal* is generated below.

4.3.1 Petri nets Model

The algorithm of the 3-slot *Signal* is using variables to abstract the slot indices. However, in the real implementation all the slots need to be considered. Consequently, the Petri nets representing the behaviour of the reader and the writer need to be generated with the slot indices applied in.

There are three write cycles in the writer. Each one deals with a write process for a particular slot. The algorithm for the writer is applied to each cycle. To make the *neither* function in *w1* statement realisable, it is defined as shown in Table 4.2.

l==1		l==2		l==3	
r==1 or 2 (r!=3)	r==3	r==2 or 3 (r!=1)	r==1	r==1 or 3 (r!=2)	r==2
w=3	**w=2**	**w=1**	**w=3**	**w=2**	**w=1**

Table 4.2: *neither* Function

Assume that the writer points to the first slot and the reader points the second one initially.

In the writer, it is idle in the first slot. A place named *"idle1"*, as shown in Figure 4.2, holding a token is able to represent this initial state. A *write_start* signal could trigger the writer to execute according to the Algorithm 2 in Chapter 2. This can be represented by a *write_start* place joining with the *idle1* place. The output place can be called *start1*. Because the trigger does not change the *write_start* signal, the *write_start* place is a reference. A read arc is sufficient to play this role. The first step of the algorithm is to write (statement *wr*). Since it is writing the first slot, a transition *wr1* could represent this event. When writing is completed, which is represented by moving the token to a place called *"wr1_done"*, the next event according to the algorithm is updating l (statement *w0*) and w (statement *w1*). To maintain the data coherence and the data freshness properties when the statements are considered as non-atomic, *w0* and *w1* need to be treated as a consecutive pair, and it does not execute simultaneously with *r0* [XC99]. Therefore, a transition can be used for updating both l and w. Since the new w partially depends on r, and various r may result in two different w, a choice with two branches follows place *wr1_done*. According to Table 4.2, when r=3, w is updated to 2, otherwise, it is updated to 3. In the Petri nets, the transition called *"w12"* with a read arc from the place *r=3* represents the first case, where *w12* means updating w from 1 to 2, which implies l is updated to 1. The output place is called *"w=2"* indicating that current w is 2. Another event informs the environment, the completion of the write cycle and sends the writer back to the idle state. In the Petri nets, this event is modelled as a fork following the place *w=2* with two output places, one of which is in the environment and the other is *idle2*. The other branch of the choice following place *wr1_done* could be modelled in the same way.

The small net at the top right corner in Figure 4.3 indicates an environment. The dashed line is a read arc pointing to the main net. The dotted arcs are from the last transition in the write cycle in the main net. The initial marking for this net is in the *!write_start* state. When a *write_start* signal comes, the *start* transition is fired, and the token goes to the *write_start* state, which triggers the process in the main net. Once the write process is complete, a token is obtained from the place *w=2*. The *done* transition is fired, and the token goes back to the *!write_start* state.

The remaining part of the Petri nets, including the cases that the writer points to the second and the third slot, can be generated in the same way, as shown in Figure 4.3.

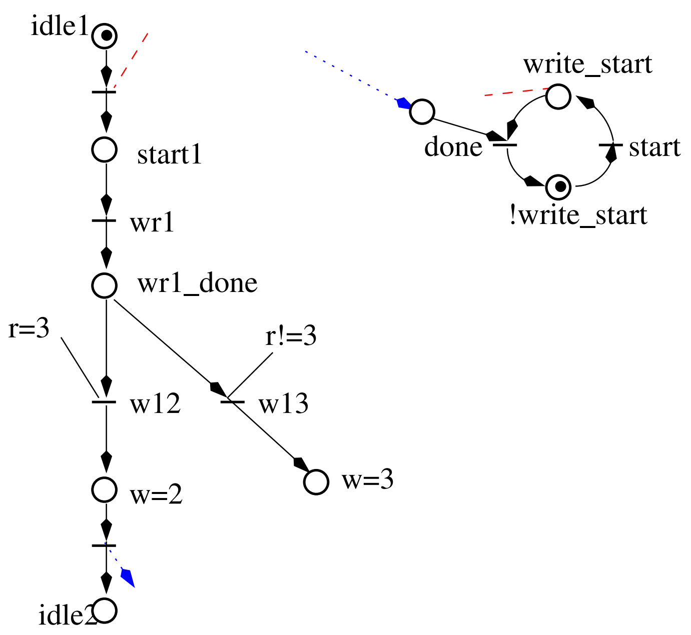


Figure 4.2: Petri nets for Writer

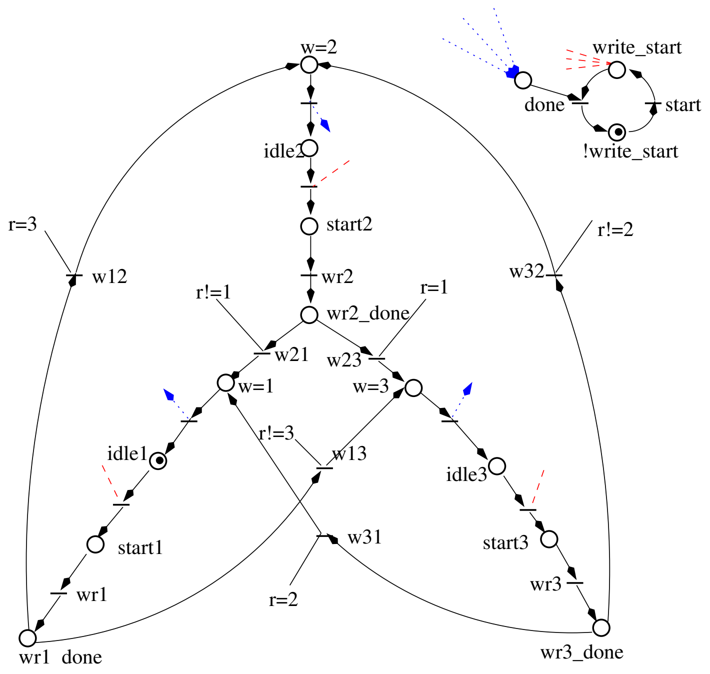


Figure 4.3: Complete Petri nets Model for Writer of 3-slot *Signal*

In the reader, it is idle in the second slot. A place called *"idle2"*, as show in Figure 4.4, is marked initially to represent this state. Similar to the writer, a *read_start* signal could trigger the reader to execute according to Algorithm 2.

This is represented by a read arc from a *read_ start* place joining with the *idle2* place in a transition. The output place is named *"start2"*. The first step of the reader is updating r to l (statement *r0*). According to the algorithm, the reader only performs an action when l and r are not the same. And r would be updated to two possible values: 1 and 3, according to the value of l. In other words, there would be two different updating procedures due to the value of l, which result in two possible states of r: r=1 and r=3. Therefore, a choice with two branches follows *start2* place. The transitions, *r21* and *r23*, in these two branches represent the updating actions. *r21* means updating r from 2 to 1, and *r23* means updating r from 2 to 3. Because the update requires l to be involved, a read arc is attached to each transition. When r is determined, a read action is performed according to the algorithm (statement *rd*). In the first branch of the choice starting from place *r=1*, the read action is modelled as a transition called *"rd1"*. The output place is named *"rd1_ done"* After finishing the read action, the read cycle is completed. An event informs the environment of the completion of the read cycle and sends the reader back to the *idle* state. In the Petri nets, this event is modelled as a fork following the place *rd1_ done* with two output places, one of which is in the environment and the other is *idle1*. The other branch of the choice following place *start2* could be modelled in the same way.

The small net on top right corner in Figure 4.4 is similar to that in Figure 4.2 indicating an environment. The dashed line are read arcs pointing to the main net. The dotted arc is from the last transition in the read cycle in the main net. The initial marking for this net is in the *!read_ start* state. When a *read_ start* signal comes, the *start* transition is fired, and the token goes to the *read_ start* state, which triggers the process in the main net. Once one of the processes is completed, a token is obtained from the *rd1_ done* place. The *done* transition is fired, and the token goes back to the *!read_ start* state.

The remaining part of the Petri nets, including the cases that the reader points

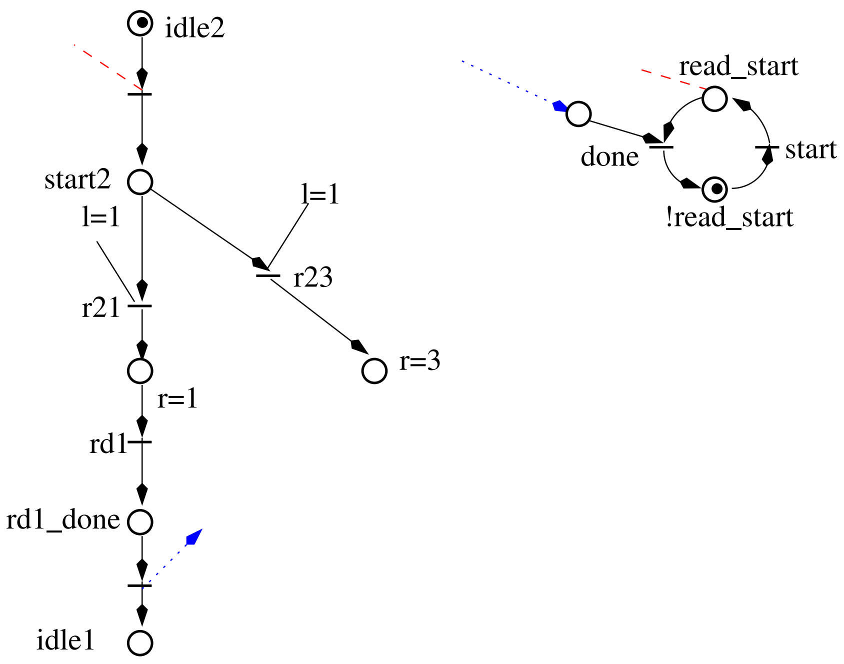


Figure 4.4: Petri nets for Reader

to the first and the third slot, can be generated in the same way, as shown in Figure 4.5.

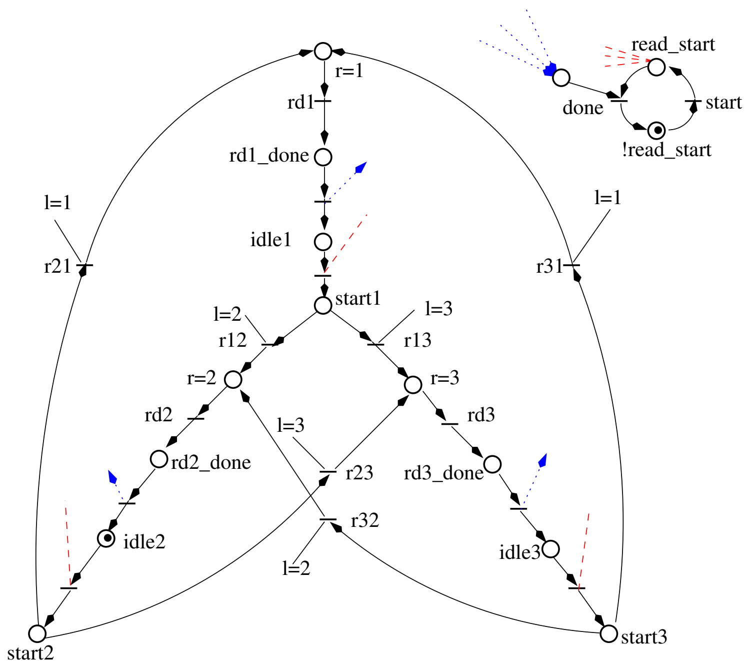


Figure 4.5: Petri net Model for the Reader of 3-slot *Signal*

4.3.2 Reader and Writer

With the Petri net specifications of the reader and the writer, the control circuit model is created in Stateflow. Because there is no concurrency in the models, the reader and the writer are translated directly from the Petri net with the basic connection models in Figure 3.1. Adding the handshakes with the environment, the Mutex and the datapath, the models are completed.

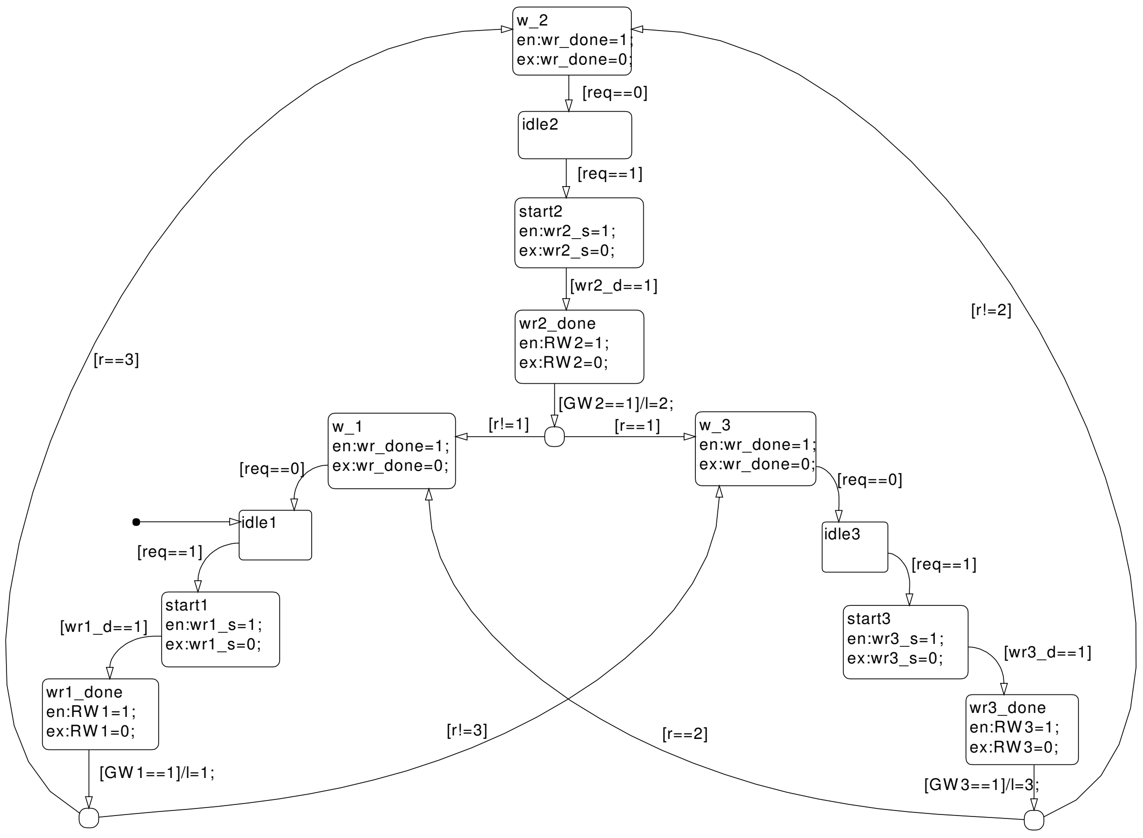


Figure 4.6: Stateflow Model for the Writer of a 3-slot *Signal*

Figure 4.6 illustrates the Stateflow model for the writer, where *req* is the request signal from the environment, and *wr_ done* is the completion signal for the write cycle. *wra_ s* is the request for writing to slot a, and *wra_ d* is the corresponding completion signal. *RWa* is the request to the Mutex after writing data into slot a, and *GWa* is the corresponding grant signal. *l* and *r* are the control variables. In the environment-ACM handshake, the writer acts as the passive component. Once it received the request, it started to perform its process: firstly handshaking with the datapath for requesting a write access (*wr* statement); when the data item has been written, handshaking with the Mutex; once the grant has been given, assigning a value to *l* (*w0* statement) and then choosing a branch for the next write cycle according to the value of *r* (*w1* statement). After the branch was chosen, a completion signal would be sent as an acknowledgement back to the environment. When *req* has been released, a write cycle is completed, and one of the idle states becomes active waiting for the new request.

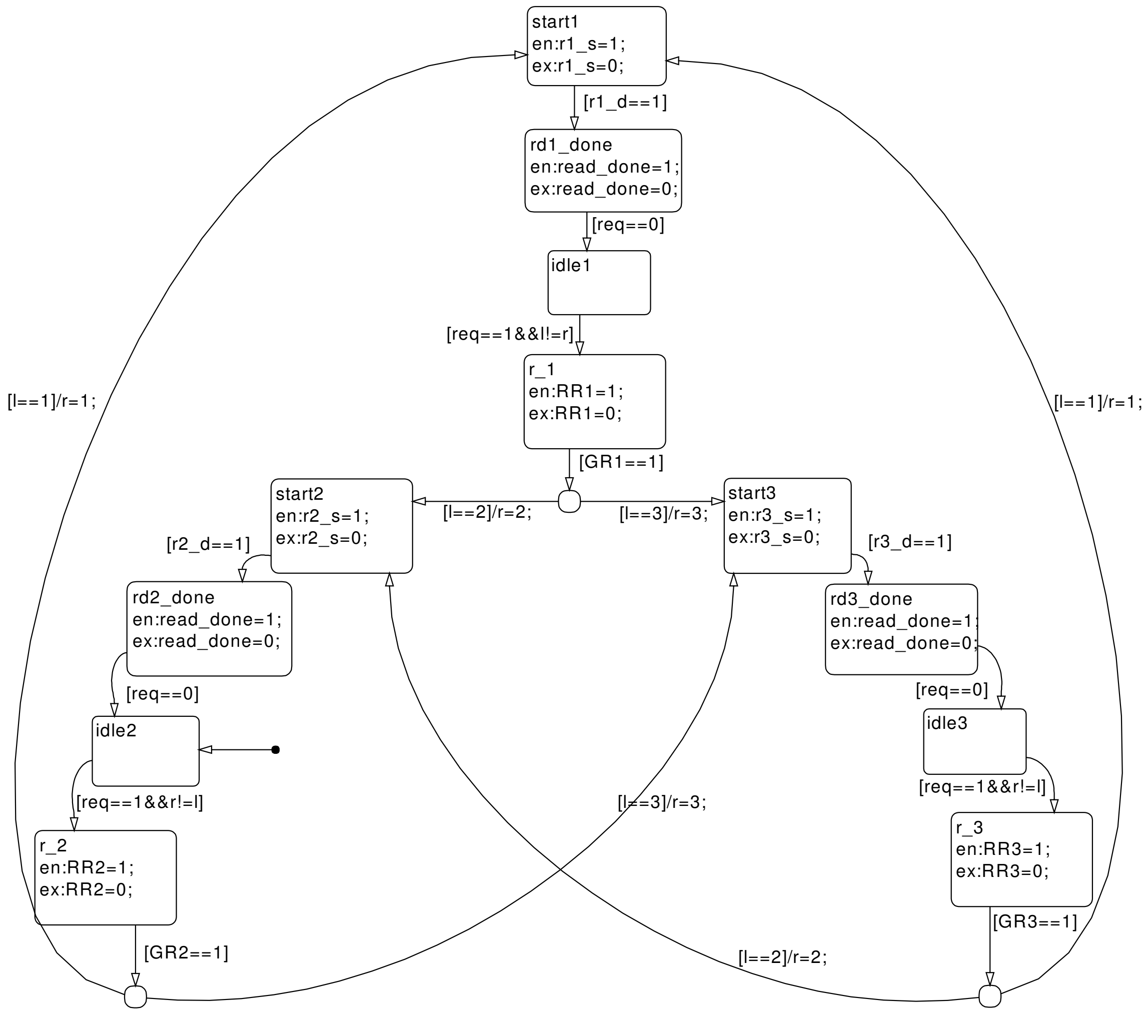


Figure 4.7: Stateflow Model for the Reader of a 3-slot *Signal*

The reader was built in the same way as shown in Figure 4.7. Figure 4.6 and

Figure 4.7 have the same structure with the Petri net models in Figure 4.3 and Figure 4.5.

4.3.3 Mutex

A Mutex could be regarded as a control unit handshaking with two or more concurrent processes in order. In the 3-slot *Signal*, the processes are the reader and the writer. The Mutex grants the process that requests first. If the requests from both the reader and the writer arrive simultaneously, the Mutex grants to one of them randomly. Thus, there are three cases for the order of the requests: writer comes first; reader comes first; and reader and writer come at the same time.

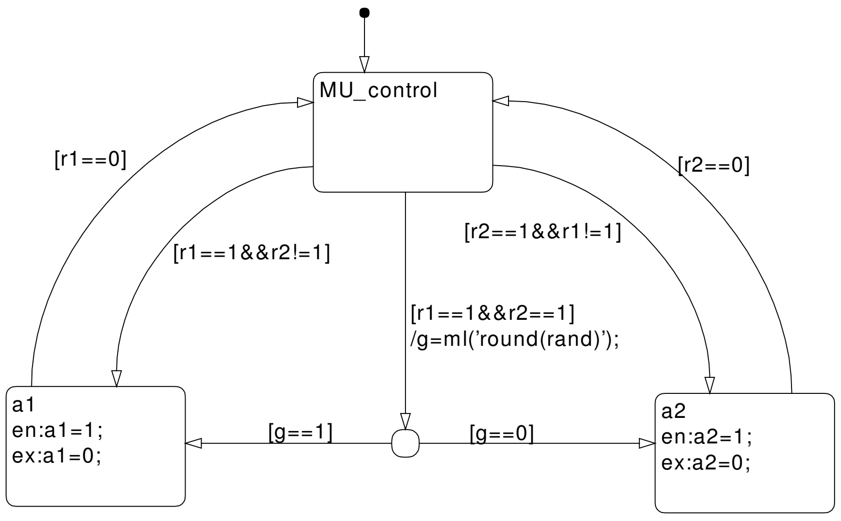


Figure 4.8: Stateflow Model for Mutex

Figure 4.8 illustrates the Mutex model in Stateflow where *r1* and *r2* indicate the requests from two process, *a1* and *a2* represent the grant signals, and g is a flag generated randomly. The three cases mentioned in the previous paragraph are translated into three conditions: [*r1==1&&r2!=1*], [*r2==1&&r1!=1*] and [*r1==1&&r2==1*]. For the first two cases, the Mutex grants the corresponding process directly. For the third case, the Mutex consults the random flag g first. According to its value, the Mutex chooses one process, and the other one has to wait until the chosen one has released its request. To generate g, two MATLAB functions were used. Function "*rand*" generates random numbers between 0 and 1, and function "*round*" rounds the value to the nearest integer. "*ml*" is the keyword for calling a MATLAB function.

4.3.4 Datapath

A datapath is used to store data items from the writer temporarily into certain slots from which they may be picked up by the reader according to the control variables. It is modelled as a subsystem block in the Simulink instead of the Stateflow chart. A subsystem block represents a system within another system. For the datapath block, the inputs include a data input, access requests from both the reader cycle and the writer cycle and, in most cases, two slot indices, one for reading and the other for writing; the outputs include a data output and access acknowledgements. However, in this 3-slot *Signal* model, the datapath would not have any slot indices, because the indices have been bundled with the request signals.

The 3-slot datapath contains two parts. The first part is used for the writer to write a data item into a data slot indicated by the access requests. The second part is used to provide a data link from the requested slot to the reader. The subsystem in Figure 4.9 illustrates the first part for the writing. The switches operate as memory slots. The input data items are fed in the first data inputs, the second data inputs are connected to their outputs. When the request arrives, the corresponding switch provides a link from its first input to the output. When the request is released, the output will switch to the second data input, which has maintained the latest value. That it can be modelled like this to hold the final value, is based on the assumption that there is a unit delay in the switch, which is true in its MATLAB model.

The triggered subsystem in Figure 4.10 shows the part for reading. A triggered subsystem only executes when the trigger event occurs. The trigger event for this

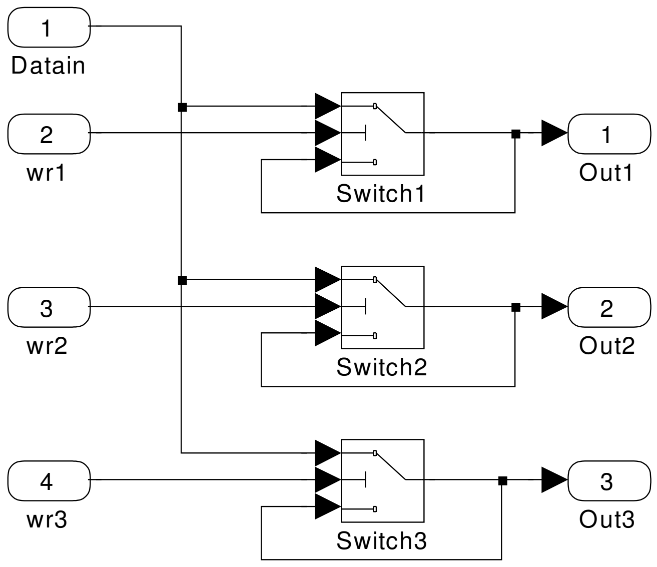


Figure 4.9: Writing into Slots

subsystem is that any of the 3 read access requests come. In the Simulink model, it is modelled as a logic OR in Figure 4.11. In Figure 4.10, the data items are multiplied by the corresponding read access requests. Because two of the requests remain at 0, only the requested slot delivers its value. Adding all the outputs from the Products, the result was the item desired. The data in the Dataout port would not change until the next read access came.

The whole datapath model, in Figure 4.11, connected the models in Figure 4.9 and Figure 4.10. Since all the blocks in the models in Figure 4.9 and Figure 4.10 are treated as atomic, the memory blocks are added to simulate the latency in the reading and writing operations. The completion signals, or the acknowledgement signals, are the delayed requests.

4.3.5 Test Environment

A test environment block is used to test whether an ACM works according to the specification. The test environment generates input data items and requests with random intervals. The interval here means the period between the previous acknowl-

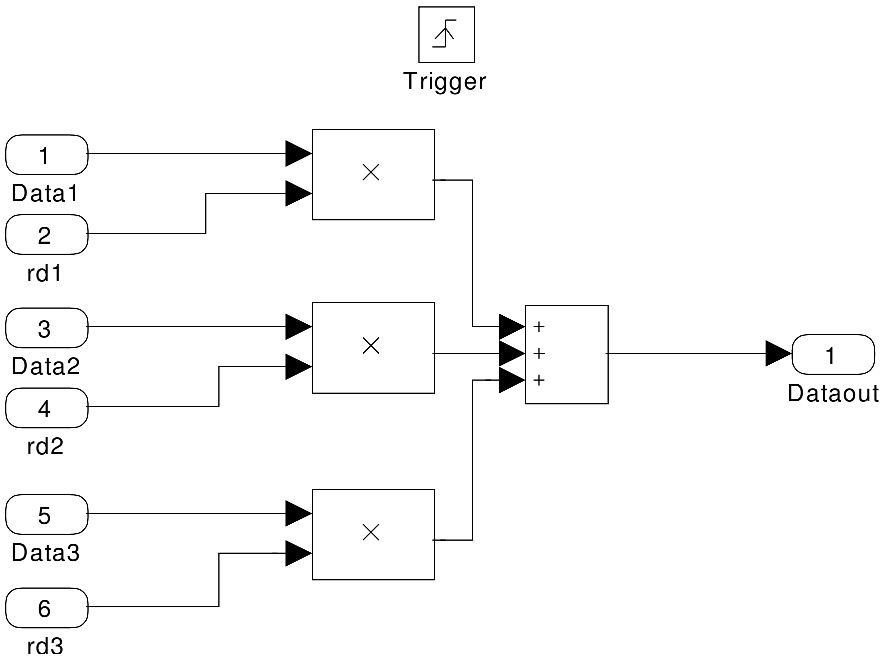


Figure 4.10: Reading from Slots

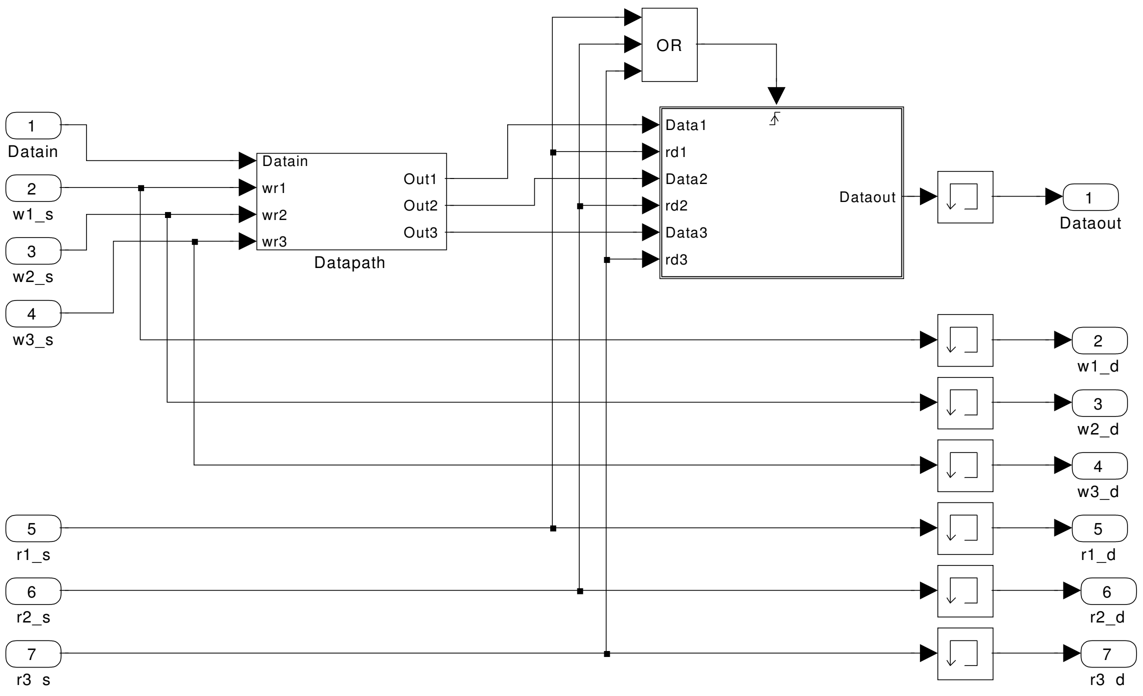


Figure 4.11: Simulink Model for Datapath

edgement and the next request. The reason for choosing the random interval is that it gives enough variation for the reader and the writer to be asynchronous. The duration of the random intervals has exponential distribution.

The random numbers are generated according to the following formula:

$randexp() = -\mu \times \log(rand());$

where *randexp()* is the random number conforming to the exponential distribution; μ is the mean of the random numbers; *rand()* is a function used to generate uniformly distributed random numbers which are between 0 and 1. Because the intervals are always positive, the mean μ must be a positive number.

This formula is taken from a MATLAB function called *exprnd()* which generates exponential distribution random numbers. For the detail of this formula, please refer to [L.86].

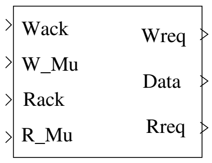


Figure 4.12: Test Environment Block

Figure 4.12 illustrates the inputs and outputs of the test environment block. *W_Mu* and *R_Mu* refer to the two means of the two random number sequences. *Wack* and *Rack* are the acknowledgements from the writer and the reader, which are used to trigger the blocks for generating random numbers. *Wreq* and *Rreq* are the requests sending to the writer and the reader. *Data* is the source of data items for the ACM's data inputs.

Figure 4.13 shows the Simulink model under the mask of the block shown in Figure 4.12. The block with a rising edge mark at the top is called a triggered subsystem, which only executes when the trigger event happens. The two triggered subsystems are identical. The subsystem plays the role of generating random intervals according to the input parameters *Mu* and *Rand*. A digital block is used as a time reference.

When an acknowledgement signal triggers the subsystem, it captures the time

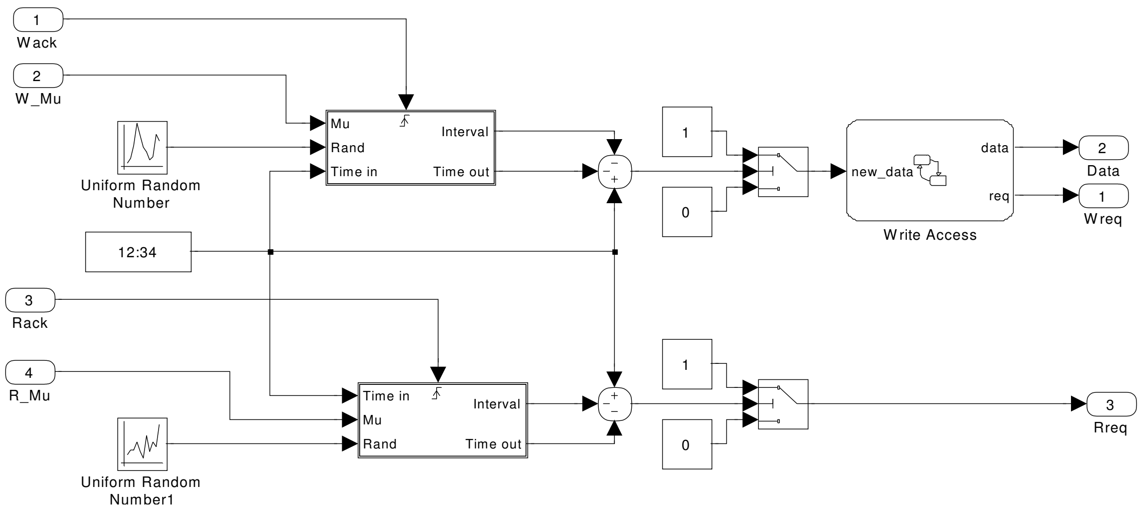


Figure 4.13: Simulink Model for Test Environment

as a reference and delivers to the output. At the same time a random number is worked out according to the inputs *Mu* and *Rand*. The captured time *tc* indicates the time when the acknowledgement arrives, while the random number is regarded as the interval *ti*. As a result, *tc* + *ti* is the time for sending the next request, that is, when the digital clock reaches *tc* + *ti*, a request is generated. In the Simulink model, it is implemented as a sum block with two negative inputs connected to the interval and the captured time, and a positive input connected to the time reference. The output of the sum block is connected to the control input of a switch. The first data input of the switch is connected to a constant block whose value is 1, while the second data input is connected to 0. The criteria for passing the first input of the switch is that the value of the control input is not less than 0. The output of the switch is sent to either the read cycle as a read request, or a chart called *Write Access* to generate data items and deliver the write requests to the write cycle.

Figure 4.14 shows the trigger subsystem for generating a random number. The trigger mark on the top refers to the trigger mode. In this case, it is a rising edge trigger system. The time input is connected to the output directly to capture the time when the trigger signal comes. The input *Mu*, the same asμ in the formula, is

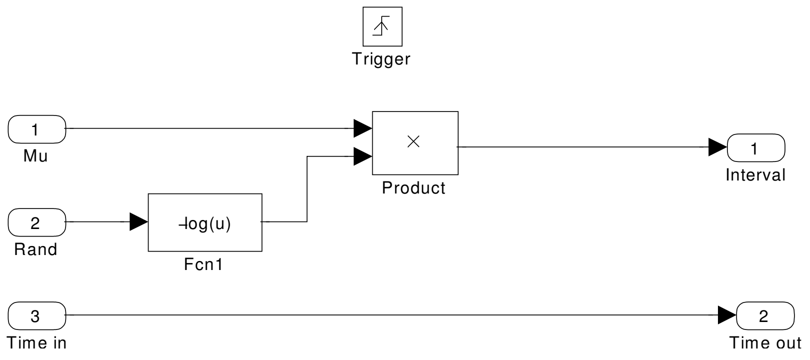


Figure 4.14: Trigger Subsystem

the mean of the random numbers. The input *Rand* obtains random numbers from the uniform random number generator. The time interval is the product of *Mu* and -log (*Rand*) according to the formula.

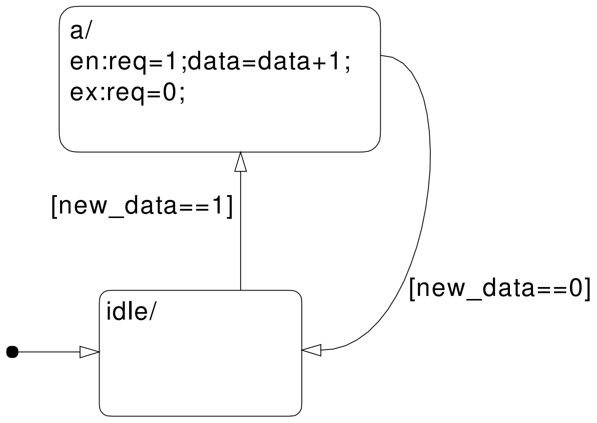


Figure 4.15: Date Item Generator

Figure 4.15 illustrates the *Write Access* chart. The default state is the *idle* state. If the input is 1 (*[new_ data==1]*), the active state moves to state *a*. In the entry action of the state *a*, a request (*req=1*) and a data item (*data=data+1*) are generated. Once the input changes to 0 (*[new_ data==0]*), the request is released (*req=0*) in the exit action, and the active state goes back to the *idle* one. In the chart, the initial value of *data* is 0 so that an increasing natural number sequence, which is easy to monitor, can be sent in the write cycle. To start the whole system, the initial value of *req* is set to 1.

Here is how the whole model works: when an acknowledgement (a rising edge) arrives, a time *tc* is captured and a interval *ti* is calculated. The output of the sum block becomes a negative value. As a result, the switch sends the value in the second data input, which is 0, to the output. This is regarded as clearing or withdrawing the previous request. Once the output of the sum block changes to a positive value when the time reference reaches $tc + ti$, the switch delivers the value 1 to the output. This signal is treated as the request to the read cycle or the trigger for the *Write Access* chart for sending the write request and the input data items.

4.3.6 Simulation Results

After manually checking the correctness for the 3-slot *Signal* model, it is plugged into the test environment. The means of the writing (Mu) and the reading (Mu1) intervals are set to two values close to each other. This gives enough variation for the writer over-writing and the reader waiting to appear during relatively short simulation runs. The waveforms in Figure 4.16 are a part of the simulation results which indicate the correctness of the OW and NRR properties.

In the Figure 4.16, after data item 5 was read, the writer delivered faster than the reader required so the items from 6 to 8 were overwritten. After that, the read requests came before the write requests. Because there were no new data items available, the reader would not process until a data item was written, and the read request stayed high. It preserves data freshness property because the reader always reads the latest item available in the slots. The reader and the writer indices were never the same value simultaneously, which implies the reader and the writer never accessed the same slot simultaneously therefore, the data coherence property is preserved.

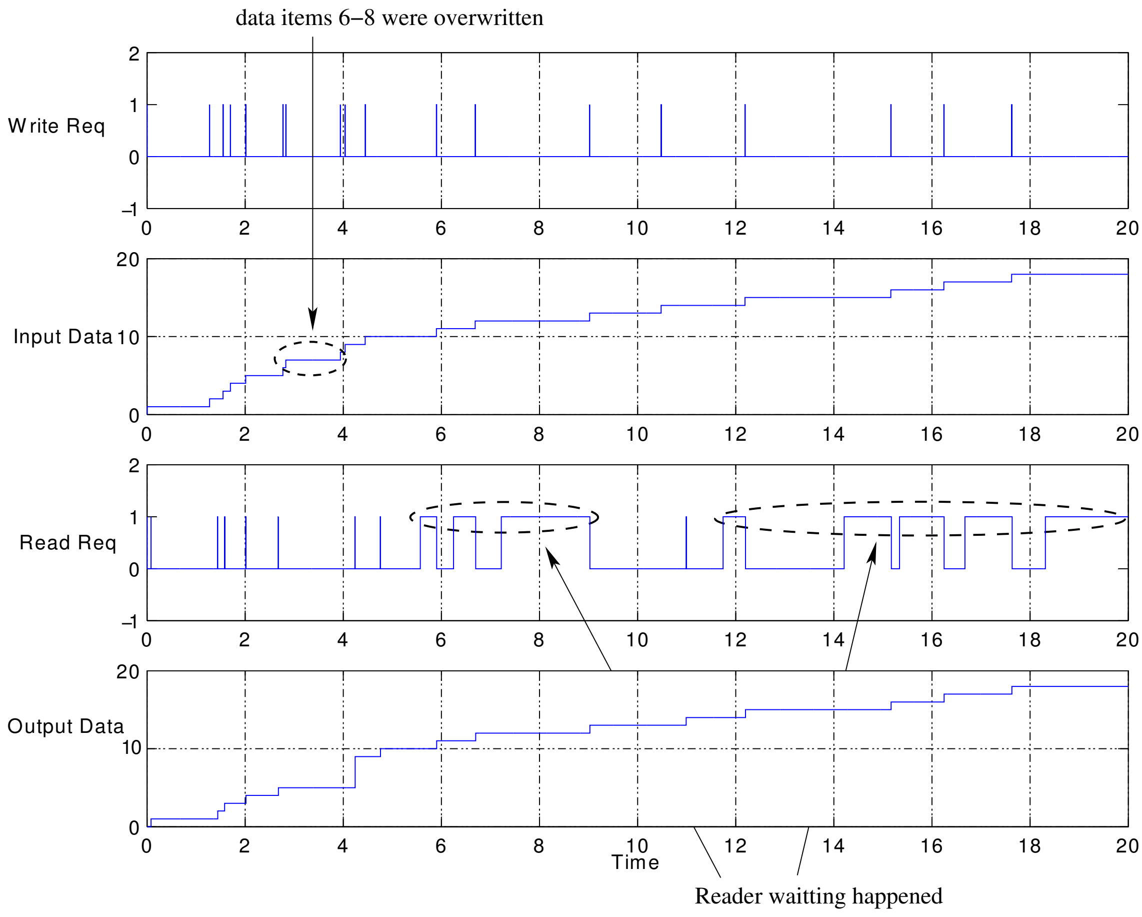


Figure 4.16: Waveforms for the 3-slot *Signal*

4.4 ACM Models in MATLAB

4.4.1 2-Slot *Channel*

4.4.1.1 Controller Model

The controller of a 2-slot *Channel* is modelled based on the Algorithm 1. It represents the actions which the writer and the reader perform when they receive requests. This implies that there are handshakes between the environment, the writer and the reader. The environment is active, and the controller is passive. This fact also applies to the controller of the 3-slot *Signal* and the 4-slot *Pool* which will be mentioned later. The algorithm itself also has one handshake pair in each side. It is the handshake with the datapath. In this case, the controller is active because it sends requests, and the datapath is passive.

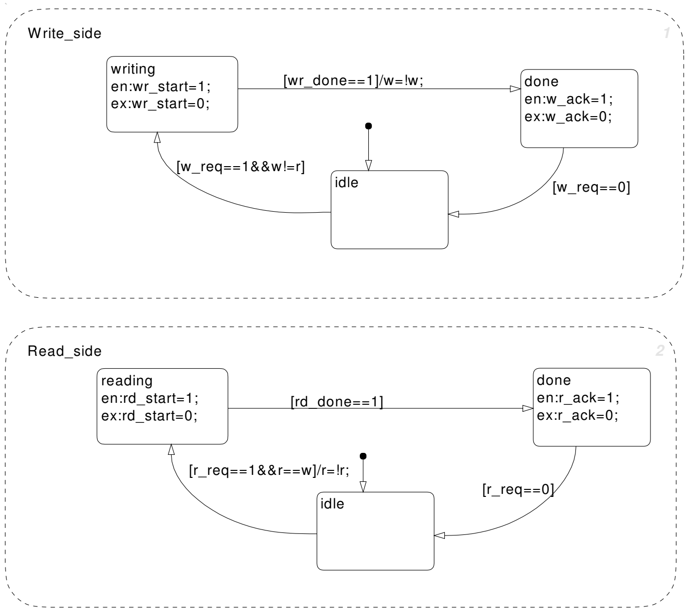


Figure 4.17: Stateflow Model for 2-slot *Channel*

The model is shown in Figure 4.17. The *wait until* statements combined with the requests are added as the transition conditions between *idle* and accessing (*reading* and *writing*) states. Because the write index is updated after a writing operation, it is put in the transition after the *writing* state as the transition action. The read index is updated in the transition action before the *reading* state since it is executes before the read operation.

There are ten variables in the chart. The ones assigned with new values, such as *wr_ start*, *rd_ start*, *w_ ack*, *r_ ack*, *r* and *w* are output parameters. The referenced ones, such as *w_ req*, *r_ req*, *wr_ done* and *rd_ done*, are input parameters.

It works as follows.

For the write side, the initial state is the *idle* one. When a write request comes and the write index is not the same as the read index, the active state moves to the *writing* state. At the same time, it sends a *wr_ start* signal to the memory in its entry action. Once the *wr_ done* signal is received from memory, which indicates the data item has been written in to the corresponding slot, it withdraws the *wr_ start* signal in the state exit action, and the write index is updated to its complement value in the transition action. The active state comes to the *done* state with sending a *w_ ack* to inform the environment about the completion of the writing operation in the entry action. When the *w_ req* is withdrawn, the *w_ ack* is withdrawn in the state exit action. Then the active state moves back to *idle*.

The read side is similar to the write side, the initial state is also the *idle* one. When an *r_ req* comes with the condition *r==w* satisfied, the index *r* updates its value to *!r*. The active state moves to the *reading* state with the entry action, sending *rd_ start* signal to the memory, being executed. When a *rd_ done* is received from the memory, which means the data item has been read by the reader, *rd_ start* is withdrawn in the state exit action. After that, the active state comes to the done state, and the entry action is executed for sending *r_ ack* to the environment. When the r_req is released, the w_ack is withdrawn, and the active state comes back to the idle one.

4.4.1.2 2-Slot Datapath Model

Figure 4.18 is a block of a datapath containing more than one slot. It has 5 input ports, including 1 data input: *Datain*, two requests: *Write_ start* and *Read_ start* and two slot indices: *w* and *r*. Because there are only two slots in the memory, *r* and *w* are set to be binary.

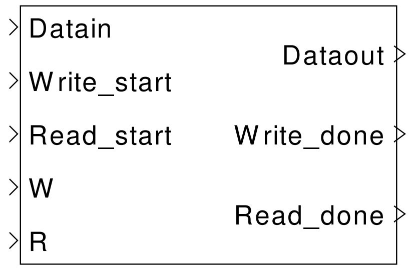


Figure 4.18: Datapath Block with more than One Slot

The datapath in this case is also divided into two stages, as shown in Figure 4.19. The first stage is for the writer and the second is for the reader.

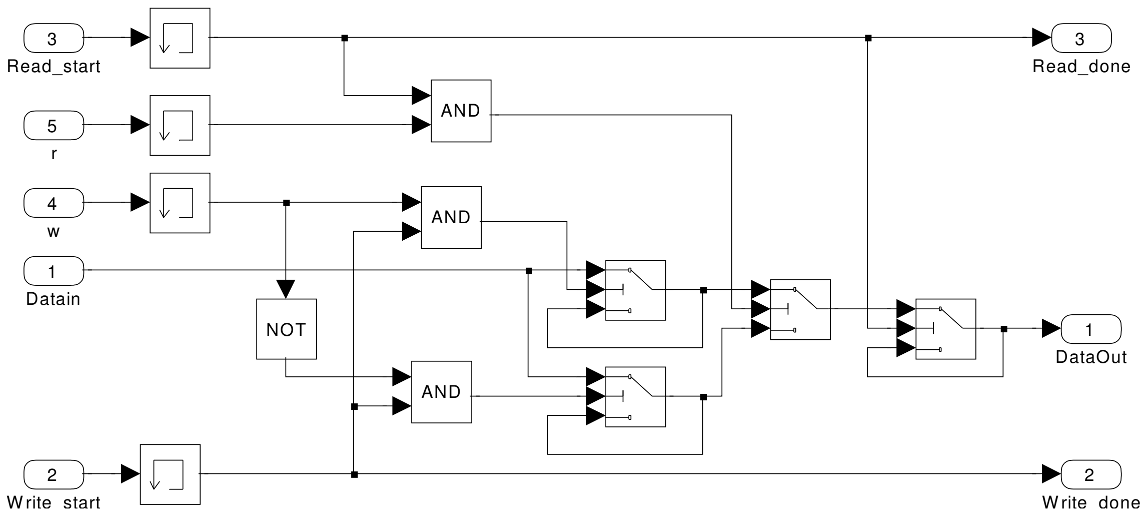


Figure 4.19: 2 Slots Memory

The first stage contains two switches in parallel representing two slots. Both the first data inputs of the two switches are connected to the input port *Datain*; the second ones are connected to the output ports to hold the data item when the slot is not selected. Index *w* conjugated with *write_ start* is used to choose which slot is used for writing. Therefore, after the AND logic, they are connected to the control ports of the switches. When *w* is 0 the data is delivered to the output of the second switch, otherwise, it is sent to that of the first one.

The second stage is made up of two switches in series. The two data inputs of the first switch connect to the two outputs from the previous stage. The result of *r* Logic AND with *read_ start* is connected to the control port to select a slot for reading. It is easy to see that this switch always passes the value in the second slot when the *read_ start* is off. But this is not what is desired, because the older item might be delivered after the latest one. In order to keep the output stable, another switch is used. It passes the output value of the previous switch when the *read_ start* is on, or holds the final value (or is set to zero, depending on the requirement) when it is off.

It works as follows. When a write request comes, the data item from port *Datain* is sent to the output of one of the first state switches according to the write index w. The data item is held in the switch for the next read when the *write_start* is withdrawn. When a read request arrives, the first switch in the second state passes the data item in the slot referred by r. The second switch holds the data until another data item is read.

With connecting the controller and the datapath, the 2-slot *Channel* is completely modelled.

4.4.2 3-Slot *Signal*

4.4.2.1 Controller Model

Using the same technique as with the 2-slot *Channel*, the Stateflow model of a 3-slot *Signal* is created as shown in Figure 4.20 according to Algorithm 2. The *neither* function is implemented as a graphic function. It is easy to see from the Table 2.4 that the result of the neither function obeys the following rules which can be easily programmed:

1) If the two parameters (l and r) are equal, the result w is the modulus of the value of the parameter plus one and 3, i.e. if [l==r], then w=mod (l+1, 3);

2) If the difference between the two parameters is 1, the result is the modulus of the value of the larger parameter plus one and 3, i.e. if [abs(l-r)==1], then w=mod (max (l, r)+1, 3);

3) If the difference between the two parameters is 2, that is, the two parameters are 0 and 2, the result is 1, i.e. if [abs(l-r)==2], then w=1.

The syntax of the graphic function statements is very similar to C. *ml* in the function denotes to call a MATLAB function. The three branches between two junctions illustrate the three connection cases between l and r, as stated in the previous section. Because this function is only called by the writer, it is a local function. As a result, it is put into the writer's round rectangle.

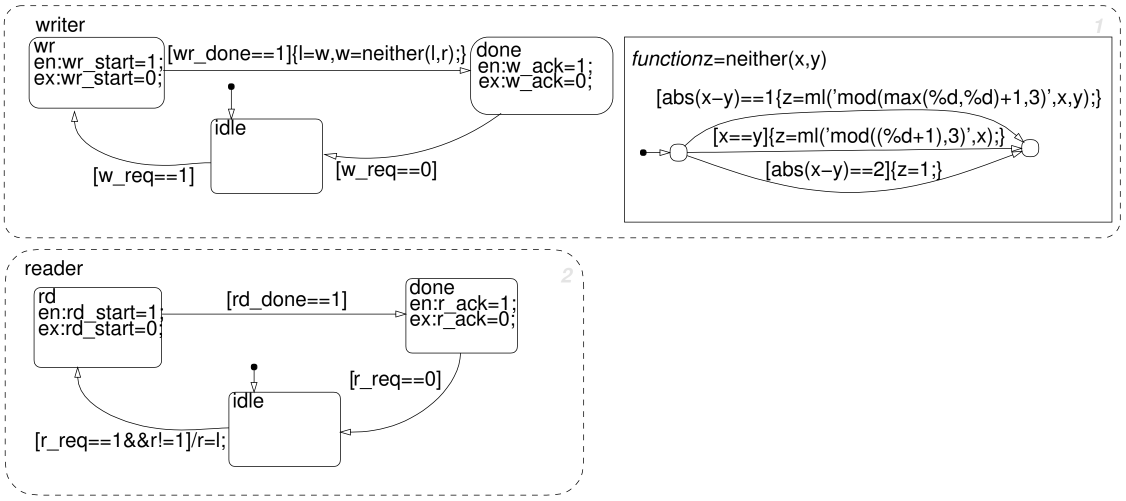


Figure 4.20: Stateflow Model for a 3-slot *Signal*

There are eleven variables used in this chart. Ten of them have the same names as the 2-slot *Channel*, and the difference is that the read index and the write index are integer types instead of binary. The variable not appearing in the 2-slot *Channel* is *l*. *l* is only used inside the chart, so it is classified as a local parameter.

Connecting this chart with a 3 slot memory which will be introduced in the next subsection, the 3-slot *Signal* is modelled completely.

The model works as follows:

For the writer, the active state moves from its initial state *idle* to *wr* when the condition *[w_ req==1]* is satisfied, i.e. a write request comes. A *wr_ start* signal is triggered when entering the *wr* state. The active state will stay at *wr* until the condition *[wr_ done==1]* is satisfied which implies that the writing operation is completed. Before the two control variables *l* and *w* are updated in the transition action, the *wr_ start* is withdrawn when leaving the *wr* state. A *w_ ack* is generated in the entry action of state *done* to acknowledge to the environment the completion of the write cycle. With the condition *[w_ req==0]* being satisfied, the write request being withdrawn, the *w_ ack* is released in the exit action of *done* state, and active state goes back to *idle*.

For the reader, the active state will stay at *idle* and *r* will not be updated to *l* until a read request comes and the read index is not the same as the most recently written index, that is, *[r_ req==1&&r!=l]* is true. When the condition is true, the *rd* state becomes active and *rd_ start* is generated in the entry action. After receiving the *rd_ done* signal from the memory (*[rd_ done==1]* is satisfied), the active state leaves *rd* with the execution of the exit action for releasing the *rd_ start*, and comes to *done* with generating *r_ ack* in its entry action. When *[r_ req==0]* is true, the reader releases the *r_ ack* in the exit action within the *done* state, and the active state moves back to *idle*.

4.4.2.2 Generic Datapath Model

The three or more slot memory structures are more complex compared to the 2-slot structure, although they have the same number of inputs and outputs. They are more complex because the slot indices *r* and *w* because integer numbers instead of binary. Figure 4.21 illustrates a 3-slot datapath.

The input data is stored into the corresponding slot when *write_ start* arrives. As stated before, a slot can be modelled as a switch and a memory block. In order to store an input data item into a certain slot, a "0" is sent to the control port of the slot, and the input data has to be fed to the second data ports. The reason for doing it this way is that the criterion for passing the first data port is preset so that the value of the control port is non-zero in MATLAB and can not be inverted.

Another switch is used for passing w to the control port of each slot only when a *write_ start* signal comes. As explained in the previous paragraph, a "0", instead of the actual *w*, is sent to the control input of the chosen slot. Therefore, the value of *w* has to be decreased to 0 before it is connected to the corresponding slot. This is achieved by adding a sum block before each slot except the first one, as shown in

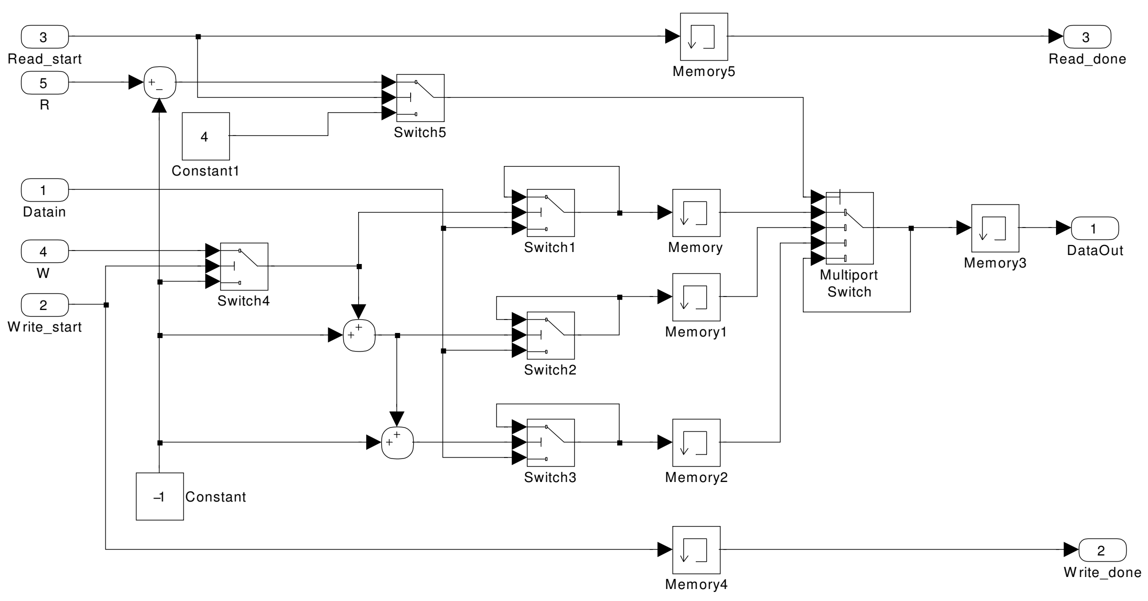


Figure 4.21: 3-Slot Datapath

Figure 4.21, since the indices start from 0.

It is a good idea to use a Multiport switch block to model the reader part because the index of r is a natural number. A Multiport switch allows the data from more than two inputs to be delivered to the output corresponding to the truncated value of the control port, as shown in Figure 2.18. The number of data input ports (*Ndin*) of the Multiport switch is determined by the number of slots (*Nslt*) in the ACM to be modelled. *Ndin*=*Nslt*+1, so here *Ndin* is 4. The first data input ports of the Multiport switch are linked to the outputs from the writer part; while the last one is connected to its own output to keep the output data stable.

In the control path modelled before, the indices are counted starting from 0. However, the control port value of the Multiport switch is a natural number counted from 1, which states the port in which data is to be delivered. To amend this difference, r+1, which appears as r-(-1) in the Simulink model, is fed to the control port instead of r when the *read_start* signal arrives. In order to keep the output stable after *read_start* is withdrawn, a constant value from the 4th port needs to be passed through. Therefore the constant 4 is fed to the control port of the Multiport switch through a switch controlled by *read_ start*.

Figure 4.21 can be extended to a 4 slot datapath by adding one more branch in the first stage, modifying the value in the *constant 1* block to 5 and adding one more data input to the Multiport switch.

The binary ”0” and ”1” can be treated as integer ”0” and ”1”, so this modelling technique can also be extended to model a 2 slot datapath by removing one branch in the first stage, modifying the value in the constant 1 block to 3 and removing one data input from the Multiport switch.

4.4.3 4-Slot *Pool*

4.4.3.1 Controller Model

4-slot Pool is modelled according to Algorithm 3. The read index and the write index of a 4-slot *Pool* in the algorithm stated are determined by two binary variable pairs. The former one depends on n and $s[n]$, while the latter one hinges on r and $v[r]$. However, the indices for a 4-slot datapath, which can be extended from the 3-slot one, are single natural numbers. Thus, the indices in the model have to be converted from two binary digits to a single number. It is possible to use binary indices as well. However, it will have two outputs for each index, which requires a redesigned datapath.

In order to do the conversion, the slots must be ordered correspondingly as shown in Table 4.3.

	Left (0)	Right (1)
Top (0)	0	2
Bottom (1)	1	3

Table 4.3: The Order of 4 Slots

The left slot pair, pair 0, is defined as the first two slots (slot 0 and slot 1), and the right pair (pair 1) is defined as slot 2 and slot 3. The conversion is binary to decimal conversion.

The Stateflow model of the control part for a 4-slot *Pool* is shown in Figure 4.22. The techniques used here are the same as those in the 2-slot *Channel*. The graphic function *b2d* plays the role of binary to decimal conversion. Since the conversion is used by both the writer and the reader, it is placed outside of both the reader and the writer rectangles as a global function.

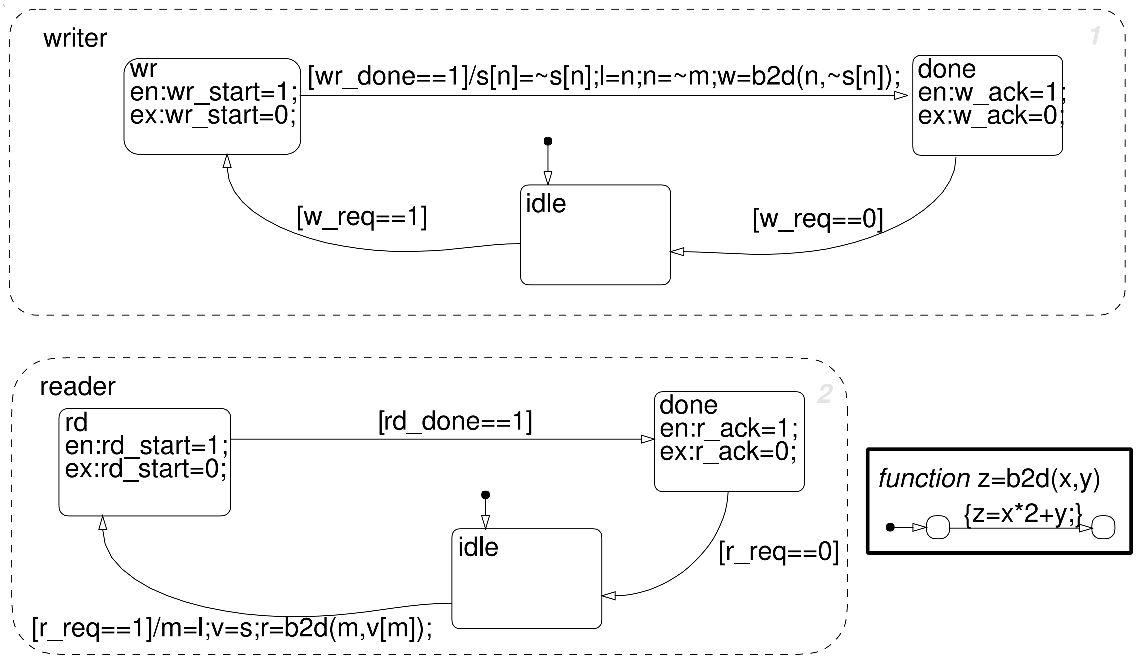


Figure 4.22: Stateflow Model for 4-slot *Pool*

There are altogether 15 variables in the chart. Eleven of them inherit their names from the 3-slot *Signal*. However *l*, referring to the most recent written pair rather than slot, is a binary variable. Two of the rest, *m* and *n*, are also binary variables. Because *r* is used as the reader index in this model, another name, *m*, is assigned to the variable *r* in the algorithm. The other two, *s* and *v* are also binary arrays with two elements. All these four variables are only used by the chart itself, hence, they are set as local ones.

By adding a 4 slot datapath as shown in Figure 4.23, the whole model is built completely.

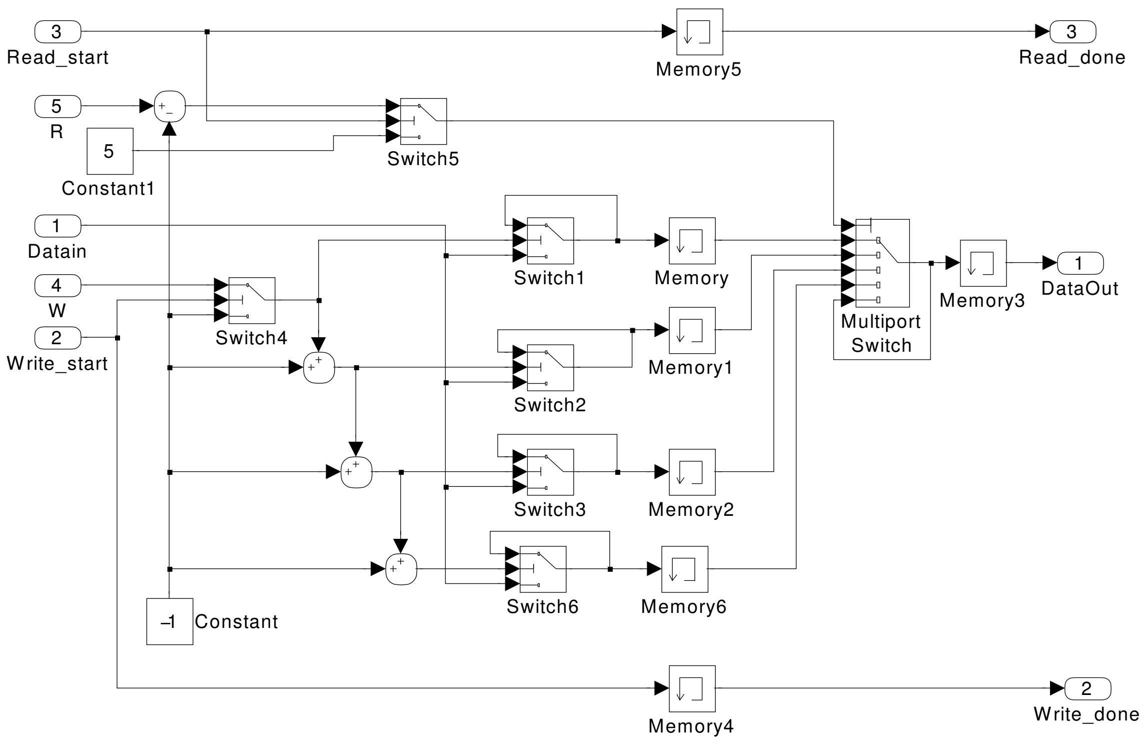


Figure 4.23: 4-slot Datapath

The following paragraphs illustrate how the model works.

When a write request comes from the environment, the active state moves from the initial state *idle* to *wr.* At the same time, a *wr_ start* is transmitted to the datapath in the entry action. The data item will be stored in the slot according to the variable *w.* Once the writing operation is completed, a *wr_ done* is sent back to the control part. On receiving this signal, the writer releases the *wr_ start* in the exit action in *wr*, updates the control variables stated in the *w1* and *w2* statements in the algorithm in the transition actions, and then enters the *done* state. In the state entry action, a *w_ ack* is delivered to acknowledge to the environment the completion of writing. The writer will not leave this state until the environment releases the request. The *w_ ack* is withdrawn before *done* becomes inactive and *idle* is active again.

The reader is in the *idle* state at the system start-up time. When receiving a read request from the environment, the reader updates the control variables *m* and *v*, and calls the *d2d* function to work out the read index *r* in the transition action. After the transition action is completed, the *rd* state becomes active and the entry action is executed to send an *rd_start* signal to the datapath. Once the data item stored in slot *r* is read, an *rd_done* is transmitted back to the reader. The reader leaves the *rd* state with the exit action being executed (releasing the *rd_start*) and enters the *done* state with the entry action being executed (acknowledging to the environment the completion of reading). When the request is withdrawn from the outside, the reader releases the acknowledgement signal and leaves the *done* state for the *idle* state.

4.4.4 Simulation Results and Discussions

4.4.4.1 Simulation Results for 2-slot *Channel*

Figure 4.24 shows the Simulink model for applying the modelled *Channel* in the test environment.

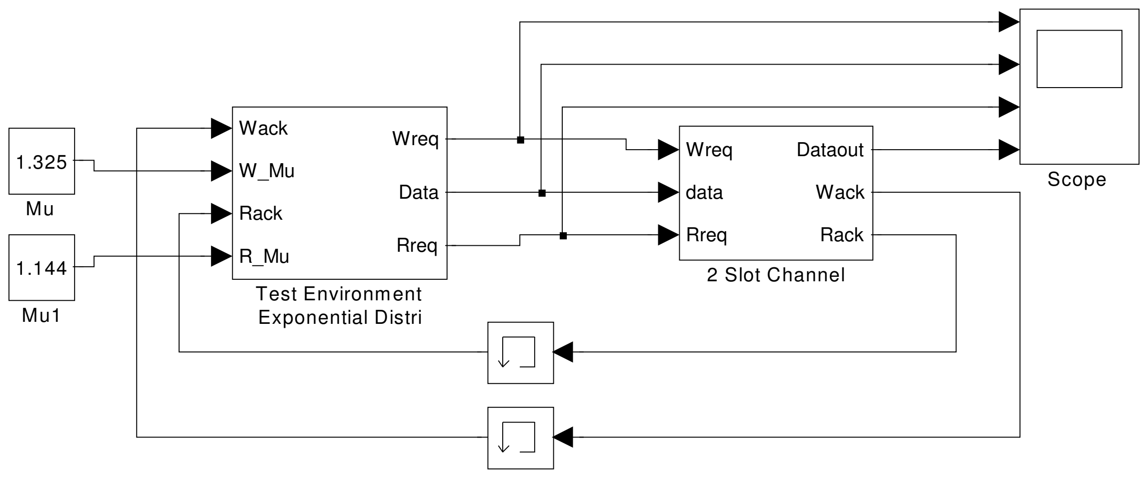


Figure 4.24: A 2-slot *Channel* in the Test Environment

The result waveforms for the 2-slot *Channel* are shown in Figure 4.25. The four waveforms in Figure 4.25 are: write requests, input data items, read requests and output data items.

Whenever a new data item arrives, a write request is generated. The rising edge of a write request refers to the arrival of a new data item, and the falling edge indicates that the new data item has been written successfully into the data slot. The rising edge of a read request indicates that there is a demand for a data item in the read process, and the falling edge means that the data item has been read successfully from the data slot. As the durations for the read and write operations are quite short, the normal requests appear as a single impulse in the waveforms. As a result, a continuous high level of a request in the waveform indicates the reader or the writer was waiting.

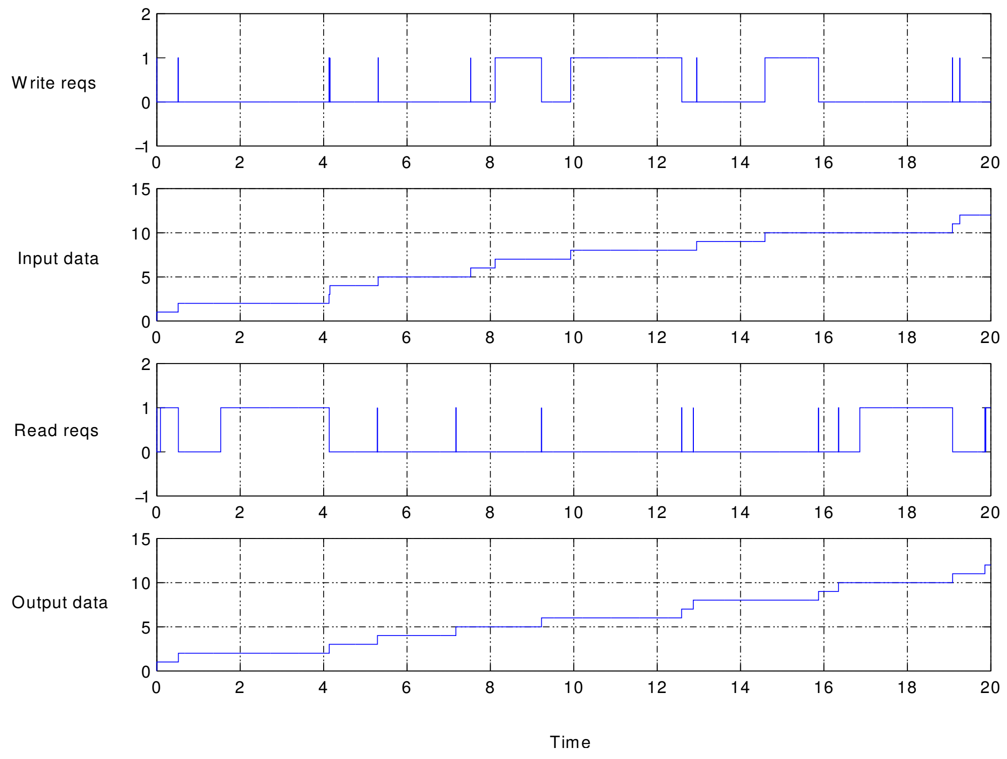


Figure 4.25: Waveforms for 2-slot *Channel*

In this figure, the writer was waiting when data items 7, 8 and 10 arrived because the *Channel* was full. After data items 1, 2 and 10 were written, the write requests were delivered slower than the read requests. The reader would not respond to the requests until a new data item was written into a slot because the *Channel* was empty.

It is obvious that the output data items kept the same order as the input ones and there is no data loss in the *Channel*. However, the asynchrony property is not preserved because the reader and the writer are dependent on each other.

4.4.4.2 Simulation Results for 3-slot *Signal*

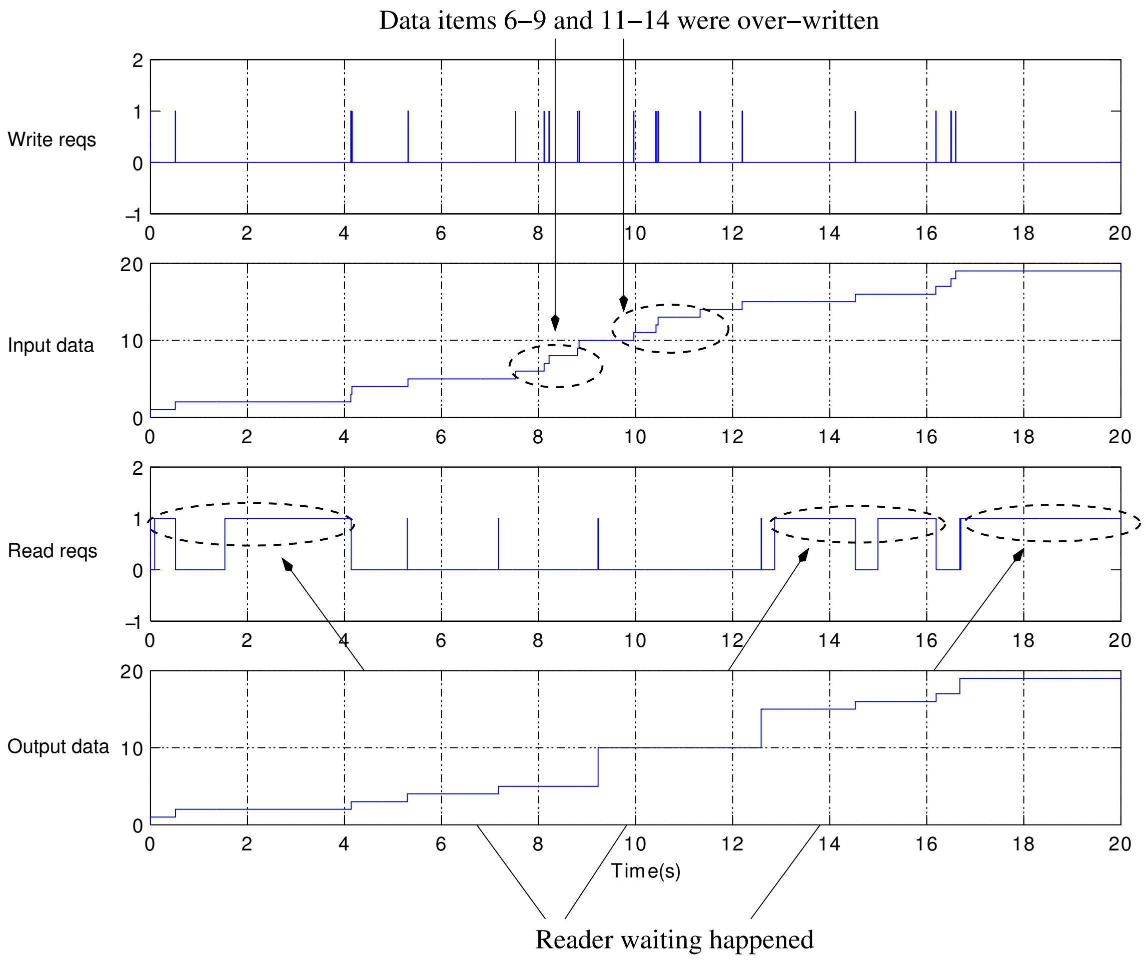


Figure 4.26: Waveforms for 3-slot *Signal*

Figure 4.26 shows the resulting waveforms for applying a 3-slot *Signal* into the test environment. The four waveforms are: write requests, input data items, read requests and output data items. In the waveforms, after data items 1, 2, 14, 15 and 16 were read from the slots, another read request arrived. However, the new data item is not available. The reader did not respond to the requests (waited) until the new data item was written into the memory. During this period, the read request stayed at a high level, as shown in Figure 4.26. If the write requests are delivered faster than the reader ones, such as in the period between data items 5 and 15, the writer will over-write the old data items to keep executing without blocking. It appears in the output waveform as jumps from 5 to 10 and then to 16. It can be seen that this 3-slot *Signal* model keeps the same order as the input data, but may have data loss in its output. It preserves the data freshness property as the reader always reads the latest data items available in the *Signal*. The coherence property was observed by monitoring whether the reader and the writer indices were the same value at the same time. No violation was detected. The writer is independent from the reader, but the reader may be blocked by the execution of the writer. This fact shows that the 3-slot *Signal* does not preserve the asynchrony property.

4.4.4.3 Simulation Results for 4-slot *Pool*

Inserting the modelled *Pool* into the test environment, the model was ready for simulation. The resulting waveforms for the test are shown in Figure 4.27. The four waveforms are: write requests, input data items, read requests and output data items. The read requests and the write requests in the waveforms all appear as impulses, which is because there is no waiting in either the reader or the writer. When the data items were delivered faster than the reader requested them, the data items were overwritten by the writer. The overwritten items would not be transmitted to the reader, therefore, it showed higher steps in the output data waveform, such as after data item 2 was read. If the reader requested more data items than the writer could provide, the reader read the latest data item again to keep it running. This appears such that the multiple read requests came without any changes in the output data waveform, such as after data item 16 was read.

It can be seen from the result that the 4-slot *Pool* model has an independent

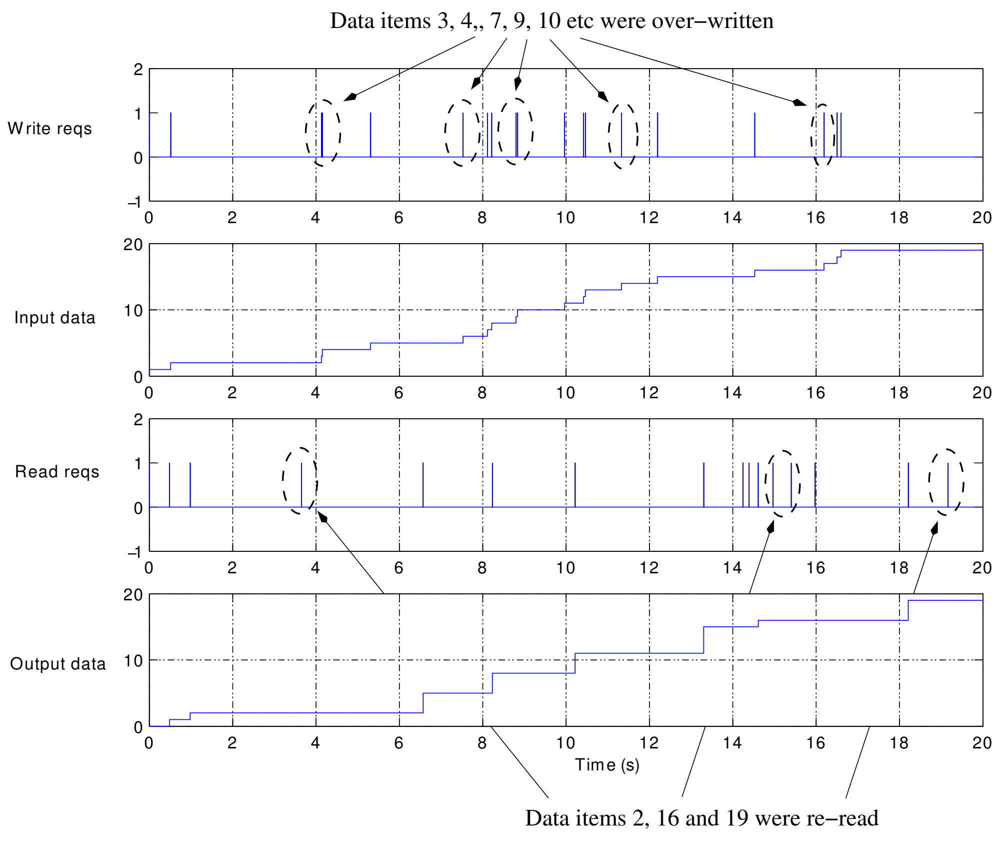


Figure 4.27: Waveforms for 4-slot *Pool*

reader and writer who will not block each other. It always delivers the most recent data items to the reader. In other words, a 4-slot *Pool* is fully asynchronous and preserves the data freshness property. As a trade-off, it cannot avoid data losses.

4.4.5 General Procedure for Modelling

4.4.5.1 General ACM Scheme

According to the 3 models described above, the general ACM scheme [Xia00], shown in Figure 4.28, includes two sequential processes, the reader and the writer cycles. These two processes are concurrent, and assumed not to be temporally related to each other.

In the writer, there is an access to a data slot at some part of the cycle; there may be accesses (including setting and reading) to control variables at some part

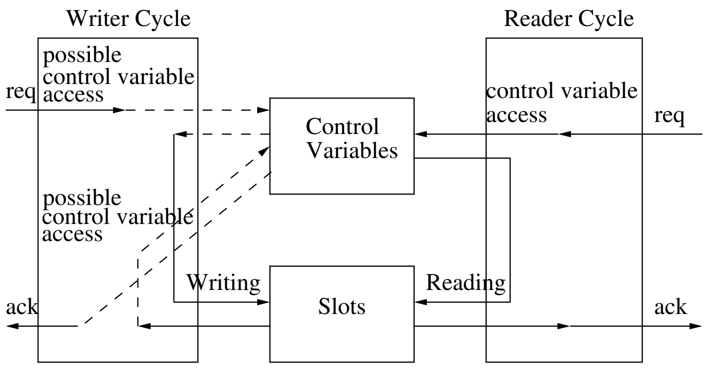


Figure 4.28: General ACM Scheme

of the cycle before or after the access to the data slot. In the reader, the access to control variables is always before access to the data slot. The reason is that the reader must read the latest data item to maintain the freshness property, and the information about which data slot contains the latest data item is indicated by one of the control variables.

As can be seen from Figure 4.28, there are three pairs of data (information) exchanges in each cycle: requesting from and acknowledging to environment, acquiring and updating control variables and accessing slots. A request from the environment indicates the start of a cycle, and an acknowledgement to the environment indicates the end of the cycle. The accessing of the control variables and of the memory slots are within the first request and acknowledgement pair. These three pairs apparently can be modelled with three handshakes. However, since a bit is the smallest granularity possible in a digital system, it is not necessary to use handshakes for the second pair, of control variables. Therefore, only two handshakes in each cycle are enough for modelling, and one is embedded within the other.

4.4.5.2 Controller Modelling

To model ACMs in Stateflow, the states in the general ACM scheme must be identified. The process of a cycle is simplified into two states: idle, and accessing (either reading or writing) slots. Accessing control variables is regarded as an atomic event which takes no time.

A process (reader or writer) stays in an *idle* state until a request comes ([*req==1*]) from the environment. The request leads to the process going into the *processing* state. When reading/writing is finished, an acknowledgement (*ack=1*) is sent back. When the request is withdrawn ([*req==0*]) from the environment, the *processing* state clears the acknowledgement (*ack=0*). The process goes back to the *idle* state accordingly. This indicates the first handshake pair (shown in Figure 4.29).

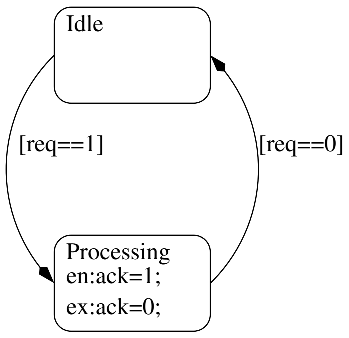


Figure 4.29: A Cycle for a Process

The *processing* contains a set of operations including referencing or setting control variables and accessing memory slots. Access operations on control variables are modelled in transitions. References are put in transition conditions, and settings are put in transition actions. From the environment point of view, the processing is a passive part which would be triggered by a request. On the other hand, from the memory point of view, the processing is an active part which would trigger an access by sending a request. Here the active part is treated as the information sender in a handshake, while the passive part is regarded as the receiver of information. Therefore, the processing has a receiver side of a handshake with the environment and a sender side of a handshake with the memory as shown in Figure 4.30. The condition [*access_ ack==1*] is an acknowledgement for releasing the *access_ req*, while at the same time, it is also a trigger for generating the acknowledgement to the environment.

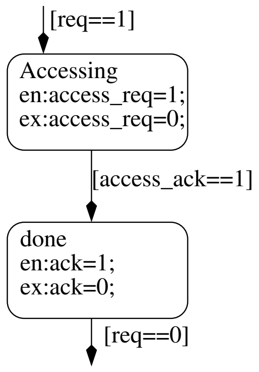


Figure 4.30: Inside the processing

Including Figure 4.30 into the *processing* state in Figure 4.29, this gives Figure 4.31, a full view of a cycle for a process with two handshakes. The default transition points to the *idle* state, which indicates that the *idle* state is the initial state. When the process receives a request, it moves from the *idle* state to the *accessing* one. Once the *accessing* state is entered, an entry action is executed, which sends an access request (*access_ req*) to the memory. As soon as an access acknowledgement is received (*access_ ack*), the active state moves out from the *accessing* state, while producing the exit action to withdraw the *access_ req*, into the *done* state, executing the entry action to acknowledge the completion of the process to the environment. When the environment withdraws the request (*[req==0]*), the *done* state executes the exit action which clears the acknowledgement, and then becomes inactive. The active state goes back to the *idle* one.

Accessing control variables may be done in any of the three transitions according to the specific algorithm.

An ACM algorithm has two cycles, a reader and a writer, running concurrently. It is modelled as two AND states, each of which contains a group of OR states. The contents of the AND states are the model of a cycle shown in Figure 4.31. This is

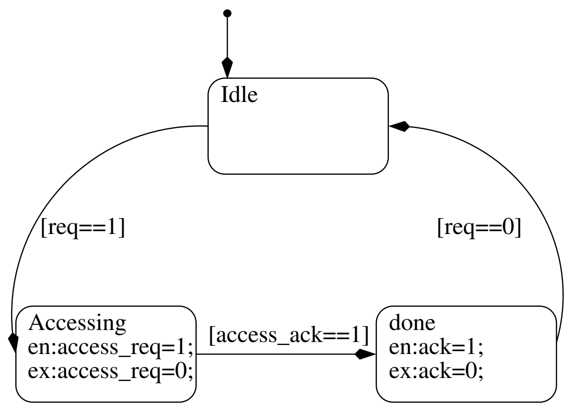


Figure 4.31: A Cycle with Two Handshakes

demonstrated in Figure 4.32. The *rd_ start* and *wr_ start* in this figure are equivalent to the *access_ req* in Figure 4.31, and *rd_ done* and *wr_ done* are equivalent to the *access_ ack*.

After modelling control actions with the graphic objects, the data objects have to be decided. Any of the variables appearing in the chart must be represented as a data object with corresponding data properties. The scope of a data object is determined by where the data comes from. The scope of the data coming from the outside of the controller is set to **Input from Simulink**. If a data item is generated internally and referenced externally, its scope is set to **Output to Simulink**. Others, which are only used internally, are set with a **Local** scope. The inputs of the controller include the reading and the writing requests from the environment (*r_ req* and *w_ req* in Figure 4.32) and the acknowledgements from the memory (*rd_ done* and *wr_ done*); the outputs include the acknowledgements to the environment (*r_ ack* and *w_ ack*), the requests to the memory (*rd_ start* and *wr_ start*) and the slot indices for reading and writing (r and w) which are control variables not shown in the figure. The block appears as shown Figure 4.33.

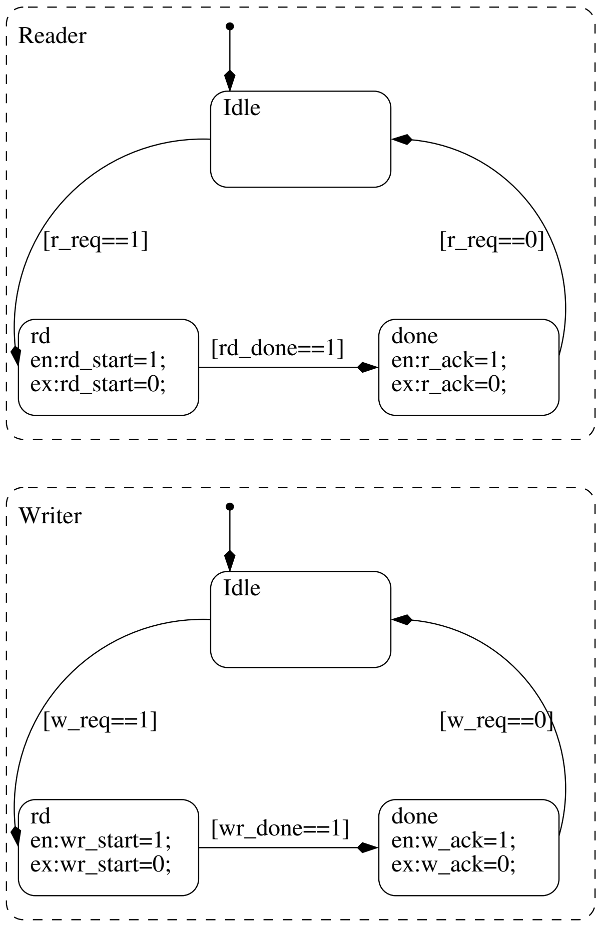


Figure 4.32: Controller Model of a General ACM

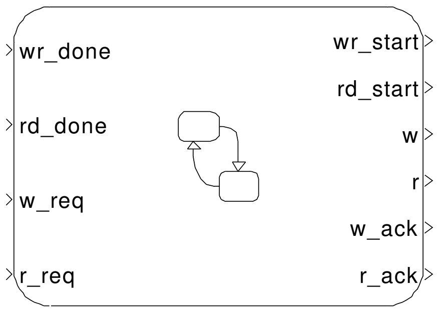


Figure 4.33: Control Block

4.4.5.3 Model of 1-Slot Datapath

The general technique for datapath modelling stated in Section 4.4.2.2 is only for the datapath containing 2 or more slots. However, a one slot datapath is also very useful in some extended ACMs which will be introduced in the following sections.

Basically, a one slot datapath is made up of two connected switches as illustrated in Figure 4.34.

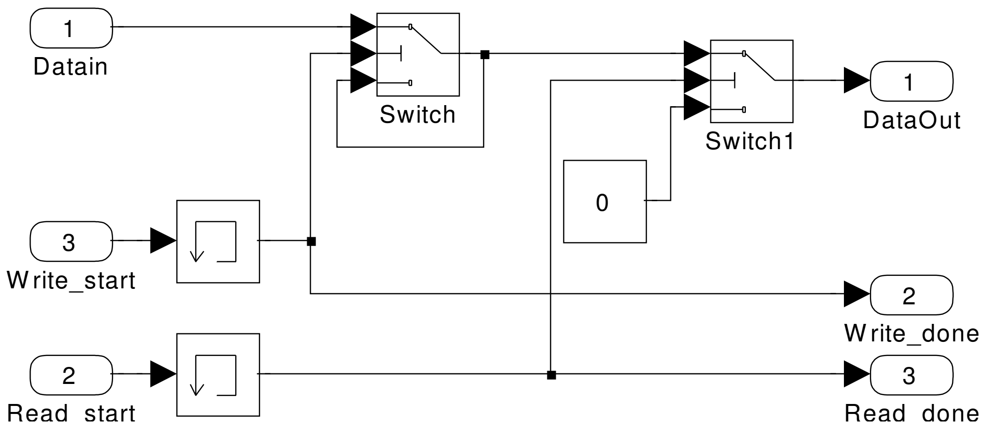


Figure 4.34: 1 Slot Memory

The first switch is used for the writer: data is sent to the first data input of the switch; the second data input is connected back to the output of the switch itself to keep the output stable, in other words, to hold the data item, when the request *w_start* is withdrawn. The *w_start* signal is fed to the control port of the switch.

The second switch is used for the reader: the output of the first switch is sent to the first data input of this switch. The second data input is connected back to the output of the switch for holding the final value or connected to zero depending on the requirement. The *read_start* variable is fed to the control port of this switch.

In the switches, the criteria for passing the first data input is that the control input does not equal to 0. This also applies to all the normal switches in this chapter.

The two memory blocks are used to simulate the process delay in the data path.

This is how it works: when a *w_start* comes (*w_start*==1), the first switch passes the first data input, the data item, to its output. When the *w_start* is withdrawn, the switch holds the data item. With the *r_start* arriving, the second switch delivers the data from the first switch to the output.

There are only three pairs of inputs and outputs in the one slot datapath: *Datain* & *Dataout* pair, *w_ start* & *w_ done* pair coming from and sending to the write cycle and *r_ start* & *r_ done* pair coming from and sending to the read cycle. Because there is only one slot in the memory, the slot indices are not needed any more.

The design of datapath only relies on the number of slots in memory. No matter what types they are, the ACMs with the same number of slots share the same memory structure. Therefore, the datapath models described in this section could be regarded as standard components in a library for reuse.

4.5 Buffered ACMs

The ACMs modelled in the previous section were based on the assumption that only one data item was transferred within the ACM. However, since, in general, the data being transferred consists of a stream of items of the same type, sometimes buffering in the ACM is useful. This section describes an investigation of ACMs which contain buffers. The writer and the reader processes are also assumed to be single-thread loops, during each cycle of which a single item of data is transferred to or from the ACM.

4.5.1 Classification for Buffered ACM

Following the template of 2*2 matrix classification schemes found in Chapter 2, the buffered ACMs are also classified into 4 types as shown in Table 4.4.

	NRR	RR
NOW	BB	RR-BB
OW	OW-BB	OW-RR-BB

Table 4.4: Buffered ACM Classification

A BB or bounded buffer without overwriting and rereading provisions, which includes most traditional inter-process data buffering schemes, may require either process to wait under certain circumstances. An RR-BB may require the writer to wait when previous data items have not been read. An OW-BB may require the reader to wait when no newer data has been made available by the writer after the previous read. An OW-RR-BB, however, does not require either side to wait under any circumstances.

Within each type, the size of the buffering in an ACM determines the quantitative inter-process asynchrony between the writer and the reader. Intuitively, the larger the buffer size, the more inter-process asynchrony there will be (i.e. the longer it will take before either one of the processes may be required to wait). Larger buffer sizes, however, increase latency. It can therefore be said that buffer size is a tool with which quantitative asynchrony may be traded off with data freshness [Sim03].

In addition, for those types where overwriting and/or rereading are permitted, larger buffer sizes reduce the frequency of overwriting and/or rereading. This smoothes the data flow when the relative speeds of the writer and reader fluctuate, and ensures that more of the items out of the writer eventually reach the reader. For example, A *Pool* is an example of when data freshness always seems much more important than data continuity. However, if some degree of non-fresh is permitted, a buffered *Pool* (OW-RR-BB) could be used to reduce the rates of re-reading or over-writing or both, and, therefore, increase data continuity. The qualitative non-blocking still exists on both sides.

In general, ACMs, by focusing on inter-process data communication asynchrony, provide the system designer of the future with a tool to satisfy difficult time-domain requirements across the system qualitatively. Adding practical methods of developing arbitrary buffer sizes for ACMs further enhances their attraction and provides users with a measure to quantitatively refine inter-process asynchrony in system and network design.

In this section, an RR-BB ACM is taken as an example to investigate the model and the properties of a buffered ACM.

4.5.2 Global Model for an RR-BB ACM

4.5.2.1 Ring structure

A Bounded Buffer (BB) ACM can be implemented with a ring structure formed by identical memory cells, as shown in Figure 4.35. One cell stores one data item at a time. The cells can be added or removed to change the size of the buffer. The two arrows in the figure indicate the reader pointer and the writer pointer. Each pointer points to the cell which is being accessed by its corresponding process. After the completion of a data access, the reader and the writer pointers are moved forward according to the specific algorithm.

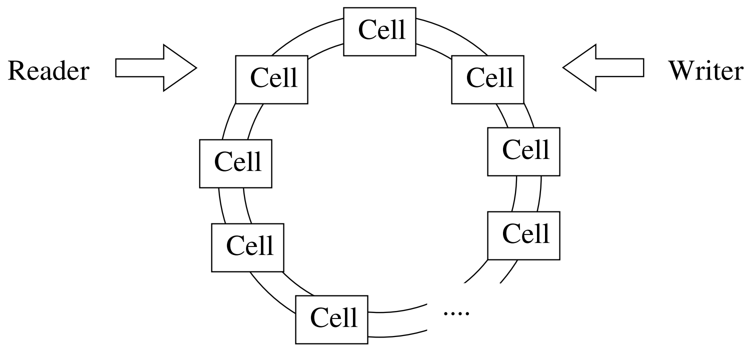


Figure 4.35: Ring Organisation of ACM Buffer

If the writer cycle is much longer than that of the reader, its pointer may point to the cell immediately ahead of the reader pointer. In this case the buffer is empty, i.e. all the data items in the buffer have already been read by the reader. Conversely, if the writer cycle is much shorter than the reader cycle, its pointer will likely point to the cell just behind the reader pointer. The buffer is full in this case and none of the data items in the buffer have been read by the reader.

Rereading, if permitted, only occurs when the buffer is empty with a new read request arriving. Overwriting, if permitted, only happens when the buffer is full with a new write request coming. The RR-BB ACM allows rereading but not overwriting.

4.5.2.2 Algorithm

Derivation of the algorithm can also be found in one of my papers [XHC+06].

Figure 4.36 shows the basic state graph specification considering only the reading and writing accesses.

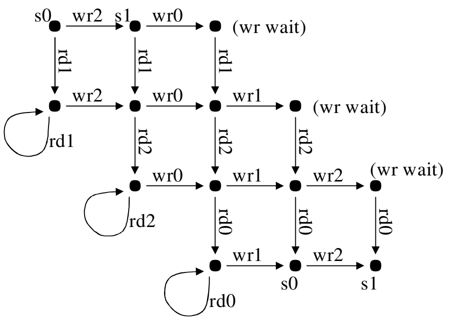


Figure 4.36: Basic State Graph of 3-Cell RR-BB

In Figure 4.36, rd$i$, i=0,1,2, indicates reading access of cell i, and wr$j$, j=0,1,2, indicates writing access of cell j. The two s0 nodes denote the same state. This is also true for the s1 nodes. The entire graph is therefore cyclic. It can be seen that the reader is never forced to wait whilst the writer will wait when the reader is accessing the cell it is scheduled to access. The 3 cells permit each process a maximum of three data accesses before needing to reread or wait. This is greater inter-process quantitative asynchrony than provided by a 2-cell RR-BB. This state graph assumes that rereading accesses the latest data item in the ACM.

With adding the “silent actions” indicating the decision making about whether to wait in the writer or to re-read in the reader, Figure 4.36 is extended to Figure 4.37.

In Figure 4.37, λ_{ij} indicates the silent action which advances the writer from cell i to cell j, and μ_{kl} indicates the silent action which advances the reader from cell k to cell l or prepares for the rereading of cell k if k=l.

Based on Figure 4.37 which can be extended to the n-cell case, an algorithm for the n-cell RR-BB ACM was derived as shown in Algorithm 4 [XHC+06]. This is

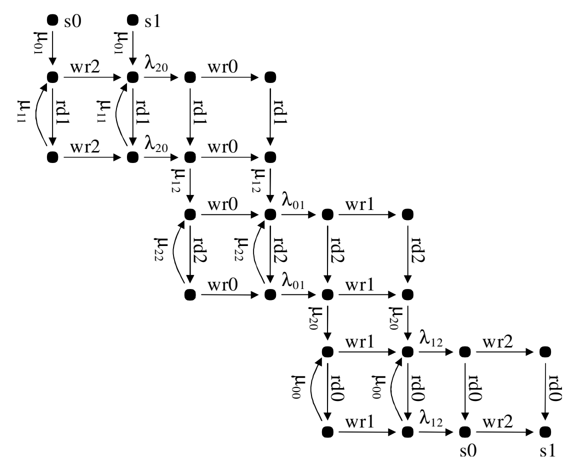


Figure 4.37: State Graph of 3-Cell RR-BB with Silent Actions

based on the synthesis method purposed in [YX01].

Algorithm 4 n-Cell RR-BB ACM Algorithm

```
Writer                          Reader
wr: write cell $w$;             r0: if ($r$+1 mod $n$)$\neq w$ then
w0: $w$:=($w$+1 mod $n$);               $r$:=($r$+1 mod $n$);
ww: wait until $r\neq w$        rd: read cell $r$;
```

Here w, $r \in \{0, 1, 2, ..., n\text{-}1\}$, and n is the total number of the memory cells in the ring. Statements *w0* and *ww* together constitute λ, and statement r0 is μ. "*mod*" in the algorithm is an operator to find out the remainder of the two arguments.

4.5.2.3 Modelling

With the algorithm, the MATLAB model can be built by using the method described in the previous section. It can be seen from the algorithm that 3 information exchange pairs are in both the reader and the writer. The first one is the exchange between the controller and the external processes, i.e. the environment. In this pair, the external environment is the active side and the controller is passive; the second one is the exchange between the controller and the datapath. In this pair, the controller is active and the datapath is passive; the third is the exchange of control variables. They are between the reader and the writer. The variables are assigned by one side and referenced by the other.

Previous studies [XC00] have shown that when all the actions, including the data access actions and the silent actions, are regarded as non-atomic, none of the ACM implementations work according to specifications if fundamental mode assumptions do not apply. However, some ultra-safe implementations have been found that work correctly, under fundamental mode assumptions, even when atomicity is assumed at a lower level, such as the beginning and the end of a control variable set or read [XC02, Cla00]. The exchange pairs of control variable actions are regarded as atomic here. The other two exchange pairs are modelled with handshakes.

Figure 4.38 shows the Stateflow model for Algorithm 4. As the active side in the information exchange with the datapath, a handshake sender model, *writing* state in the writer and *reading* state in the reader, is used to send the access request to the datapath. As the passive side in the information exchange with the external processes, a handshake receiver model, *done* state in both the reader and the writer, is used to acknowledge to the environment the completion of the data access. The *idle* state represents waiting for the next cycle request from the external processes. The ww statement is merged into the *w_ idle* state because it is also conditional waiting. The two wait conditions are AND-ed to produce the equivalent result.

From the algorithm, *w* is updated after write access. It is then implemented in the transition action after the *writing* state. The if expression in *r0* is modelled as two transitions from a junction to the *reading* state. One of the transitions has the transition action for updating *r* attached. A graphic function is used to define the "mod" function in the transition actions.

The writer will not become active until the write request comes and *w* is not the same as *r*. When the writer becomes active, a *write_ start* signal is sent to the buffer, in order to write the new data item to the corresponding cell. When the writer receives a *write_ done* signal from the buffer, indicating the completion of *wr*,

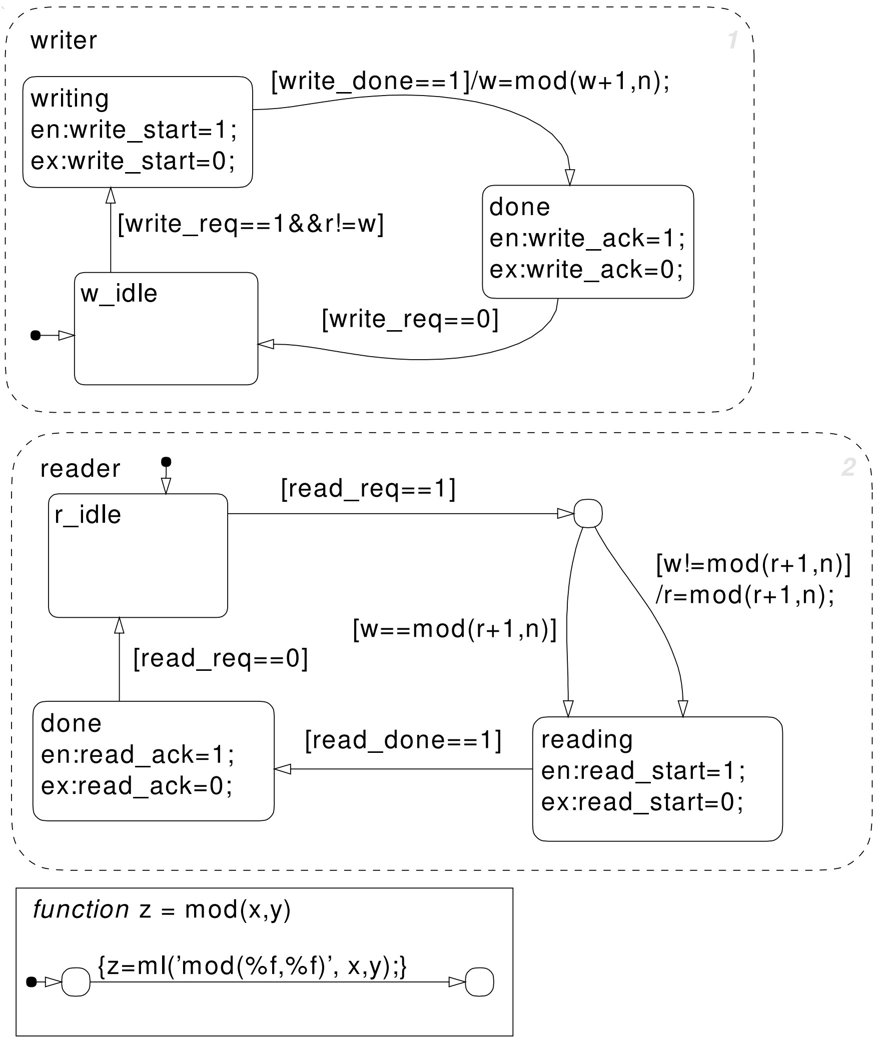


Figure 4.38: Stateflow Model for a General RR-BB

it will move the writer pointer to the next cell. To finish this operation, the writer needs to check if the current cell is the one with the highest index. If it is, the writer pointer will be moved to the cell number 0 (set w to 0), which is the cell with the lowest index. Otherwise, it will increment the value of w. After that, a *write_ ack* is sent back to the environment. Then the writer will wait for the releasing of the *write_ req* before going back to the *w_ idle* state.

The reader is similar to the writer. When a read request comes from the environment, the reader will check if the next cell is occupied by the writer or not. The same i+1 mod n exercise is carried out to determine the index of the next cell (either i+1 or 0). If the next cell is occupied by the writer, the reader pointer will remain at the current cell. If not, the reader pointer will be moved forward according to the i+1 mod n rule. Then the reader sends a *read_ start* signal to the buffer in order to read the data item in the corresponding cell. On completion of reading, the reader will receive a *read_ done* signal from the buffer. A *read_ ack* is sent to the environment, and then the active state moves to the *r_ idle* state waiting for the next *read_ req* signal.

The variables which appear in the model include:

n: the number of slots in the ACM. It is an integer entered by the designer;

write_ req and *read_ req*: the requests for the writer and the reader to start a cycle. They are inputs from the environment;

write_ ack and *read_ ack*: the acknowledgements from the writer and the reader indicating the completion of a cycle. They are outputs to the environment;

write_ start and *read_ start*: the data access requests. They are outputs to the buffer;

write_ done and *read_ done*: the completion signals of data access. They are inputs from the buffer;

w and r: the indices for the writer and the reader. They are local variables and initialised to the different integers within 0 and n-1, normally, w=1 and r=n-1. In the start-up time, the buffer should be empty. To satisfy this condition, these two variables must be initialised to adjacent slots, and w=mod(r+1,n).

The buffer of an n-cell RR-BB ACM is the same as the n-slot memory which can be extended from the 3-slot one modelled in Section 4.4.2.2.

4.5.2.4 Simulation Results and Discussions

Plugging a 3-cell RR-BB ACM into the test environment model, its resulting waveforms are shown in Figure 4.39. Rereading occurred when read requests came without new data items being available, as in the case after the data items 4, 7, 9 and

10 were read. In this simulation, n was set to 3, therefore, the writer waited if two consecutive data items had not been read, as in the case after items 12 and 13 were written.

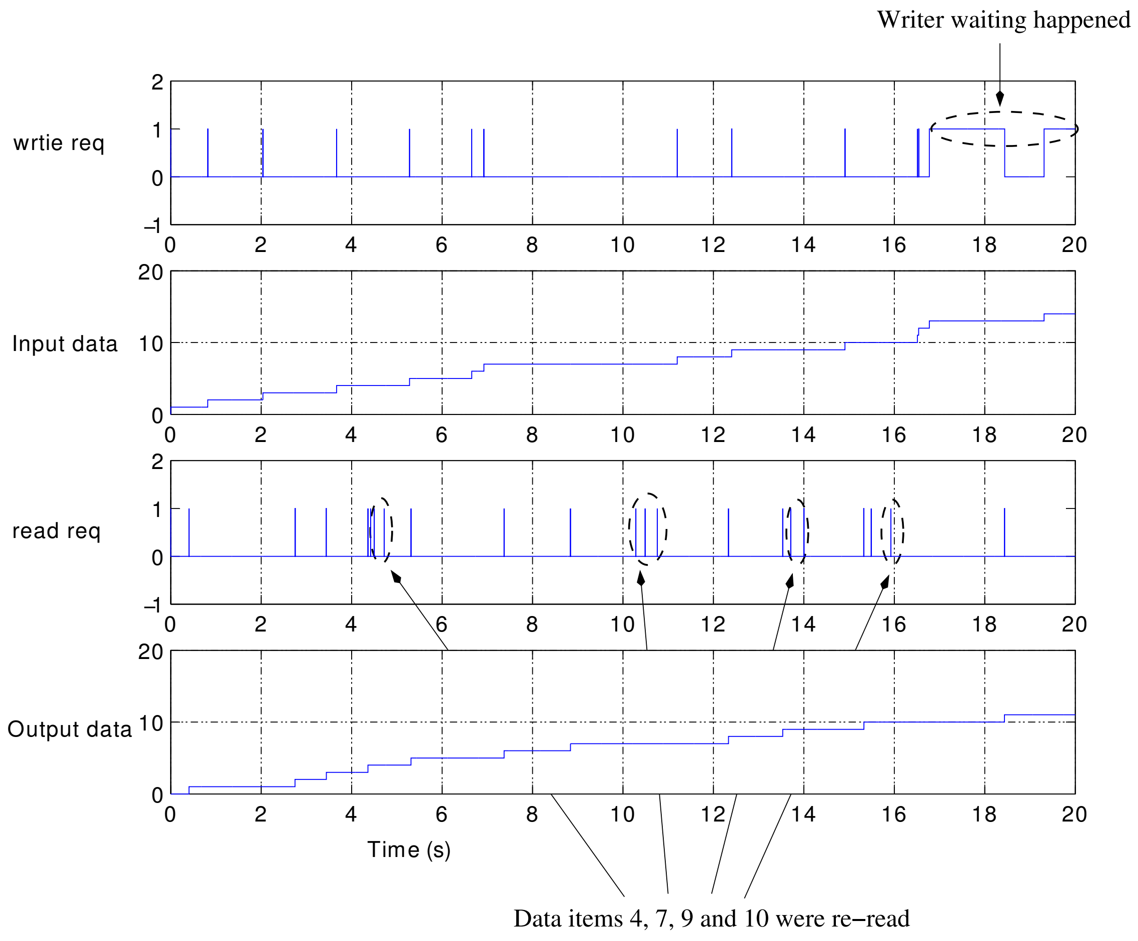


Figure 4.39: Simulation Result for 3-Cell RR-BB

Algorithm 4, though neat and easily understandable, is not suitable for hardware implementation. In particular, the integer control variables w and r will need many protections in order to be considered atomic. The global view nature of the indexing also means that the actual setting and reading of these variables will include multiplexing and de-multiplexing on a scale depending on the number n. The fork and join operations needed mean that an implementation of n cells, for instance, cannot be easily built upon one of n-1 cells.

4.5.3 Modular Design Model for an RR-BB ACM

The cellular structure of these kind of buffered ACMs suggests that it may be possible to construct a standard individual cell, including its own controller and datapath, complete with its own local control variables, and then use n of these for an n-cell solution. This modular design approach is much better suited for hardware implementations.

4.5.3.1 Algorithm for Modular Design RR-BB

Considering a single cell within the ring structure of Figure 4.35 and Algorithm 4 and observing the part of the state graph in Figure 4.37 pertaining to a particular cell, it can be seen that a single cell element in an n-cell ACM needs to have two local index control variables, w and r. Both these variables will take binary values and the value 1 would indicate that the pertinent process is pointed at this cell. A cell, particularly its control variables, will be visible to its immediate preceding neighbour but not any other cell.

A localised algorithm for a single cell is described in Algorithm 5.

Algorithm 5 Algorithm for Modular Design RR-BB ACM

```
Writer:                          Reader:
wr: write;                       r0: if $wnext$=0 then
w0: $w$:=0; $wnext$:=1;          begin $r$:=0; $rnext$:=1;
ww: wait until $rnext$=0;        advance to next rd; end
wa: advance to next              rd: read;
```

The action of advancing to the next cell in the writer causes the end of execution of the current cell's writer algorithm and the beginning of the next cell's one from *wr*. The action of advancing to the next rd in the reader causes the end of execution of the current cell's reader algorithm and the beginning of the next cell's one from *rd*. The reader algorithm loops at the same cell until the condition $wnext$=0 is met. The writer will wait at a cell until the condition $rnext$=0 is met. Note that the writer algorithm sets both w and *wnext* and reads *rnext*, and the reader algorithm sets both r and *rnext* and reads *wnext*.

It is then a straightforward process to construct a single cell template for hardware, a number of which can then be connected into a ring of the desired number of cells. Figure 4.40 shows how two adjacent cells were connected.

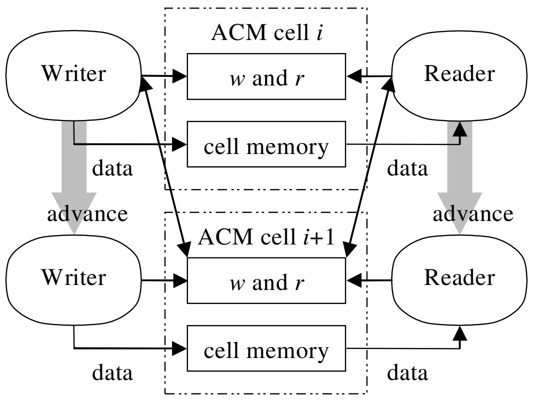


Figure 4.40: Connection Between Two Cells

4.5.3.2 Modelling

Two more information exchange pairs are in the writer/reader in addition to the three mentioned in the previous algorithm because of the existence of the action “advance to next”. One is the action itself; the other is the response for the action from the previous cell. Furthermore, w and r are also exchanged between two adjacent cells. Besides being cleared and referenced in the current cell, they are also set and referenced by the previous cell. Therefore, there are 6 information exchange pairs altogether in each cycle of the algorithm.

Obviously, the information exchanges between the ACM and the environment, between the controller and the datapath, and between two cells should be implemented by handshakes. The operations on local variables are treated as atomic and can be modelled in the condition actions.

The handshakes for the first two types of information exchanged have already been discussed in the previous chapter. Here, only the handshakes between cells are considered. In the handshake for the "advance to next" action, the current cell is active, and the next cell is passive. In the handshake for the response to the advance action, the current cell is passive, and the previous cell is active. In the handshake for the action of setting a variable in the next cell, the current cell is active, and the next cell is passive. The setting action is atomic so the passive part in this handshake is implemented in the condition actions. Figure 4.41 shows the Stateflow model of a RR-BB modular design cell.

The *reset* states in Figure 4.41 are used to initialise the position of the pointers. If the pointer moves to the current cell, the system is in the *ready* state, otherwise, it is in the *idle* state.

In the writer, once a write request comes, it sends an access signal (*write_ start*) to the datapath. After the write operation is completed, a completion signal is sent back to the writer. On receiving the completion signal, the writer updates variable w to 0 and sends the request to set w in the next cell. Once the next w is set, the writer delivers the acknowledgement to the environment. The writer will not advance to the next cell until r in the next cell is 0. Then the current writer goes to the idle state. Although the writing process is completed, the writer still needs to respond to the request for setting w (implemented in the transition after the *w_ idle* state) and the advance action (in the *w_ ready* state) from the previous writer before it is active for writing again.

In the reader, if it is in the *r_ ready* state at the beginning, when a read request comes, it checks whether the next cell is occupied by the writer ([*w_nxt*==1]). If the next cell is free from the writer, the reader clears its r, sets the next r, and advances to next cell, then the current reader goes to the *r_ idle* state. Otherwise, the reader sends the access request to the memory. On receiving the completion signal from the datapath, the reader acknowledges the environment, and then goes to the *r_ ready* state. If the reader is in the *r_ idle* state at the beginning, it sets

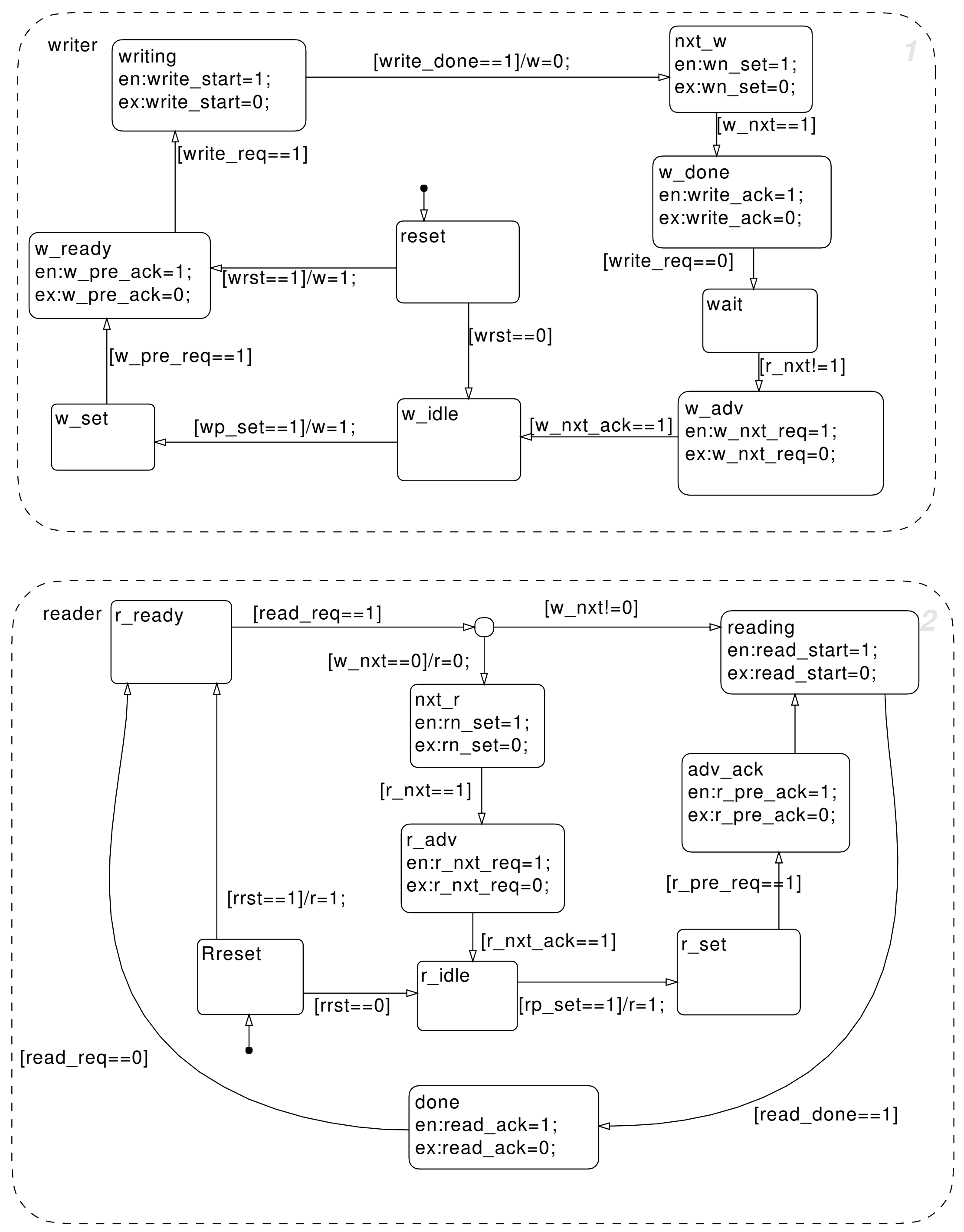


Figure 4.41: Stateflow Model for RR-BB Modular Design

the current *r* when it receives a request from the previous cell, and comes to the *r_set* state. At the same time, the advance request comes to the reader from the current cell. The reader acknowledges the request, and then reads the data item in the current datapath.

The advance action in the writer is at the end of each cycle when the cell has already been deactivated. That action in the reader is within each cycle which switches off the current reader and turns on the next one within the same read request period.

In this model, the variables include:

write_req and *read_req*: the requests for the writer and the reader to start a cycle. They are the inputs from the environment;

write_ack and *read_ack*: the acknowledgements from the writer and the reader indicating the completion of a cycle. They are the outputs to the environment;

write_start and *read_start*: the data access requests. They are the outputs to the buffer;

write_done and *read_done*: the completion signals of data access. They are the inputs from the buffer;

w and *r*: the indices for the current writer and reader. They are the outputs to the previous cell;

wn_set and *rn_set*: the requests for the indices for the next cell. They are the outputs to the next cell;

w_nxt and *r_nxt*: the indices for the writer and the reader in the next cell. They are the inputs from the next cell;

w_nxt_req and *r_nxt_req*: the advance requests to the next cell. They are the outputs to the next cell;

w_nxt_ack and *r_nxt_*ack: the acknowledgements for the advance. They are the inputs from the next cell;

wp_set and *rp_set*: the setting indices requests from the previous cell. They are the inputs from the previous cell;

w_pre_req and *r_pre_req*: the advance requests from the previous cell. They are the inputs from the previous cell;

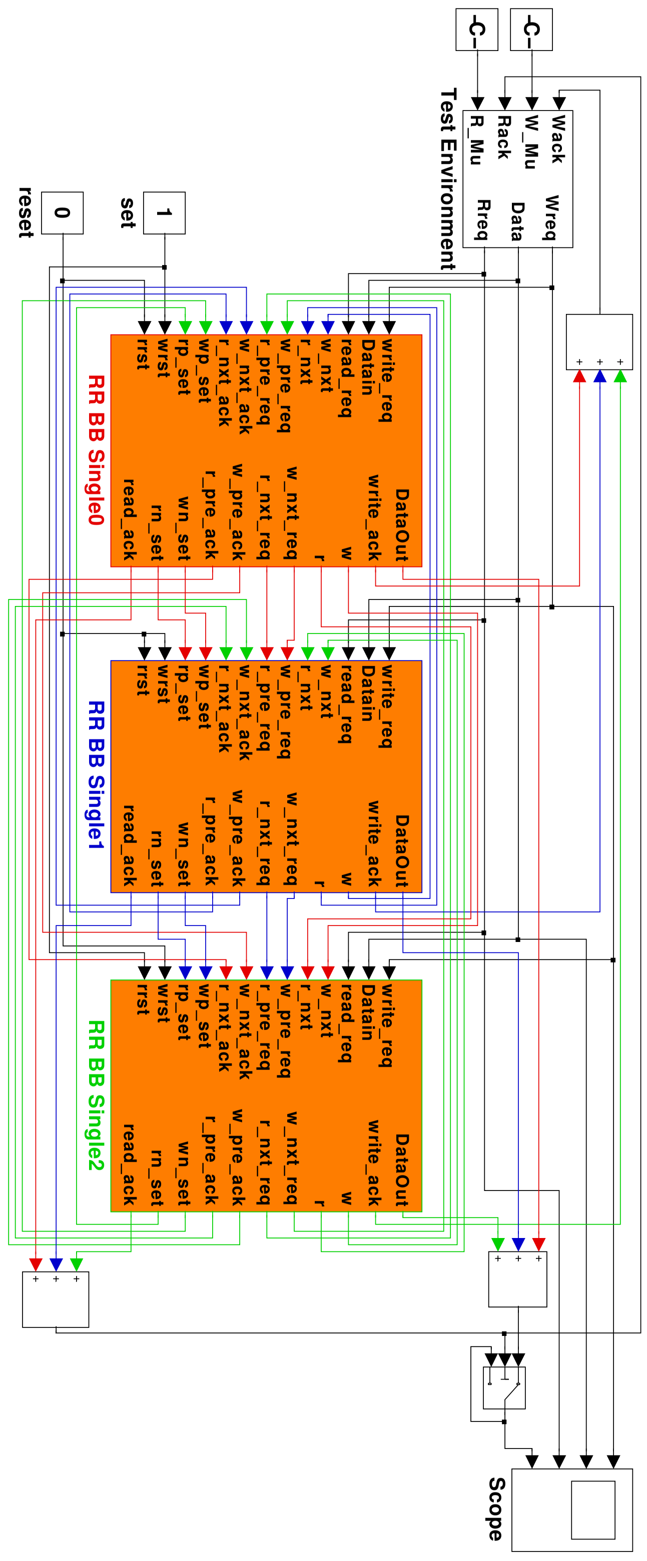


Figure 4.42: Simulink Model for a 3-Cell RR-BB

w_pre_ack and *r_pre_ack*: the acknowledgements for the advance. They are the outputs to the next cell;

wrst and *rrst*: the initialising signals to the writer and the reader. They are the inputs from the environment.

The datapath for the cell is the same as the one-slot one illustrated in Figure 4.34.

A 3-cell model is made up by connecting 3 identical modular design models together as shown in Figure 4.42.

In this model, the first cell (RR BB Single0) is initialised for writing and the last cell (RR BB Single2) is initialised for reading. To distinguish the connections easily, different colours are used for the connection wires. The wires from the environment are in black, those from the first cell are in red; those from the second one are in blue; and those from the last one are in green. The data items, the write requests and the read request are fed in all the three cells. There would be a possible problem in hardware design if the number of cells is quite large because the global input signals could lead to distribution delays. However, in real applications, there would not be a large number of cells in the buffer. All the three data outputs, the write acknowledgements and the read acknowledgements from cells are added to feed back to the test environment or to the Scope. They can be added to represent the overall results because at least two of them are 0 at any time.

4.5.3.3 Simulation Results

When the ACM model is inserted into the test environment described in Section 4.3.4, the resulting waveform for Figure 4.42 is shown in Figure 4.43. Rereading occurred after data items 1, 2, 7 and 12 were read, and writer waiting happened after data items 10, 14 and 15 were written. These maintained the properties specified for the RR-BB ACM.

For both simulations for the global view and the modular design, the time taken by the reader and writer processes outside the ACM was controlled by two inde-

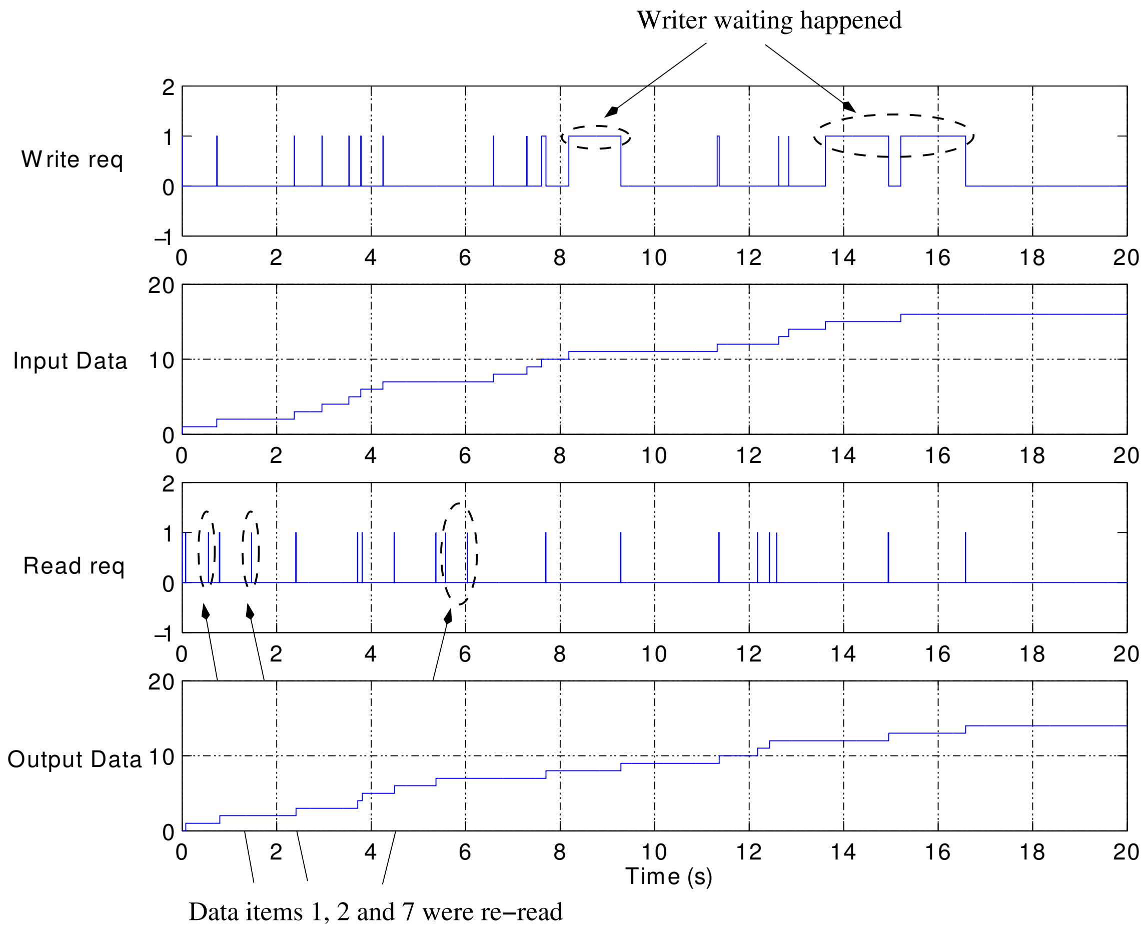


Figure 4.43: Waveforms for 3-Cell Modular Design RR-BB Model

pendent random number generators with normal distribution. This gave enough variation for writer waiting and reader rereading to appear during relatively short simulation runs. As shown in the simulation results, the reader does not always obtain the most up to date data item in the ACM. This is to be expected because of the buffering. However, the order of the sequence is preserved. The simulations do not turn up results contrary to the specifications of the RR-BB ACM.

4.5.4 Discussions

To investigate the other properties, such as data latency and data continuity, the method adopted was to simulate the models several times while changing one parameter out of three each time. The three parameters were: the number of cells, the delivery speed of the reader, and that of the writer. The delivery speeds were controlled by the mean values of the exponential distributions.

During the simulation, the following parameters were monitored:

Nw: the number of items being written;

Nr: the number items being read;

Nww: the number of items being forced to wait for writing;

Nrr: the number of items being re-read;

MLatency: the mean value of data latency (re-read items were not counted). The data latency refers to the time between each data item being ready for writing and it being read by the reader.

W%: writer waiting rate; *Nww*/*Nw**100%.

R%: re-reading rate; *Nrr*/(*Nr*+*Nrr*)*100%.

Mdi: the mean of the numbers of items behind the input ones when items are read. Mdi represents the freshness.

Table 4.5 was obtained by increasing the number of cells while keeping the same deliver speeds.

In the table, *Rmu* and *Wmu* are the mean of the random intervals for the reader and the writer; Global represents that the model used in the simulation was a global view model and Modular represents the modular design model; *N* is the number of cells.

Table 4.5 (a) describes the scenario that the delivery speeds for the reader and the writer were close to each other. The results shows that with increasing the number of cells, *Nw*, *Nr*, *MLatency* and *Mdi* are increased, and *Nww*, *Nrr*, W% and R% are decreased. Although the results from the global view model and that from the modular design one were not exactly the same because of the different internal structures, they still showed the same properties. Apparently, additional cells in the buffer gave more chances to the writer and the reader to write data items to and

Rmu=1.035, Wmu=1.113, Length=1000

	N	Nw	Nr	Nww	Nrr	MLatency	W%	R%	Mdi
Global	3	737	736	218	287	1.329	29.58	28.05	0.9118
	6	829	825	98	198	2.313	11.82	19.35	2.083
	10	853	845	53	178	3.54	6.21	17.4	3.42
	20	896	893	5	130	5.352	0.56	12.71	5.422
	25	915	915	0	108	6.194	0	10.56	6.268
	26	915	915	0	108	6.194	0	10.56	6.268
Modular	3	741	741	219	264	1.404	29.55	26.27	0.919
	6	841	840	113	140	2.817	13.44	14.29	2.263

(a)

Rmu=1.035, Wmu=10.13, Length=1000

	N	Nw	Nr	Nww	Nrr	MLatency	W%	R%	Mdi
Global	3	96	96	0	927	0.096	0	90.62	0.0521
	6	96	96	0	927	0.096	0	90.62	0.0521
Modular	3	94	94	1	865	0.145	1.06	90.2	0.1277
	6	82	81	0	828	0.12	0	91.09	0.1235

(b)

Rmu=10.35, Wmu=1.113, Length=1000

	N	Nw	Nr	Nww	Nrr	MLatency	W%	R%	Mdi
Global	3	97	94	88	0	29.883	90.72	0	1.875
	6	100	94	87	0	58.479	87	0	4.616
Modular	3	102	99	90	3	26.961	88.24	2.94	1.842
	6	106	100	86	0	55	81.13	0	4.524

(c)

Table 4.5: The Properties for RR-BB ACMs When Increasing the Number of Cells, (a) *Rmu=1*.035, *Wmu=1*.113; (b) *Rmu=1*.035, *Wmu=10*.13; (c) *Rmu=10*.35, *Wmu=1*.113

read new items from the buffer. Consequently, the occurrences of the writer waiting and re-reading were decreased, and so did the writer waiting rate and the re-reading rate which indicated better data continuity. With more cells adding to the buffer, each item may stay in the buffer longer. This led to the increase of the data latency and *Mdi* (worse freshness).

From the table, it also could be seen that when N increased to 25, the writer waiting rate dropped to 0. After that, more cells would not improve the performance any more. A conclusion could be drawn from this: if the average speed of the reader is faster than that of the writer, even when the difference is small, when the number of cells reaches a certain number *N*, additional cells will not affect the performance.

Table 4.5 (b) describes the scenario that the delivery speed for the reader was far faster than that for the writer. There was almost no difference for all the variables monitored, especially the global view case. The tiny difference in the modular design case still showed the same tendency to that in scenario (a). Nw was the same as *Nr*, which implied that all the data items that had been written were read, i.e. the ACM was empty. This is straightforward, because whenever a data item was written, it was read by the fast reader immediately. As a result, writer waiting seldom happened (*Nww*=0); writer waiting rate was 0%; data latency is relatively short (no more than 0.01). The reader re-read the most recent item again and again until the new one came. This led to a high re-reading rate (over 90%) and a good freshness performance (*Mdi* were near to 0). In this scenario, the additional cells did not improve the performance significantly.

Table 4.5 (c) describes the scenario that the delivery speed for the reader was far slower than that for the writer. With increasing *N*, there were tiny differences in *Nr*, *Nrr* and R%. However, the *MLatency* and *Mdi* were significantly increased. The result of *Nw*-*Nr* was *N*. This could be explained by the fact that the writer was fast enough to write the data items in all the buffers and wait for more cells being released from the reader. Therefore, the ACM was full almost all of the time, and re-reading seldom happened (*Nrr*=0 and R%=0). For the same reason, when *N* was increased, more data items could be written into the buffers, but all of them had to wait for longer to be read by the reader. This led to an increase of data latency (*MLatency*) and a decrease of freshness (*Mdi*). In this scenario, the additional cells did not improve the data continuity (W% and R%) significantly, but performed even worse in data latency.

Table 4.6 illustrates the performance of a 3-cell buffered RR-BB ACM at different

N=3, Length=1000

Rum	Wmu	Nw	Nr	Nww	Nrr	MLatency	W%	R%	Mdi
10.35	1.113	97	94	88	0	29.883	91	0	1.875
2.035	1.113	451	448	262	35	4.747	58	7.25	1.456
1.035	1.113	737	736	218	287	1.329	30	28.05	0.9118
1.035	2.113	461	461	62	562	0.73	13	54.94	0.5315
1.035	10.13	96	96	0	927	0.096	0	90.62	0.05208

Table 4.6: The Properties for RR-BB ACMs When Changing Delivery Speeds

speeds. The table showed that the simulation results for these two models had exactly the same properties. It showed that with increasing the speed of the reader or decreasing that of the writer, *MLatency*, W% and Mdi were decreasing, and R% was increasing. It could also be seen from the table that both *Nw* and *Nr* were the largest when the speed of the reader and the writer were closest. This could be explained as follows. When the speed of the writer was higher, it had to wait while the buffer was full. Therefore, the data latency and *Mdi* were relatively long and the total number of items written in the buffer was restricted by the reader. When the reader was faster, the faster the reader was, the more times re-reading happened. As a result, R% was high and *Nr* was restricted by the writer. Although W% was decreasing while R% was increasing in these cases, if the average values were taken, it could be found that the closer the speed, the better the data continuity.

4.6 Hardware Implementation for a 3-Slot *Signal*

As claimed before, ACM can be represented by Petri nets. A technique called direct translation [XYS$^+$00, Sha03], also known as direct mapping [SBY03] can be used to create circuits according to the Petri nets. This technique uses a "David cell" [Dav77], a kind of memory element, representing each Petri net place, therefore it produces a straightforward implementation of the PN model in hardware. The transitions were implemented as certain processes between adjacent David cells.

The idea of implementing a 3-slot *Signal* into hardware is to find out its Petri nets representation first and then translate it into circuits by direct translation.

4.6.1 Block Diagram

An ACM is made up of control circuits and a data path. The control circuits are used to determine the value of each control variable according to the Petri net specifications described in the previous section. For a 3-slot *Signal* type ACM, as mentioned before, to maintain the data freshness and data coherence, the *w0* and *w1* pair must not be executed with *r0* simultaneously. A Mutex, which will be introduced in the following section, plays this role. It connects with both the reader and the writer to hold one request until the other one is completed if the two requests come very close. The datapath has 3 data storage elements which can be accessed by both the reader and the writer according to the control variables.

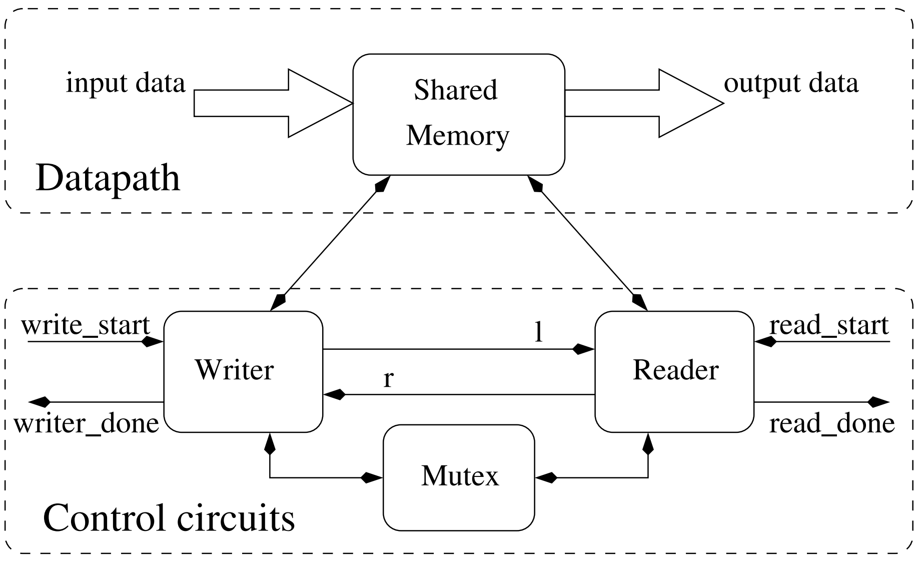


Figure 4.44: Block Diagram for 3-slot *Signal*

Figure 4.44 shows the block diagram for a 3-slot *Signal*. When the writer receives a *write_ start* signal from the environment, it requests to access the shared memory for writing the input data item into the chosen slot. After the write operation is completed, the writer sends a request to the Mutex in order to check if the *r0* statement is executing. When the Mutex gives it the grant, it updates *l* and then according to *r* determines a new value of *w*. When all these processes are completed, a *write_done* is sent back. When the *read_start* signal comes to the reader, the reader sends a request to the Mutex. Once it receives the grant, it decides to wait or update the control variable *r* according to the current value of *l*, and then reads the data item in slot *r*.

The core components used within the 3-slot *Signal* implementation include David cell (DC), Mutual Exclusion element (Mutex), SYNC Arbiter, Multiplexer and some latches etc. It also needs to be mentioned that this design is based on a dual rail and low level active mode.

4.6.2 Implementation of Control Circuits

The control circuits are split into a reader and a writer. Each of them is connected to the Mutex.

The statement of *w1* is implemented by an L-Latch. The function of the L-latch is to determine and hold the variable *l* according to the current value of *w*. As stated before, to maintain the data freshness and the data coherence, the *w0* and *w1* statements are put together as a consecutive pair. Therefore, after *l* is set, a completion signal is sent to the subsequent component. Figure 4.45 shows the L-latch for this design. The inputs of this latch are active low, while the outputs are active high.

The inputs are one-hot signals, that is, only one of the three inputs is active at a time. The *clear* input, connected to the second complex gate, is used to reset the circuit when the system starts up. The input named with *set n* means setting *l* to n. The output named with *l_n* indicates that the current value of *l* is n, and *setn_done* is the completion signal.

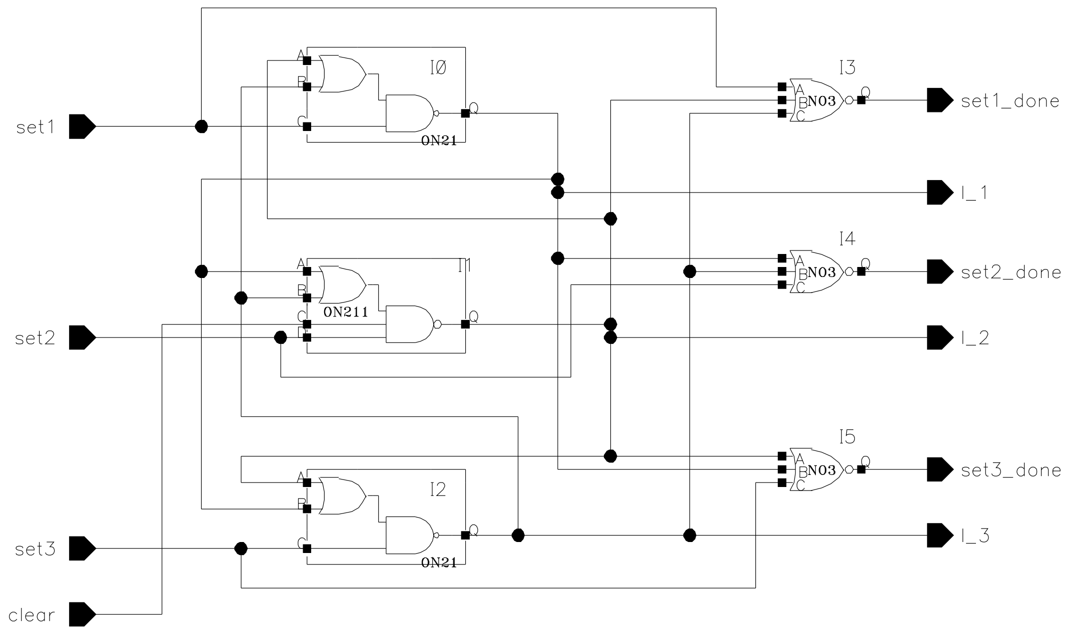


Figure 4.45: L-Latch

Initially, all the *set* inputs are high. A *clear* pulse sets the output of the second complex gate (*I1*) to 1. Consequently, *l_2* becomes high, which indicates that *l* is initialised to 2. At the same time, it is fed to the input A of both the other two complex gates (*I0* and *I2*), which makes their outputs set to low. These two low signals are connected back to inputs A and B of *I1*, which keeps its output low regardless of the *clear* signal.

The inputs are one hot signals, so there will only be one input out of three active. Once a *set* signal arrives, it sets the correspondent *l_n* and *setn_done*. The output of the complex gate is fed to one of the first two inputs of the other two complex gates, which resets their outputs. These outputs also fed back to its inputs A and B to hold its output regardless of whether the *set* is withdrawn. The release for the *set* signal only leads to a reset of output *set_done*.

The signal flow for the writer is illustrated in Figure 4.46. It is translated from the Petri net in Figure 4.3. The figure only shows one branch for the Petri net. However, the other two branches are exactly the same. In the Petri net, each branch starts

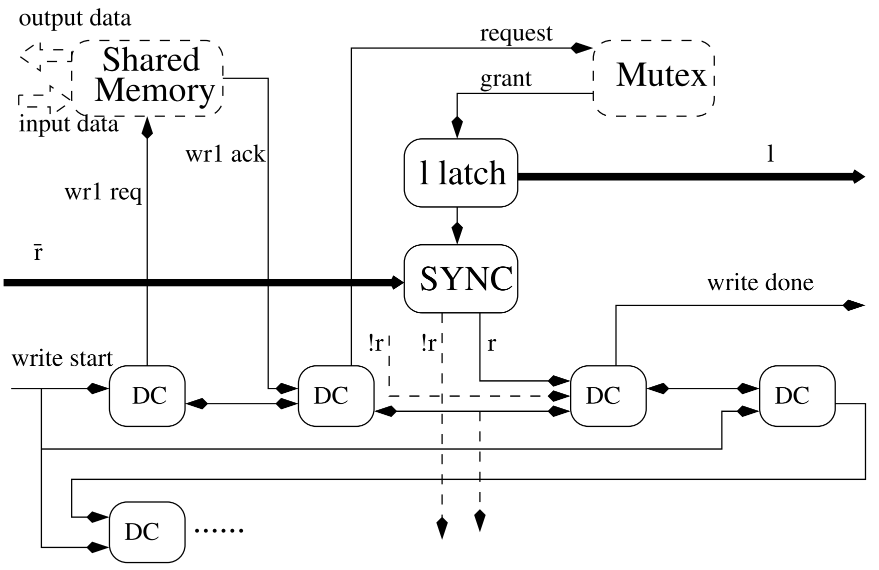


Figure 4.46: Signal Flow for the Writer

from the *startn* state, ends at the *idlem* state, and contains 4 states. Consequently, one branch in Figure 4.46 contains 4 DCs. They are translated from the states *startn*, *wrn_ done*, *w=m* and *idlem*. The transition *wrn* is the write access for slot *n*. It is implemented by sending an access request from the first DC to the shared memory and receiving the completion signal at the second DC. Transition *wnm* is implemented by the circuits between the second and the third DC. The second DC sends a request to the Mutex. The grant from the Mutex is regarded as the set signal to the L latch. When the value of *l* is stored, the completion signal is fed to the SYNC as the clock. According to the current value of *r*, the SYNC chooses one branch to deliver a request to the third DC that determines the slot to store the next new data item. As the third DC receives the request, it acknowledges the second one and then sends a *write done* signal to inform the environment of the completion of the write cycle. Once the environment releases the write request, the fourth DC will be informed and then acknowledge the third DC to withdraw the completion signal.

The R latch is used in the reader for determining and holding the control variable *r*. Figure 4.47 shows the circuit for the r latch. It has a clear input and six *smn* inputs where *m, n* =1, 2, 3, which indicates set *r* from m to n. The outputs include three pairs of dual rail signals and six completion signals. The inputs except the clear are active high and the outputs are active low. To convert the inputs into active low, six inverters are added.

The main part of the circuit is made up of three modified RS flip-flops. The clear signal sets the first and the third flip-flops and resets the second one, that is, *r* is initialised to 2. Once an *smn* signal arrives, it resets the corresponding flip-flop *n*. The output *rn* (low) conjugating with input *snm* (low) and output *nrk* (low), where *k* is neither *m* nor *n*, sets the flip-flop *m*, and conjugating with input *snk* (low) and output *nrm* (low) keeps the flip-flop *k* set. With the complete resetting of flip-flop *n* and setting of flip-flop *m*, a completion signal *dmn* is generated.

For one flip-flop, say *n*, both the signals *smn* and *skn* are able to reset it. Therefore, each flip-flop has two reset inputs, which are implemented with multi-input NAND gates. Because resetting for either of the other two flip-flops will cause setting for the current one, there have to be two set inputs for each flip-flop as well. As the set signal is a conjugation of three low signals, two OR gates are used as the first stage of the complex gate in each flip-flop.

The signal flow for the reader is illustrated in Figure 4.48. The four DCs in this Figure are translated from one branch in the Petri net specification for the reader. The *Comp* is a component to compare if two inputs are the same. Here, it is used to compare the current value of *l* and *r*. If they are not the same and DC1 sends a request, the request will be passed to the Mutex through a C element. The grant from the Mutex sends to the R latch the set signal to update control variable *r*. When updating is completed, the completion signal is delivered to DC2 as a request. This DC informs the first one and then generates the read access request and delivers it to the shared memory for reading the data item in the corresponding slot. After finishing reading, the completion signal triggers DC3 generating a *read*

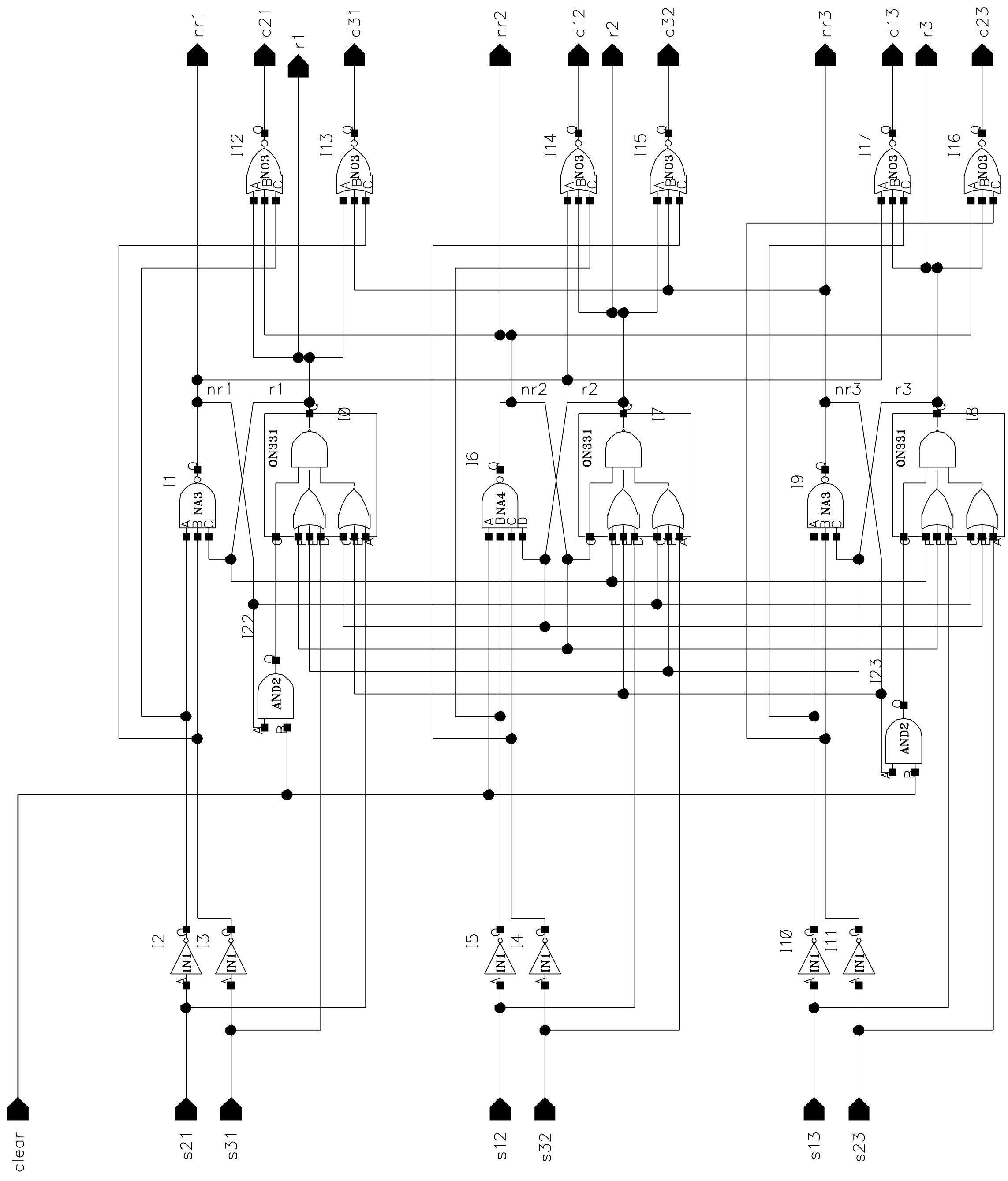


Figure 4.47: R-Latch

done for acknowledging to the environment the completion for the read cycle. When the read request is released by the environment, DC4 acknowledges DC3 to withdraw the *read done* signal.

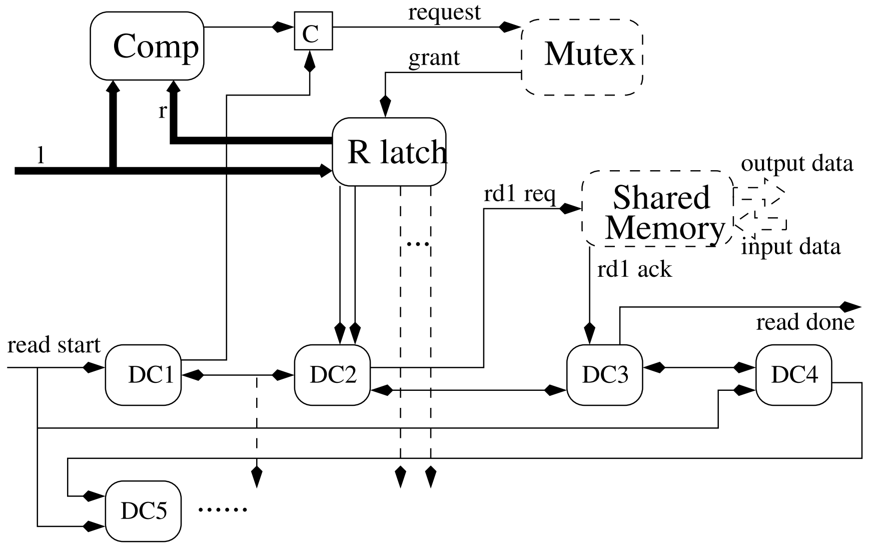


Figure 4.48: Signal Flow for the Reader

4.6.3 Implementation of the Datapath

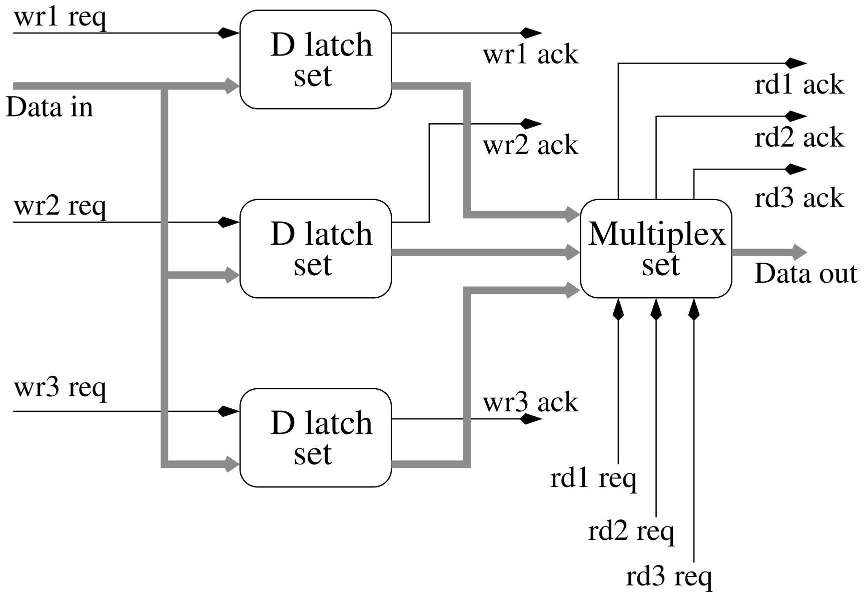


Figure 4.49: Signal Flow for Datapath

The signal flow for the datapath is illustrated in Figure 4.49. This implementation has a data word size of a byte, i.e. 8-bit. A D latch set is a combination of 8 D latches, and a multiplex set is a combination of 8 multiplexers. When it receives a request from the writer, the input data item is stored into the corresponding D latch set. When writing is completed, the acknowledgement is sent back to the writer. The data outputs from the 3 D latch sets connect to the three data inputs of the multiplex set. When a read request comes to the multiplex set, one data input is delivered to the output accordingly, and then an acknowledgement is sent back to the reader.

4.6.4 Resulting Waveforms and Discussions

Digital simulations of the circuit were carried out with the Cadence tool. The test-bench is a piece of code written in Verilog (see Appendix A). It generates the source data items and the requests for the reader and the writer. The time taken by the reader and writer outside the ACM, or the interval between adjacent requests was controlled by two independent random number generators. As a commonly used distribution in reliability engineering [Boh96], the exponential distribution was assumed, and the same mean value was set for both w0 to wr and rd to r0. This gave enough variation for reader waiting and writer overwriting conditions to appear. Figure 4.50 shows the resulting waveform of one digital simulation. The *write_ start* is not included in the waveforms because the change of input data implies the *write_ start* arriving.

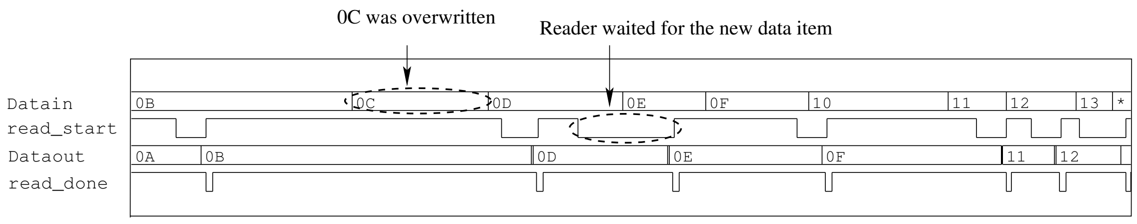


Figure 4.50: Result Waveforms

In the Dataout stream, there are several temporary data values which appear as thick bars, such as the ones before 0D and 0E. The occurrence of these data values were caused by the speed differences among the 8 multiplexers. It would not affect what the reader read because the data was stable before the *read_ done* was generated.

In this sequence, after data item 0D was read by the reader, another read request arrived. The reader did not respond to requests until a new data item was available. During this period, the *read_ start* signal stayed low. On the other hand, when the writer delivered the data items quickly, such as 0C to 0D, overwriting occurred (0C was overwritten). The overwritten data items could not be delivered to the reader, therefore, there is data loss in the *Signal*.

It can be seen from the resulting waveforms that whenever the *read_ start* signal came, the *Signal* outputted the most recent data item it had obtained. In the other words, the *Signal* maintains the property of freshness. The other fact is that, although there were some missing data items, the order of the data items received by the reader was still the same as that written into the *Signal* by the writer. That is, the *Signal* preserves the data sequencing property.

With monitoring the reading and the writing indices, it was also found that the data coherence property was also preserved, because the reading and the writing indices were never the same value at the same time. This is due to the correct designed algorithm.

Although the waveforms in Figure 4.50 are not exactly the same as those in Figure 4.16, they imply the same properties, such as writer overwriting, reader waiting and freshness. The differences include the overall simulation time, the active level, the intervals between requests and the representation of data items. The differences were mainly caused by the different simulation environments and the randomly generated requests.

4.7 Conclusion

There are two ways to model ACMs in MATLAB environment. One is from Petri net specifications, and the other is directly from algorithms.

The former one is based on the method of transforming Petri nets to Stateflow explained in Chapter 3. The models created by this method are closer to hardware implementations, as circuits could be built with the same PN specifications by using direct translation technique. However, these models are complex because they are more focused on internal structure, which is not necessary in investigating their applications. A 3-slot *Signal* was used as example to propose this method and the hardware implementations. The results show that both the MATLAB model and the hardware circuits imply the same properties, such as writer overwriting, reader waiting and freshness.

The latter one is focused on the behaviours of ACMs. Control path of each ACM is built with handshake models after analysing the information exchange pairs within the algorithm. Generating and releasing requests / acknowledges are carried out in entry actions and exit actions of states. Waiting for acknowledges is a condition of a transition. Updating and referencing control variables are modelled in transition actions. Datapath designs only depend on the number of slots in the memory. For example, a 3-slot Signal shares the same memory structure with a 3-slot Pool. Furthermore, the datapaths of all 3-slot ACMs are the same. This also applies to all 2-slot and 4-slot ACMs.

The same method is also used in modelling buffered ACMs from algorithms. A buffered ACM could be modelled in two ways, a global view model and a modular design one. Compared to the latter, the former one could be more easily built in MATLAB. However, the latter was more suitable for hardware design. As the simulation results of an RR-BB for these two types of models showed exactly the same properties, most of the further discussions are based on the global view model.

An RR-BB ACM has the following properties: the writer writes data items to the buffer until the ACM is full, and, at this time, it does not respond to further requests until the state is changed. The reader reads the data items in the buffer in order until the ACM is empty, and it re-reads the latest item until the new one is available. It gives the best performance, best data continuity and average data latency, when the speed difference between the reader and the writer is small.

Additional cells normally increase the data continuity while they increase the data latency. However, after they increased to a certain number, the additional cells will not improve the performance when the reader is faster, and even increase the data latency and worsen the data freshness when the writer is faster.

Chapter 5

Application in Control Systems

5.1 Introduction

In the previous chapters, ACM models have been integrated into the popular application-level tool MATLAB. With these models, ACM applications could be investigated. In this chapter the brushless DC motor will be taken as the example to discuss the effect of including ACMs in such engineering application systems as control systems, especially when analogue parts are present.

5.2 Brushless DC control system

5.2.1 Introduction

A brushless DC control system consists of a mechanical part (the motor), an electrical part (the motor's drive sub-system) and an electronic part (the integrated circuit controller). Cascade control is usually used for this drive, with an inner loop controlling motor current or torque and an outer loop controlling motor speed. These two controllers could be implemented by PID controllers.

The output of the current controller is the drive to be applied on the motor. It is modulated by a PWM which generates a drive voltage. The level "on" and "off" in a PWM cycle is switched by a transistor. Because overheating caused by a high current could damage the transistor, the demand current calculated by the speed controller is normally limited within a certain range.

5.2.2 System Model

In this section, we will investigate each part of the system shown in Figure 2.12.

5.2.2.1 Motor

The transfer functions of the motor can be derived from the torque balance equation and the electrical equation.

The torque balance equation is:

$$J\frac{d\omega(t)}{dt} + \omega(t)D = K_m i(t), \tag{5.2.1}$$

where J is the inertia, $\omega(t)$ is the angular velocity, D is the drag load, K_m is the motor constant and $i(t)$ is instantaneous current.

The electrical equation:

$$L\frac{di(t)}{dt} + Ri(t) = v_a(t) - E\omega(t), \tag{5.2.2}$$

where L is the motor inductance, and R is the motor resistance, $v_a(t)$ is the applied voltage and E is the back e.m.f. constant.

Take the Laplace transformation and re-arrange these two equations, the transfer functions of the motor would be:

$$\Omega(s) = \frac{K_m}{Js + D} I(s), \tag{5.2.3}$$

and

$$I(s) = \frac{V_a(s) - E\Omega(s)}{Ls + R}. \tag{5.2.4}$$

The model of the motor is built according to these two transfer functions as shown in Figure 5.1.

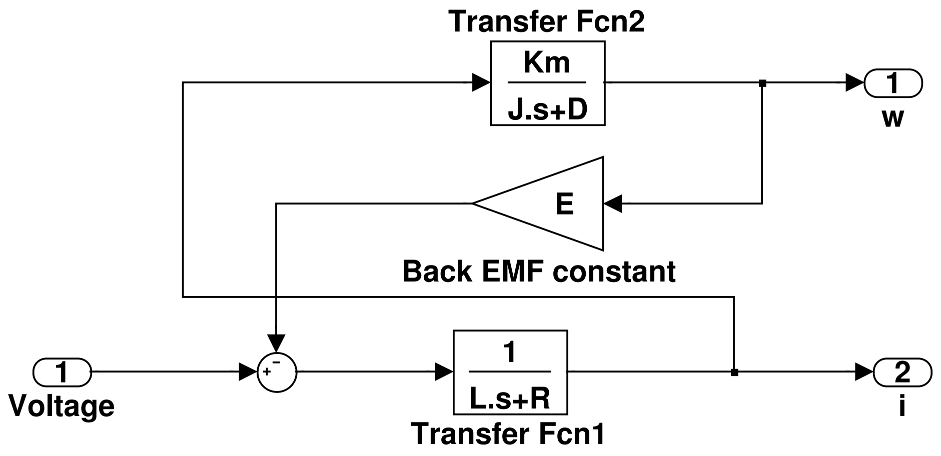


Figure 5.1: Model of a Motor

The angular velocity and the current are related to each other. The **Transfer Fcn2** represents the Equation 5.2.3. It is used to find out the angular velocity according to the current. The product of the angular velocity and the back e.m.f. constant is the back e.m.f. voltage. Subtracting it from the applied voltage gives the numerator of the Equation 5.2.4. Feeding the result to the block **Transfer Fcn1** (the denominator of the Equation 5.2.4), the output of the block would be the current.

5.2.2.2 PWM

Pulse Width Modulation is a technique employed to regulate the output power by changing the pulse width. In a digital system, an integral number of steps related to the clock frequency is used as the pulse width. Thus, there are two frequencies in the system: the clock frequency, and the PWM frequency. There is only one pulse in each PWM cycle, and one pulse is made up of several clock cycles. The pulse may be located in three positions: in the start of the PWM cycle, in the centre of the cycle or in the end of the cycle.

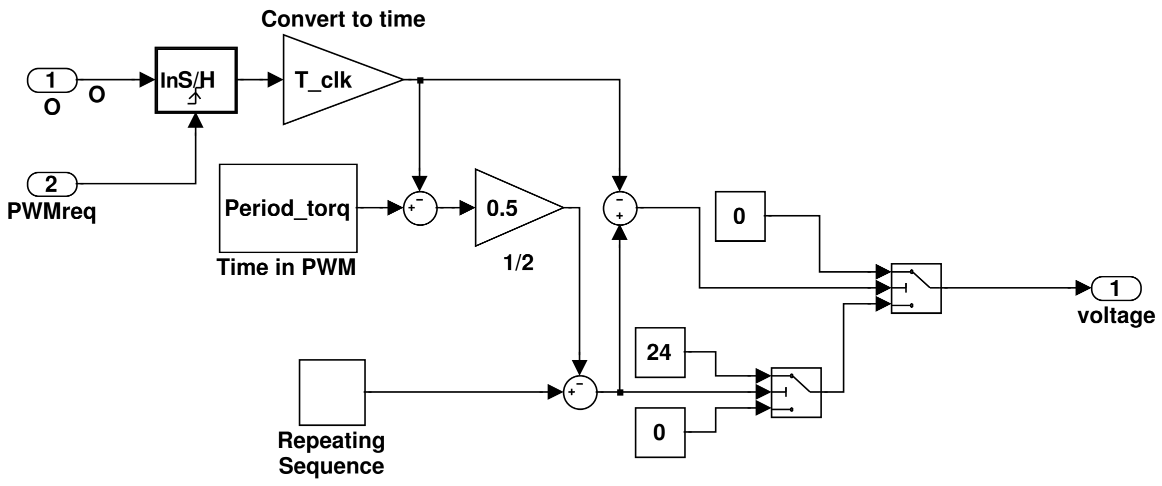


Figure 5.2: PWM Block

Figure 5.2 shows the PWM block in the DC motor controller. In this block, the pulse was set to the centre. To do this, the total **OFF** time in each PWM cycle was calculated by the time in a PWM cycle (**Period_torq**) subtracting the time for each pulse. The time for each pulse was achieved by the number of steps **ON** multiplied by the clock period (**T_clk**). The time reference here was generated by a ramp tooth sequence repeating on the PWM frequency. It reset its output every PWM cycle. When the time reference reached half of the **OFF** time, the block delivered 24 volts to the output. When the time reference reached half of the **OFF** time plus the time for the pulse, the block switched the output from 24 to 0 volts.

5.2.2.3 Complete Model

The complete model for a brushless DC motor is shown in Figure 5.3.

The **Reference Speed** is in the unit of *RPM*. The speed output of the **Motor** is in the unit of *rad/s*. Both of them should be converted to *Hz* before they are fed

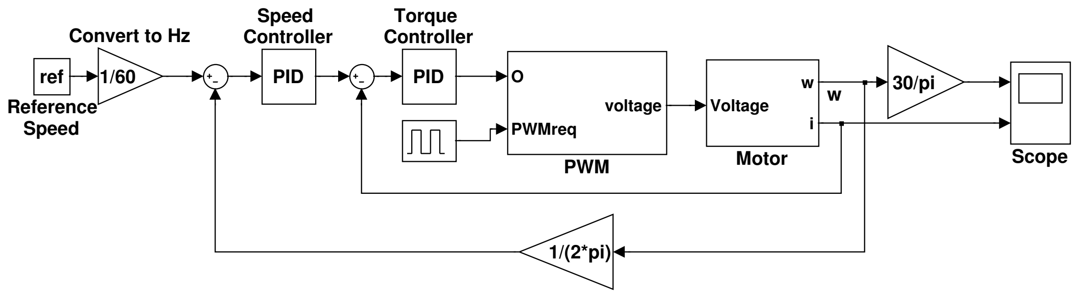


Figure 5.3: Complete Modal for a Brushless DC Motor

into the **sum** block, because the speed controller was designed to require a speed error in *Hz*. The speed controller and the torque controller were modelled by two PID blocks. The error between the instantaneous current, from the **Motor**, and the required current, calculated by the **Speed Controller**, was input to the **Torque Controller** to find out the amount of drive. The **PWM**, triggered by a pulse generator, worked out the voltage to drive the motor according to the output value from the **Torque Controller**. The two outputs of the **Motor**: speed (converted to *RPM*) and current were monitored by a **Scope**.

5.2.3 System Analysis

The system parameters were obtained from [Bla96]. They are: $L = 0.04mH$; $R = 0.7ohms$; $J = 10^{-5}kgm^2/s^2$; $E = 1.53 \times 10^{-2}Nm/A$; $K_m = 0.0153Vs/rad$; $D = 10^{-6}Nms$; $ref = 6000RPM$. The PID coefficients for the speed controller are $K_p = 0.024$; $K_i = 0.023$; $K_d = 0$. Those for the torque controller are $C_p = 0.003$; $C_i = 2881$; $C_d = 0$. The clock frequencies for the speed controller and the torque controller are 1 kHz and 30 kHz. Applying them to the model and simulating the model, it gave the resulting waveforms as shown in Figure 5.4.

The top waveform is the angular velocity and the bottom one is the current. When the system was started up, the actual speed was 0. The speed was increasing rapidly within the first 0.5 minutes. After the overshoot, it stabilised at 6000 rpm which was the desired speed. By zooming in the waveform for the angular velocity,

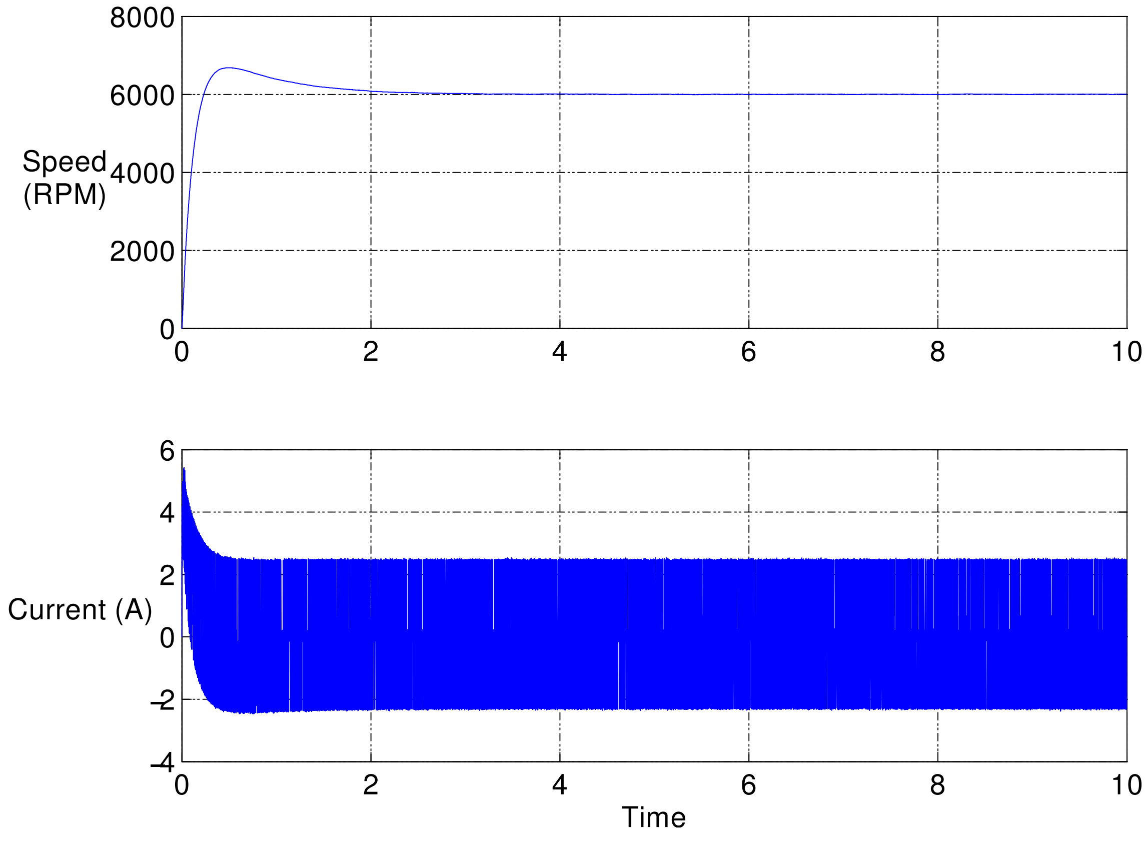


Figure 5.4: Waveforms for the Brushless DC Motor

it can be seen that the velocity is oscillating between 5999.96 to 6000.01 at a high frequency at about 30kHz (the PWM frequency). There was a high pulse at the very beginning in the current waveform, which was necessary to drive the motor from rest to a high speed. With the speed being stable, the current oscillated around 0 amps. The oscillations for the angular velocity and the current are apparently caused by the PWM.

The system worked as it expected. As the initial motor speed was 0, the difference between it and the reference speed was significantly large. The large difference demanded a large current to drive the motor. The current controller made this demand reality. Driven by the large current, the speed increased rapidly. With the speed increasing, the speed error decreased, which lowered the demand current. As the result, the change rate of the speed was lower and lower, and finally, stabilised at the reference speed. When the speed stabilised, the error of the speed became 0. The speed controller was dominated by the integral term, which made the demand current oscillate around 0.

To analyse the stability of the system, the model was linearised by the Control Design tool in Simulink. The open loop transfer function for the angular velocity is:

$$G(s) = \frac{248.5s + 2.386 \times 10^8 s + 2.287 \times 10^8}{s^4 + 1.751 \times 10^4 s^3 + 7.084 \times 10^6 s^2 + 3.174 \times 10^7 s}. \tag{5.2.5}$$

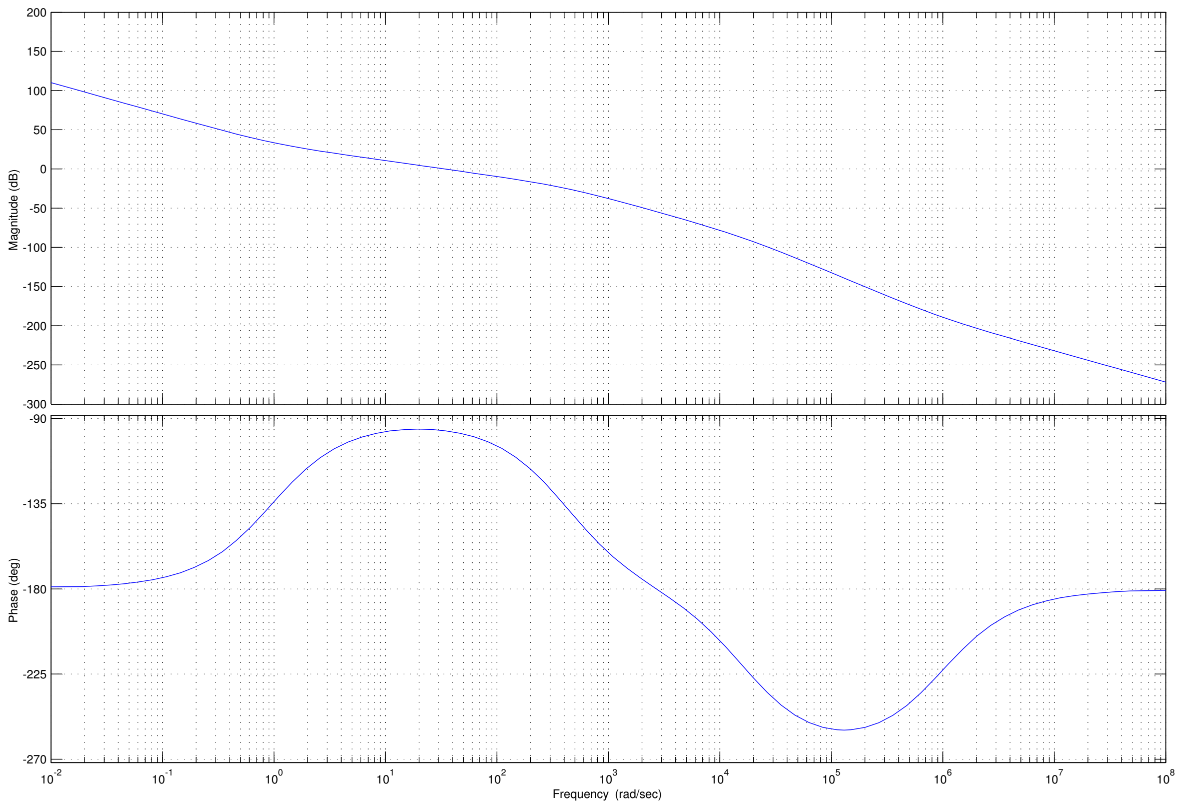


Figure 5.5: Bode Plot for the Open Loop of the Brushless DC Motor System

The Bode plot [FPEN06] is shown in Figure 5.5. From the Bode plot we can see that the Phase is about -97^o when the magnitude is 1, or 0 dB, and the magnitude is about -54 dB, i.e., the absolute value is between 0 and 1, at -180^o. According to the stability criterion stated in 2.4.3.1, the system is stable.

5.3 Asynchronous Solutions

5.3.1 Introduction

The synchronous model for the brushless DC motor system has been built and was analysed in the previous section. In the synchronous solution, the global clock is necessary. The shortcomings of the global clock solution include the problems of the propagation delay, the clock skew and EMI from the high frequencies, etc. All these problems may affect the overall performance. Therefore, it is important to discuss the asynchronous solutions for this system.

There are three different time domains in this system. The first one is the torque controller which executes at a high speed. The second one is the speed controller which runs at a comparatively lower frequency. The third one is the shaft encoder whose sampling rate relies on the speed of the shaft, i.e., non-uniform. These three parts would be running in their local clock frequencies, therefore, the whole system could be implemented as a GALS system. The communications within them need to be carefully investigated.

5.3.2 Buffer solution

One of the asynchronous solutions is using a buffer between two different time domains to cope with the synchronisation. The data provider stores the data into the buffer; the data receiver reads data from the buffer. The longer the buffer is, the more the asynchrony can be tolerated.

The drawbacks of the buffer solution are obvious. It only suitable for connecting two processes whose speeds are close to each other. If one process is faster than the other, the buffer will always be full or empty. The faster process will have to wait until the other process reads a data item from the buffer or writes a data item to the buffer. Therefore, its speed will be drawn to the same speed as the slow one.

Furthermore, the speed of whole system will be drawn to the same speed as the slowest process.

In motor control systems especially, if the inner and outer loops are not temporally decoupled, potential digital hazards such as deadlocks can propagate through from one loop to another. The function of the inner control loop is normally safety-critical, because even temporary failure there could have catastrophic effects such as causing the power electronic elements or fuses to fail. If such a motor is used in a safety-critical application (for instance in an aeroplane fuel pump), such failures which cannot be recovered on-line must always be avoided. As a result, the capability of the inner loop to continue functioning even when the outer loop has stopped working is of vital importance. This means that even though both the speed controller and the torque controller may be integrated into the same piece of silicon, they must in reality be temporally independent of each other.

5.3.3 DAC-ADC solution

Another asynchronous approach could be that the link between the speed controller and torque controller is in effect implemented as an analogue connection, with the digital output from the speed controller first converted into analogue then re-sampled to provide the input for the torque controller. This kind of temporal decoupling is essential in these kinds of distributed systems.

Assuming the same technology is being used to implement them in hardware, the part of the hardware where the speed controller is implemented could have large amounts of excess computational capacity due to the difference in speed requirements for the speed controller and the torque controller. This makes it attractive to attempt to make use of this capacity for other tasks, i.e. to effectively implement the speed controller part as one of the threads in a multi-tasking processing element. This makes it possible for its progress to be affected by other factors outside the immediate control system boundary. Well-implemented operating systems such as real-time kernels may take care of the safety-critical implications of such complications by ensuring that critical threads do not wait for information from other threads.

At the basic hardware level of the data connection between the torque controller and the speed controller of an embedded hard-wired controller chip, this kind of non-blocking communication can be implemented by using an analogue link. However, this implies an analogue/digital hybrid chip, and this might not be practical.

5.3.4 Solution with ACMs

With ACMs, the same kind of temporal decoupling stated in the previous section can be realised without resorting to inserting an analogue wire between two digital devices. The *Pool* type ACMs, especially, mimic this function of an analogue wire perfectly. When a *Pool* is "full", the writer overwrites one of the items in it instead of waiting for a space to appear, and when it is "empty" the reader rereads the item it read during the previous cycle instead of waiting for a new item to appear. This is functionally the same as connecting the writer with the reader through a Digital to Analogue (D/A) and Analogue to Digital (A/D) converter pair, assuming perfect level-matching in the converters.

5.4 Asynchronous Model

5.4.1 Introduction

To investigate the asynchronous solutions for the system, an asynchronous model should be built. As known, handshake protocols are used in asynchronous communications. The components that need to communicate with the other time domains will have to be redesigned with handshake interfaces. As discussed in Chapter 4, handshakes can be modelled in Stateflow, these components are to be remodelled in Stateflow. The components include two controllers and an ADC for the speed output from the motor. The ADC was not shown in the model stated in Figure 5.3, because MATLAB itself had already converted the speed into a digital form and delivered it synchronously. However, in the asynchronous model, an asynchronous ADC had to be discussed because it only executes when it is necessary and would trigger the followed asynchronous communication component.

5.4.2 Asynchronous PI Controllers

As stated in Chapter 2, a PID controller combines proportional, integral, and derivative control actions together, as shown in Figure 5.6. The generic transfer function is:

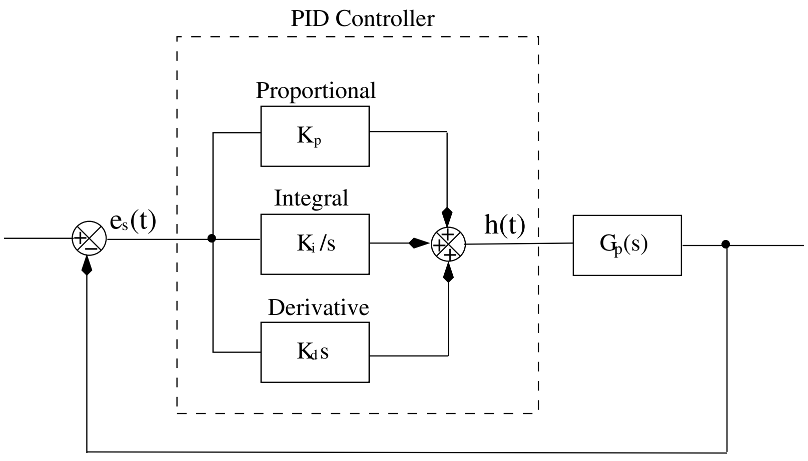


Figure 5.6: PID Controller

$$H(s) = K_p + \frac{K_i}{s} + K_d s \tag{5.4.1}$$

or

$$h(t) = K_p e_s(t) + K_i \int_0^t e_s(t)dt + K_d \frac{de_s(t)}{dt} \tag{5.4.2}$$

K_p, K_i and K_d are the proportional, integral and derivative coefficients, *h(t)* is the output signal, e_s*(t)* is the error between the actual and the reference input signal. Because neither of the two controllers in this motor system contain the derivative term, we only discuss the PI controller case here. A controller may receive data from one process, and deliver its output to the other process. If both of these two processes are executed in different speeds to the controller, the handshake interfaces have to be added to both the input and the output of the controller.

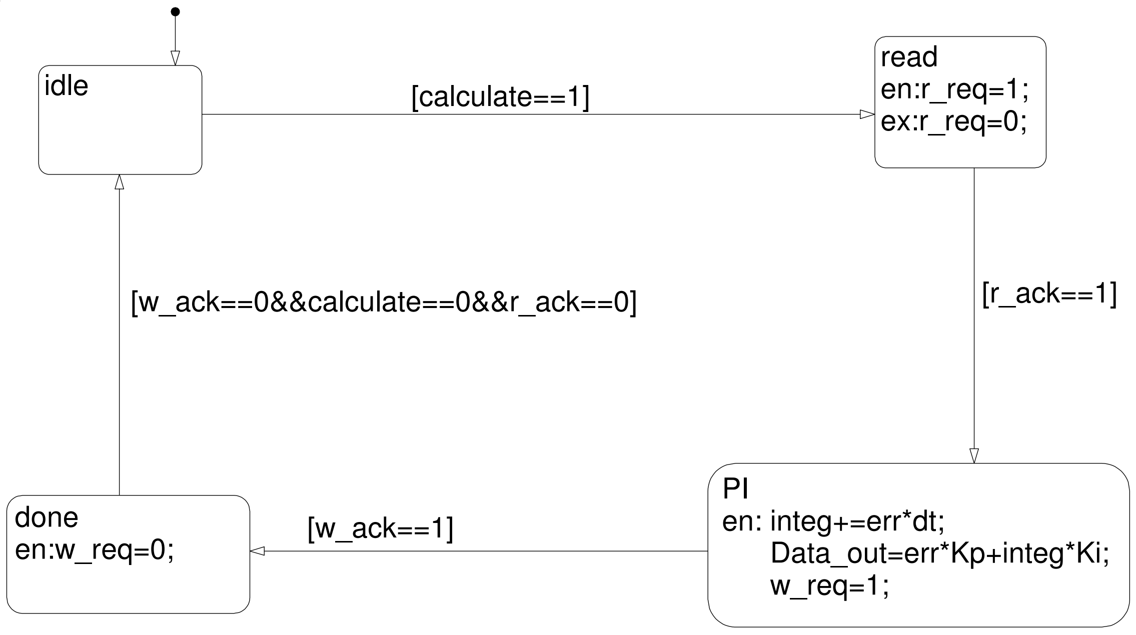


Figure 5.7: Asynchronous PI Controller Model

The PI calculation could be carried out within one State, and the input and the output handshake interfaces were implemented in two States. The Stateflow model of an asynchronous PI controller is shown in Figure 5.7.

Here, *calculate* signal is the trigger of the execution, *r_ req* is the request sending to the previous process for seeking data, *w_ req* is the request sending to the following process for delivering the result, *r_ ack* and *w_ ack* are the corresponding acknowledge signals, *err* is the error signal, *dt* is the time difference between two calculations, *integ* is the integral, and *Data_ out* is the output data.

This is how it works: the chart is initially in the *idle* state. When a *calculate* signal triggers it, a request (*r_ req*) is sent to the previous process for seeking data. After receiving the acknowledge (*r_ ack*), it releases the request (*r_ req*) and then performs a PI calculation in state *PI*. Firstly, accumulate the product of *err* and *dt* to find out the integral (*integ*). Secondly, work out the result (*Data_ out*) by adding the P and I terms. When *Data_ out* is calculated, a request (*w_ req*) is sent to the following process for delivering the result. After receiving the acknowledge (*w_ ack*), the model releases the request (*w_ req*). When both the two acknowledges (*r_ ack* and *w_ ack*) and the trigger signal (*calculate*) are withdrawn, a whole cycle is completed. The chart goes back to the *idle* state.

Here, the trigger signal (*calculate*), the acknowledges (*r_ ack* and *w_ ack*), the time difference (*dt*) and the error (*err*) are the input signals. The result (*Data_ out*) and the requests (*r_ req* and *w_ req*) are the output signals.

5.4.2.1 Speed Controller

The speed controller in this system receives data items from the ADC and sends its output to the current controller. The execution speed of the ADC depends on the angular velocity of the shaft in the motor, and the current controller executes at a high frequency compared to the speed controller. Both the ADC and the current controller are in a different time domain to the speed controller, therefore, the handshake interface for both the input and the output of the speed controller should be retained.

In addition, to protect the transistor in the PWM, the desired current should be limited within a certain range. If the calculated current exceeds the limit, the output has to be set to the nearest bound. This could be modelled within an additional state and transitions between *PI* and *done*. As shown in Figure 5.8. The calculation result was assigned to *tmp* instead of *Data_ out* in this case. If tmp exceeded the range between the upper limit (*limit_ up*) and the lower limit (*limit_ low*), *Data_ out*

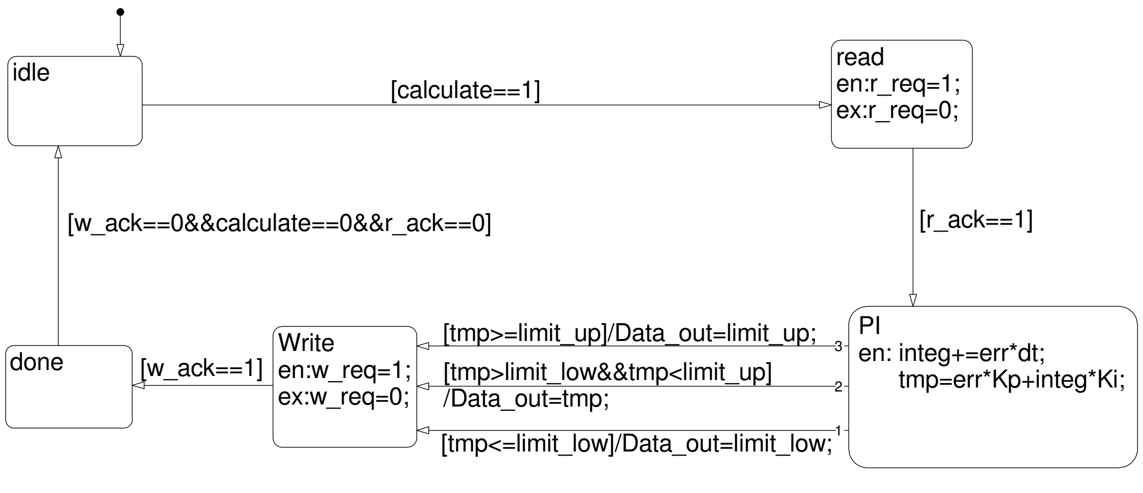


Figure 5.8: Stateflow Model of a Speed Controller

would be set to the nearest limit. Otherwise, the output of the controller would be the calculated result. After the output was determined, the controller would send the write request to the component that followed.

5.4.2.2 Torque Controller

The torque controller receives data from the speed controller and sends its output to the PWM. As in the different time domains, it needs a handshake interface for receiving data from the speed controller. As known, the output of the torque controller is the amount of drive in form of a pulse width for each PWM cycle; therefore, the controller should work in the same frequency to the PWM. As a result, there is no need to use a handshake interface for handling the communication for its output. Furthermore, because the pulse width can only be from 0 to full, the output should also be limited.

Compared to Figure 5.8, the differences in Figure 5.9 include removing the handshake interface for the output (state *Write* and the transition following) and one of the three transition conditions *"w_ ack==0"* after the *done* state.

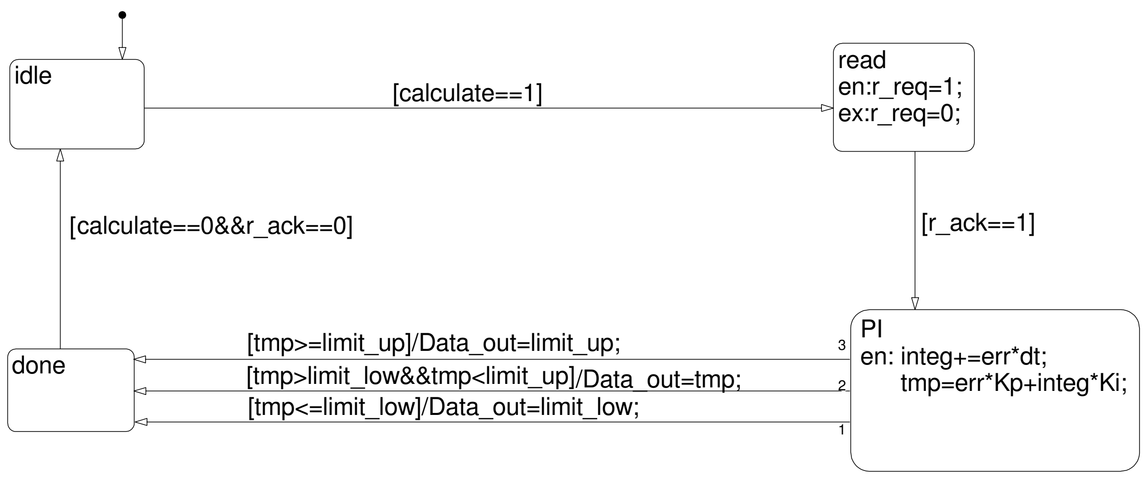


Figure 5.9: Stateflow Model of a Torque Controller

5.4.3 Asynchronous ADC

The asynchronous ADC is used to convert the analogue position information obtained by the sensors into the speed in a digital form asynchronously. In the motor system, it is actually a shaft encoder.

As stated before, the speed of the motor is calculated by dividing the angle between two sensors by the time interval between two pulses generated by the sensors. In the asynchronous system, the speed data should be generated whenever the sensors trigger. At this time, there are two parameters which need to be measured: the angle and the time interval. Angle measurement needs to be mentioned here. In the real motor, the angle is known, however, it is an internal value in the motor block which cannot be obtained directly in the Simulink model. To measure the angle, the speed output from the motor needs to be integrated. The time interval could be obtained by the number of a high local frequency clock, or the capacity in an LC circuit, etc.

Modelling the asynchronous ADC includes modelling the trigger of the sensors and modelling the speed calculation.

The trigger could be modelled as the condition that the input angle reaches to certain levels. The step of these levels is the angle between two adjacent sensors. The following is how it can be achieved. One level is taken, normally 0 rad, as the initial reference. When the angle goes beyond the range [*reference*, *reference + step*) this represents the sensors triggering. At this time, the ADC updates the reference to the level which it just crossed, and then performs a speed calculation.

The speed calculation is calculated by dividing the angle difference by the time difference between two triggers. The trigger event should be that the angle reaches the next or previous level, in other words, the trigger condition is [*angle == reference + step*] or [*angle == reference*]. The angle difference should be exactly the step between two levels that is 60 degrees or pi/3 rad in this example. However, in MATLAB modelling, the angle itself is normally a floating point number, and the reference and step are also floating point numbers. In MATLAB, there is a known problem that the comparison of two floating point numbers could hardly be equal, that is, the condition of [*angle == reference + step*] or [*angle == reference*] could hardly be true. To make it work, the trigger conditions are changed to the angle crossing instead of reaching the levels, i.e. [*angle >= reference + step*] or [*angle < reference*]. Under this condition, the angle difference would not be the constant step any more, but the difference between the input angle for the current trigger and that for the previous one. Therefore, after the speed calculation, the current input angle should be stored so that it could be used in calculating the angle difference in next cycle. The time difference could be calculated in the same way.

Figure 5.10 shows the model of the asynchronous ADC in Stateflow. The *start* state was used to start the whole system. The two transitions after the *idle* state model are the trigger of the sensors. The speed calculation was performed in *output* state. Because the speed controller requires a speed in the unit of Hz, the output had to be converted from an angular velocity (rad/s) to a frequency (Hz), which was achieved by dividing by $2 \times \pi$. After the calculation, the current time and angle were stored, and then a write request was sent. Normally, it should wait for an

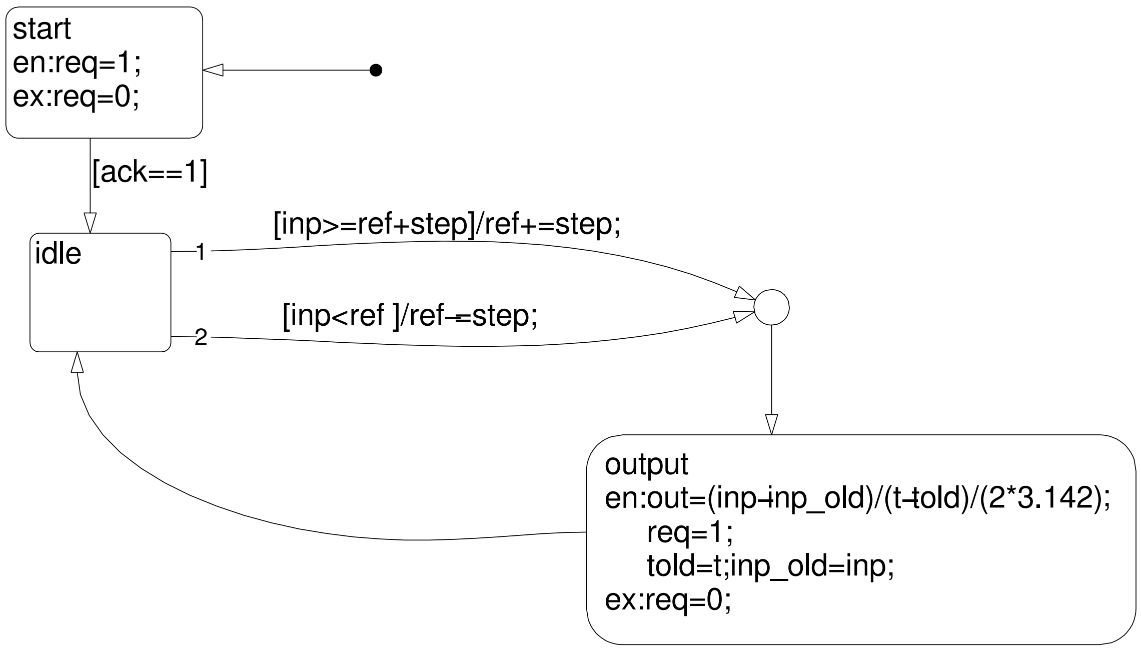


Figure 5.10: Stateflow Model of an Asynchronous ADC

acknowledgement to complete the handshake. However, the ADC must be executed in real-time because the extra wait for an acknowledgement in the cycle may cause missing samples and incorrect speed outputs. Therefore, there was no condition in the transition between the state *output* and *idle*.

5.4.4 Whole System

Having all the essential components modelled, the whole system could be built as shown in Figure 5.11.

Compared to the model in Figure 5.3, the two PID blocks were changed to two blocks containing Stateflow models; an integrator and an asynchronous ADC were added in the feedback link in the outer loop; two intermediate blocks were added, one was between the ADC and the speed controller, and the other was between the speed controller and the torque controller. These two intermediate blocks were the asynchronous communication connections, they could be buffers or ACMs.

With inclusion of buffers or ACMs into the system, delayed (because of wait), re-

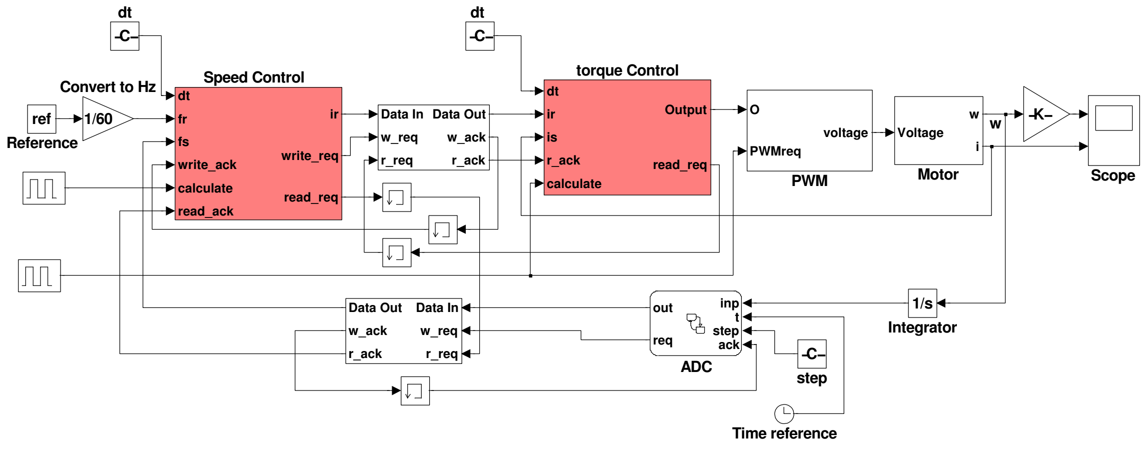


Figure 5.11: Model of the Whole System

peated (because of re-reading) and missing (because of over-writing) samples could occur during transmission. It is worth discussing the impact of them on the behaviour of this system.

A short delay would temporarily break the loop, and worsen the system performance [ZRG04]. The longer the delay, the worse the systems performance. A sufficiently long delay would effectively break the loop, and the system would fail because of it.

Missing samples and repeated samples which arrive at the correct moment are equivalent to adding noise to the system. However, the noise is small enough to be ignored. For example, consider a signal is sampled by an asynchronous ADC whose step is x, and a process, who uses the signal as an input, executes at a rate of $1/Ts$. If the signal is a constant, the asynchronous ADC will only sample once at the initial stage, and the process would read the same data item every cycle. The equivalent added noise is the difference between the actual value and the level, which is smaller than x. If the signal changes comparatively slowly, the previous data item is reread by the process only when the signal has not reached the next level in asynchronous ADC. Thus, the equivalent added noise is also the difference between the actual value and the level, which is still smaller than x. As x is the smallest unit in the asynchronous ADC, any value smaller than that has to be ignored. If the signal changes comparatively fast, some data items could be missed, and the process reads the latest data item provided by the asynchronous ADC. All the samples missed would be within the time period *Ts*. As *Ts* is the smallest time unit in the process, the missed samples within this unit would not affect the performance of a real-time system.

5.5 Results with Different ACMs

5.5.1 Introduction

In this section, the behaviour of the brushless DC motor system with inclusion of different types of ACMs will be shown and compared. As *Channel* can be known as a buffer with a handshake interface, we will take *Channel* as buffer in the simulations.

As the reference speed is 6000 RPM, the speed output would be about 100 Hz. The 6 sensors located in the shaft triggers the asynchronous ADC running at around 600 Hz at the steady state. Compared to the speed controller, which runs at 1 kHz, the asynchronous ADC is slower. That is, the speed controller is a busy reader, while the asynchronous ADC is a lazy writer. A *Pool* or a *Message* should be used to manage the communication between the two processes. However, *Signal* and *Channel* will still be included into the system so that we can see the different system performance according to the different types of ACMs.

Similarly, the torque controller, running at 30 kHz, is a busy reader, while the speed controller is a lazy writer. A *Pool* or a *Message* should be used to manage the communication between the two controllers. Using the other two types of ACM will introduce blocking to the torque controller, which may cause the current to go out of control, consequently overheating the PWM transistor and damage the system.

The simulations in this section will be divided in two catalogues, those in a fault free case and those in a fault case. In the fault case, we will introduce blocking into the outer loop. In the asynchronous systems, the blocking in the outer loop should not propagate to the inner loop, and this would prevent the brushless DC motor system from damage.

Nyquist or Bode plots are often used in designing both continuous-time and discrete-time feedback-systems and predicting their performance. However, these tools cannot be used to illustrate the impact of including ACMs. The inclusion of ACMs will introduce varying delay. Although a constant delay can be easily modelled in MATLAB by a function called "pade", the varying delay cannot be easily modelled as a transfer function. Therefore, it is hard to apply these techniques to illustrate the impact of including ACMs.

5.5.2 Simulation Results for the Fault-Free Case

There are two ACMs in the system. To investigate different behaviours according to different types, one of the two ACMs can be fixed and the other changed.

A *Channel* was fixed into the feedback link, which would block the speed controller temporarily, and varying the ACM between the two controllers. In these cases, the execution rate of speed controller would depend on that of the asynchronous ADC which could not reach 1 kHz.

The simulation results for the *Pool-Channel* case and those for the *Message-Channel* case were similar to each other. The resulting waveforms are shown in Figure 5.12.

Compared to Figure 5.4, the overshoot of the speed in Figure 5.12 was smaller. However, in general, these two figures illustrated similar information: the speed increased rapidly at start up stage, reached a peak at about 0.5 second, then declined and stabilised at the reference speed eventually. The current was largest at the beginning, then declined and oscillated around 0 amps.

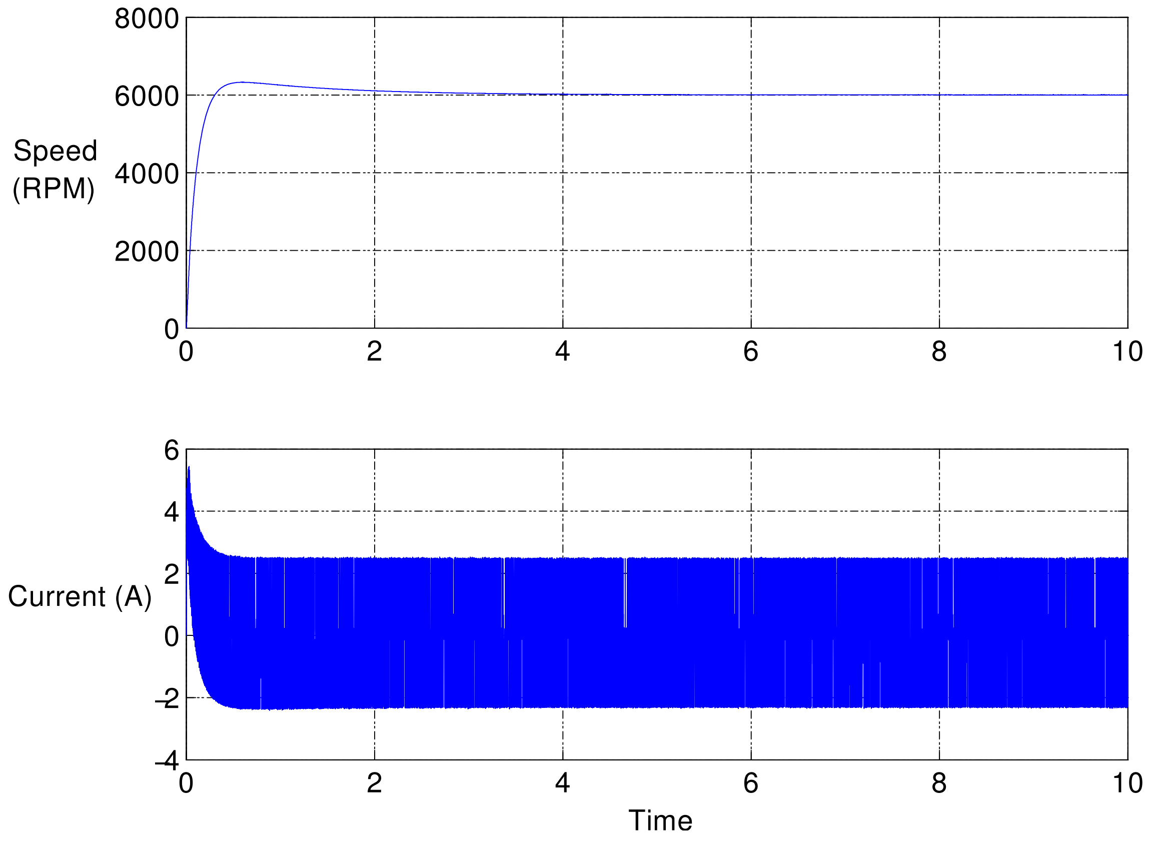


Figure 5.12: Simulation Results for the *Pool-Channel* Case

The simulation results for the *Signal-Channel* case and those for the *Channel-Channel* case were similar to each other. The resulting waveforms are shown in Figure 5.13.

The performance shown in Figure 5.13 was significantly worse than those in Figure 5.12 and Figure 5.4. The current increased slowly at the beginning. The response time of the speed was more than 3 seconds longer. The overshoot of the speed was also larger than it was in Figure 5.12.

All this behaviour could be explained as following: because of the use of the *Signal* or the *Channel* between two controllers, the torque controller had to wait for 29 cycles to obtain an updated data item. During these 29 cycles, the inner loop was blocked, so the PWM could only supply the voltage according to the last value it received to drive the motor, which would normally not be sufficient.

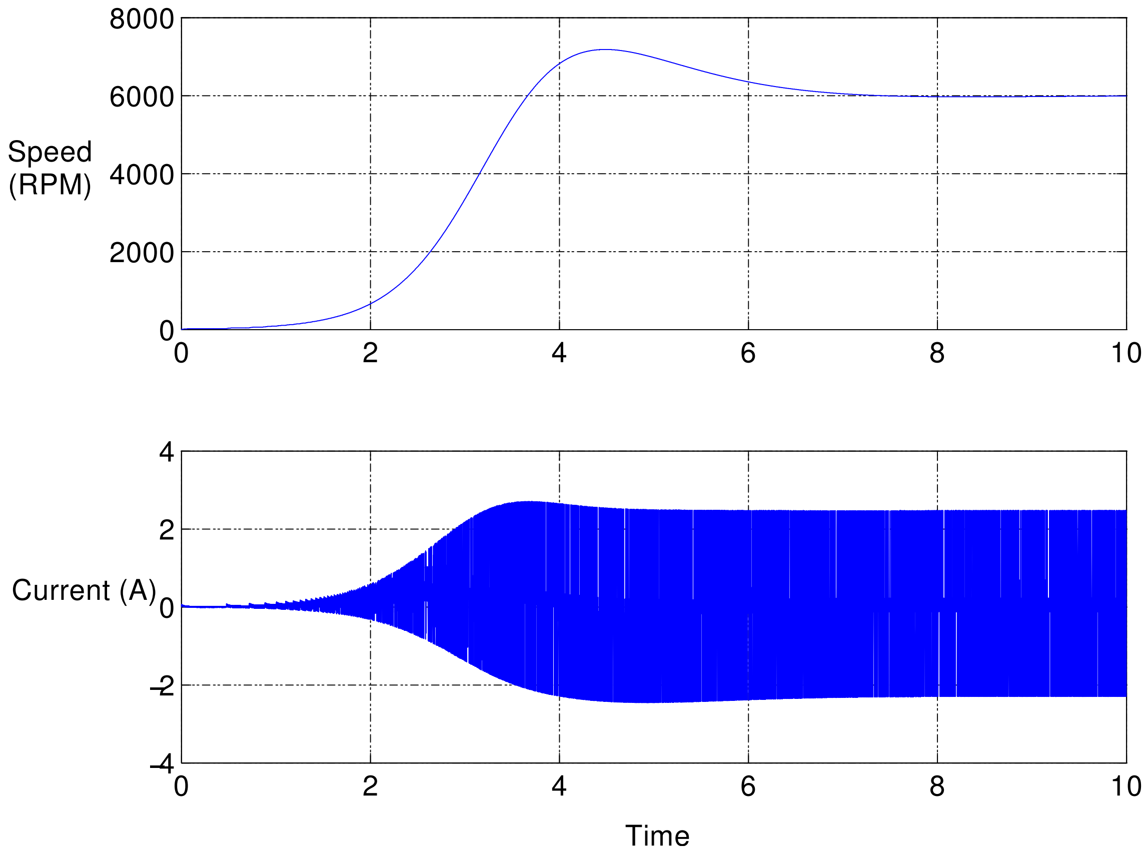


Figure 5.13: Simulation Results for the *Signal-Channel* Case

As the torque controller is exactly 30 times faster than the speed controller, if the torque controller was managed to read once every 30 cycles, implemented by including a counter into the controller, the performance was improved as shown in Figure 5.14.

Compared to Figure 5.12, the speed curves were very similar. The difference was focused on the current. The current in Figure 5.12 started from a high value, about 5 amps, and then oscillated at 0 amps with amplitude of 2 amps. The current in Figure 5.14 started from 0, increased to around 4 amps, and then oscillated at 0 amps with amplitude of 2 amps.

The slower start of the current was because the *Channel* in the feed back link would block the speed controller at the start-up stage. This would make the speed controller execute at a rate much lower than 1 kHz. Although the torque controller

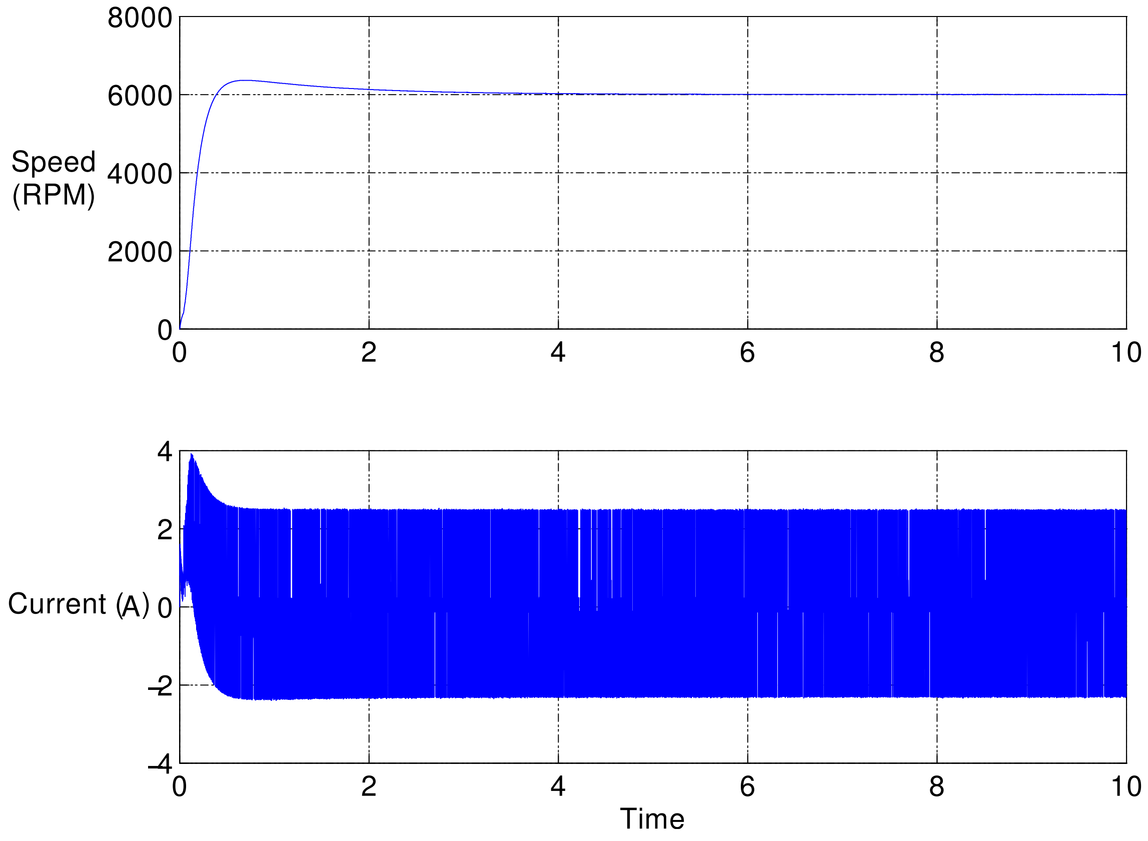


Figure 5.14: Simulation Results for the *Signal-Channel* Case - Refined

updated its input at only 1 kHz, the *Channel* (*Signal*) between the two controllers would still block the torque controller temporarily. Because of the existence of the temporary block in the torque controller, the current response would be slower.

Figures 5.15, 5.16, and 5.17 show the bar graphs of the rising times, peak times and peak values for analogue case and the cases when the four types of ACMs were included between the two controllers while a *Channel* or a *Signal* in the feedback link.

The reasons for the differences between these two categories were that although the *Channel* or the *Signal* in the feedback link reduced the speed of execution in the speed controller, the *Pool* or the *Message* would keep the torque controller running normally, which would guarantee the current under control. The *Channel* or the *Signal* between the two controllers would still block the torque controller, especially

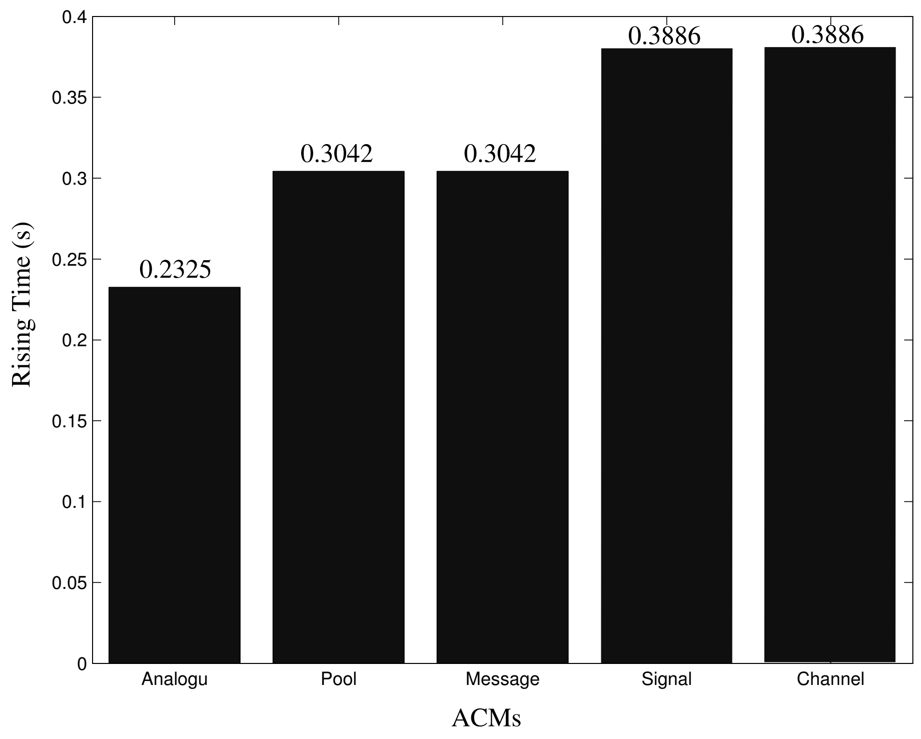


Figure 5.15: Bar Graph for Rising Times for the System for a Step Input

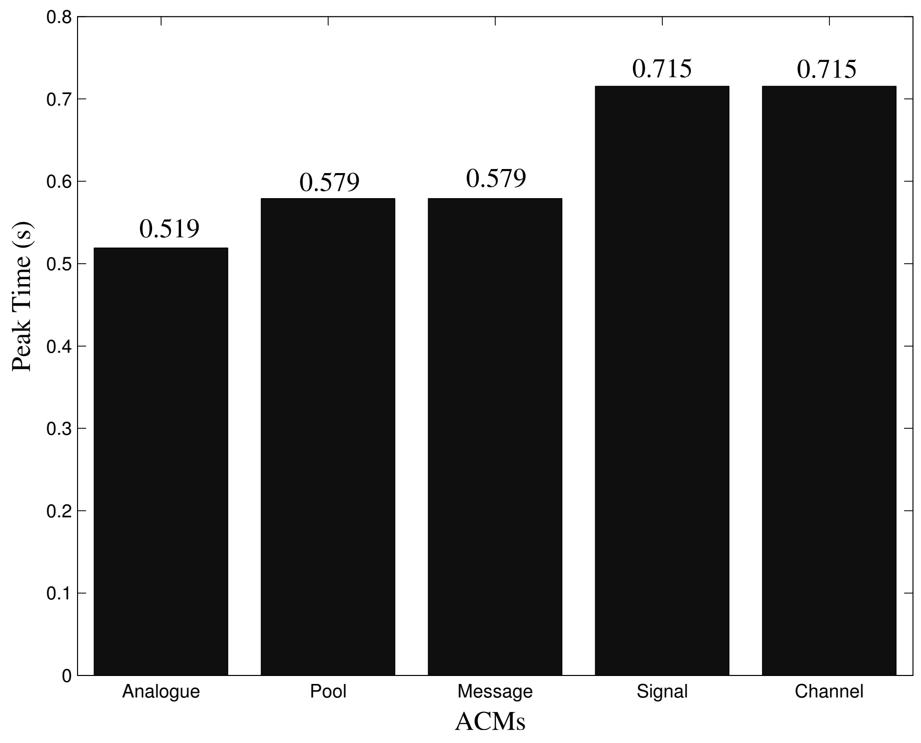


Figure 5.16: Bar Graph for Peak Times for the System for a Step Input

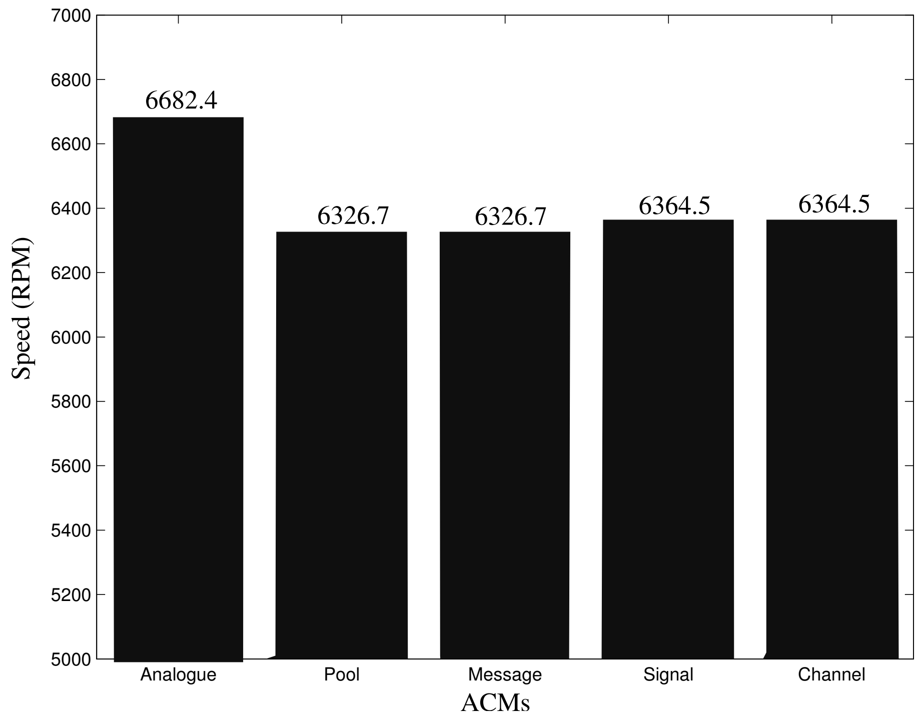


Figure 5.17: Bar Graph for Peak Values for the System for a Step Input

at the start-up stage. Because at this stage, the actual speed was very low, the sensors would trigger the asynchronous ADC at a low rate. The *Channel* (or the *Signal*) in the feedback link would reduce the execution rate of the speed controller to the same as that of the asynchronous ADC. The *Channel* (or the *Signal*) between the two controllers would also reduce the execution rate of the torque controller, but in different manner. The torque controller would run in the following fashion, after every 30 cycles, it was blocked for waiting for a new data item. The slower response of the current would slow down the response time of the speed. As a result, the peak value would be larger, because the motor could not reduce the current in time. With the angular velocity increasing, the blocking duration was decreased.

The analogue case had a shorter response tine and a larger overshoot value compared to the case of including ACMs. This is because the ACMs, especially the one in the feedback path, introduced a delay into the system and jitter to the speed controller. On one hand, the delay and jitter made the system response time longer.

On the other hand, at the start up stage, they resulted in larger speed errors than the actual ones, and consequently, larger drives to correct the errors, which made the peak values smaller.

The second group of the experiments were putting a *Pool* into the feedback link and varying the ACM between the two controllers. In these cases, the speed controller would never be blocked and its execution rate would be fixed at 1 kHz.

Figure 5.18 was the simulation results for the *Channel-Pool* case. It was the same to the *Signal-Pool* case. In these two cases, the refined torque controller was used. Figure 5.19 was the simulation results for the *Pool-Pool* case which was the same as that for the *Message-Pool* case.

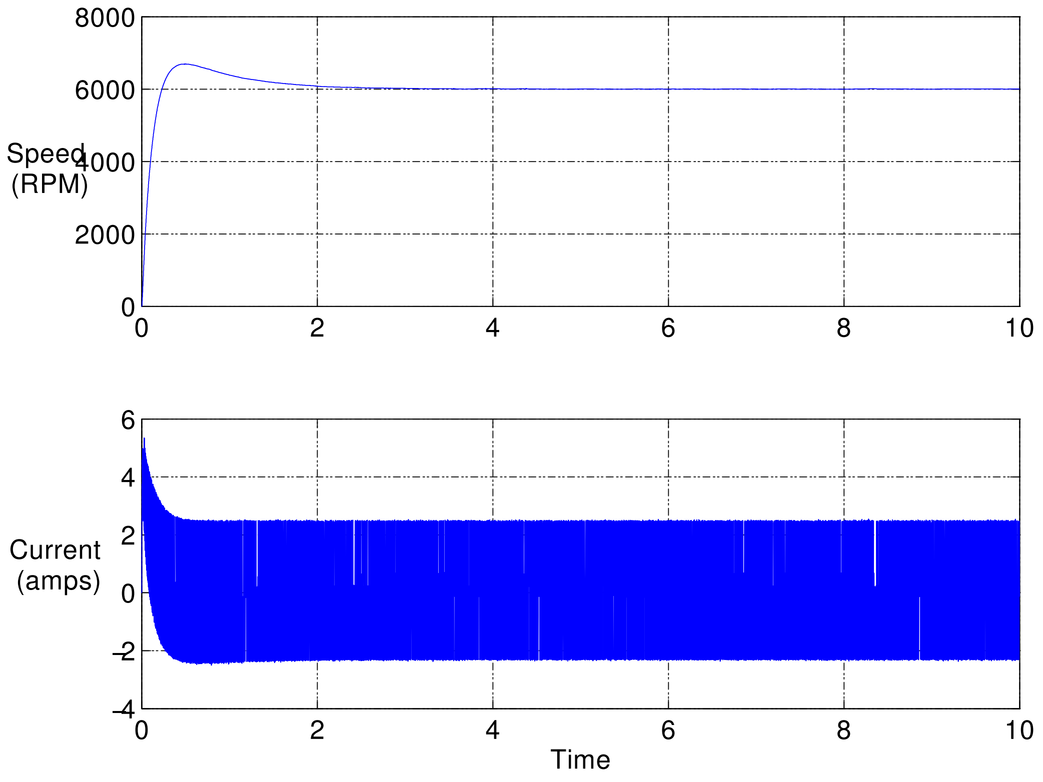


Figure 5.18: Simulation Results for the *Channel-Pool* Case

There were no significant differences between Figure 5.18 and Figure 5.19. In the *Pool-Pool* case, there were no blocks in the system at all. The performance was

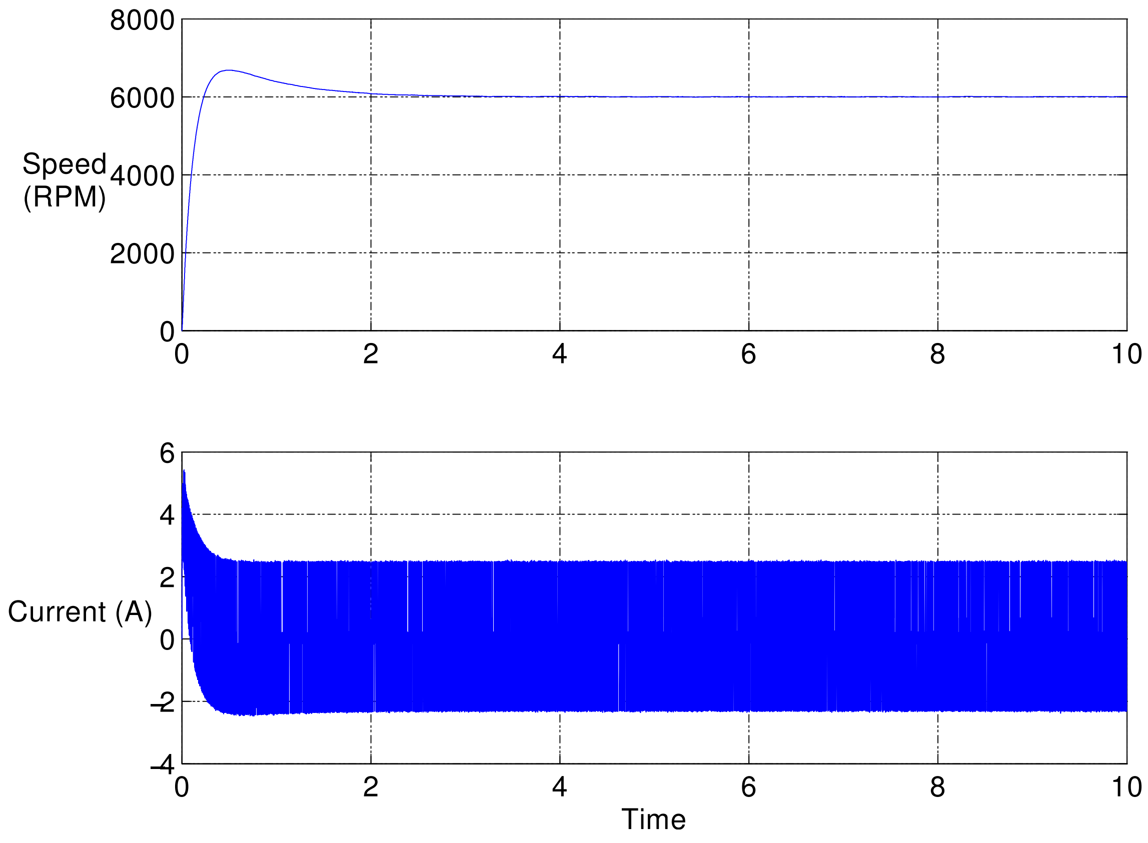


Figure 5.19: Simulation Results for the *Pool-Pool* Case

close to that shown in Figure 5.4. In the *Channel-Pool* case, there was no block in the speed controller, which made it run at 1 kHz. The refined torque controller only updated its input at 1 kHz, therefore, the *Channel* could handle it without introducing any blocks in the torque controller. Although the torque controller in the *Pool-Pool* case enquired from its input every cycle (30 kHz), the data was updated by the speed controller at a rate of 1 kHz, that is, the torque controller used the same data item for the calculations 30 times before it could be changed. This is equivalent to enquiring once then performing 30 cycle calculations. This could explain the similarities between Figure 5.18 and Figure 5.19.

5.5.3 A Fault within the Outer Loop

From the previous section we can see that a *Channel* in the feedback link could hardly make the speed controller run at its full rate (1 kHz). A significant low motor speed could block the speed controller. Therefore, *Pool* or *Message* should be chosen as the ACM in the feedback path.

The previous section also showed that a refined torque controller would improve the system performance significantly when the *Channel* was between the two controllers. However, a fault in the outer loop would still block the torque controller, which may cause the current to go out of control and damage the switching transistor in the PWM.

Suppose that a load which was 110 times the initial load (D) was added at the fourth second and then a load, which was 10,000 times D, was added at the sixth second. A fault which blocked the outer loop was forced at 5.5 seconds. If there is a fault in the outer loop, the speed controller will be blocked. The *Channel* between the two controllers will propagate the fault to the inner loop, which will block the torque controller. This will damage the system. The reason will be discussed later.

Figure 5.20 shows the simulation result of the *Channel-Pool* case.

In this Figure, the top waveform is the speed of the motor, the middle one is the current, and the bottom one is the average current. Because there were no differences in the first four seconds between the cases shown in Figure 5.18 and Figure 5.20, only the performance after the fourth second will be discussed.

In the first waveform, the speed was dropped rapidly from the 4th second due to the added load. After reaching the valley, it gradually increased. At 5.5 seconds, where the fault occurred, the speed stopped increasing and levelled off. At the 6th second, when the other significant large load was added, the speed dropped to and stayed at a near 0 level.

In the second waveform, the current was increased rapidly from the 4th second.

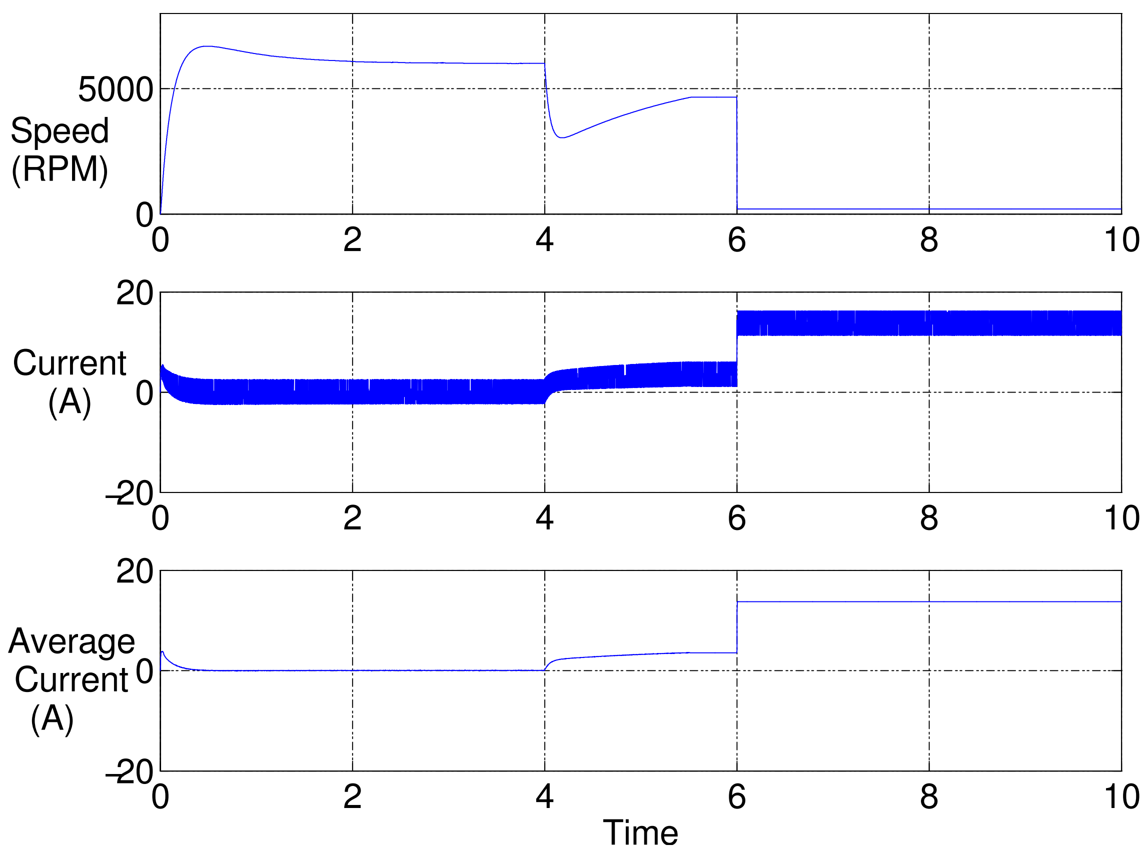


Figure 5.20: Simulation Results for the *Channel-Pool* Case with a Fault

When the speed reached the valley, the current slowed down its increasing rate. At 5.5 seconds, the current stopped increasing and oscillated at a certain level. At the 6th second, the current jumped to and oscillated around 15 amps. The average current in the third waveform illustrated the change of the current more clearly. It can be seen that the current had exceed the limit. The high current would overheat the transistor in the PWM.

Equation 5.2.3 and 5.2.4 must be referred to, to explain the reason. When the load was added in the 4th second, the denominator in Equation 5.2.3 would increase. As a result, the speed would decrease. The decreased speed would increase the numerator in Equation 5.2.4, accordingly increasing the current. The increased current would feed back to Equation 5.2.3 to make the speed rise up again. At the same time, the difference between the reference and the actual speed required a larger current. The torque controller would send the information to the PWM to increase the voltage. In the real life, it can be easily understood: we have to put more energy (voltage) to maintain the motor speed when a load is added.

When the fault occurred, the *Channel* blocked the torque controller. The PWM generated a constant voltage. The constant voltage led to a constant current according to Equation 5.2.4, while the constant current drove a constant speed, which was lower than the reference speed, according to Equation 5.2.3. In the real life, when the driving voltage stops increasing, the speed will stop increasing at the same time.

At the 6th second, an extremely large load was added. This load almost stopped the rotation of the motor. The speed nearly dropped down to 0. Because the torque controller was blocked by the *Channel*, the applied voltage could not be updated. The fall of the speed increased the current, according to Equation 5.2.4, to a significant large value. This caused a fatal damage to the system.

As known, the torque controller is used to maintain the current under control. If the fault had not been propagated to the torque controller, the system would not be damaged by the bursting current. To stop this propagation, a *Pool* could be chosen to play the role of delivering data between the two controllers.

Figure 5.21 shows the resulting waveforms of the *Pool*-*Pool* case.

The current in this figure oscillated at a level under 5 amps after the huge load was added at the 6th second. It shows that the *Pool* did stop the propagation of the fault.

Another fact was that the amplitude of the oscillation was also decreased after the 6th second. To explain this, we have to refer to Equation 5.2.4 again.

To maintain the current under control, the torque controller had to reduce the applied voltage to counteract the decreased back EMF to respond to the speed fell. As known, the applied voltage is an equivalent voltage generated by switching between 0 volts and 24 volts in the PWM. Reducing the voltage was actually de-

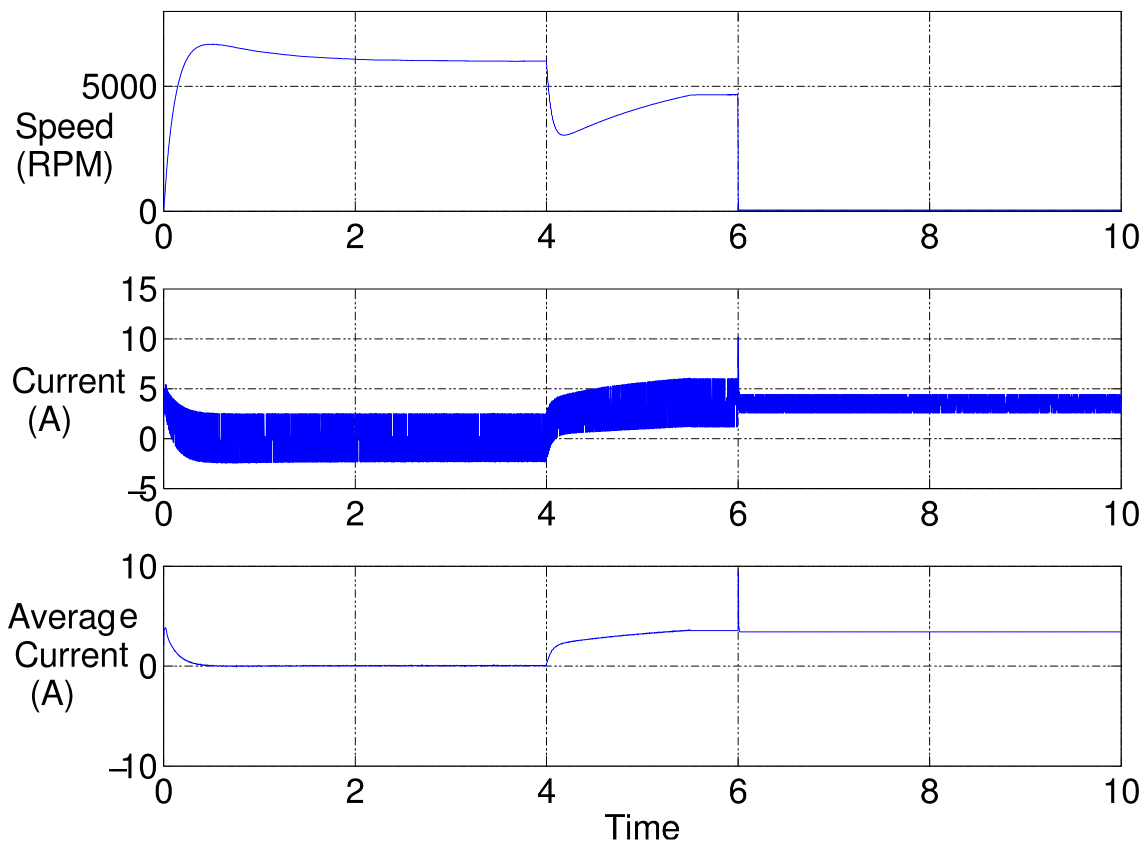


Figure 5.21: Simulation Results for the *Pool-Pool* Case with a Fault

creasing the pulse width. When the pulse width decreased, the charging time for the current would also be decreased. With the same time constant, it appeared as reduced amplitude.

5.6 Further Discussions

The simulation results in this chapter showed that, in this particular example, all the *Pool* type ACMs could be replaced by the *Message* type ones, and all the *Channel*s could be replaced by the *Signal*s. In this example, the writers are slower than the reader, which ensures that overwriting would never happen, while rereading occurs frequently. The overwriting property of the ACMs is not important. Without considering the overwriting property, a *Pool* is the same as a *Message*, and a *Channel* is the same as a *Signal*.

However, this is not always the case. For instance, if the reference speed increased to 18000 RPM, the asynchronous ADC may deliver the data items around 1.8 kHz, which is faster than the rate of the speed controller. In this case, the performance of the system with inclusion of the four type ACMs into the feedback link may vary.

This case was not discussed because an increase of the reference speed may accompany the need of increasing the frequency of the speed controller.

5.7 Conclusions

In this chapter, a brushless DC motor control system was used as an example to investigate the effect of including various ACMs into a control system, especially when analogue parts are present.

The following conclusions could be drawn from this chapter. The number of the ACMs needed in a system is determined by the number of time domains existing in this system and how these time domains connected with each other. A handshake interface needs to be added to the components connected with ACMs. ACMs could play the role of delivering data items between the components running at different speeds in a control system. And the most importantly, the *Pool* type ACM has the ability to prevent a fault from propagation which could protect the control system from being damaged.

Chapter 6

Conclusions and Future Work

As important components for asynchronous communications, the ACMs were introduced [Sim90], studied [Sim92, XC02, Cla00, DC01, Dav97] and implemented [SXY00b, Sha03, XYS$^+$00, YX01]. The main aims of this thesis are building ACMs at the application level and investigating the effects when ACMs are included in a control system.

To build ACMs specified by Petri nets, a method of converting Petri nets to Stateflow models is proposed. A 3-slot Signal was modelled using this method. Because the model built by this method is concentrating on the detail of the internal structure, which is complex and not necessary when investigating applications, another modelling method was proposed. By this method, ACMs are modelled according to the algorithms. The standard ACMs (*Signal, Channel, Message* and *Pool*) and the buffered ones are modelled using this approach, and studied. With the models, an investigation of ACM applications in control systems is proposed.

6.1 Conclusions

ACMs are often described in Petri nets. To model ACMs, a conversion method from Petri nets to Stateflow was proposed in Chapter 3. Petri nets models are built by linking basic connections together. These connections include: linear connections, forks, joins, choices and merges. The conversion approach is based on successfully converting these basic connections. Because of fundamental differences, the direct conversion is only suitable for well-formed Petri nets. Conversion of a pipeline becomes difficult because of the **One Active State Rule** in a Stateflow environment. Instead of translating a transition in Petri nets to a transition in Stateflow, it has to be modelled as two states, one of which represents when the transition is enabled and the other indicates when it is disabled.

The control paths of ACMs could be modelled in MATLAB by two approaches. The first one is translating Petri nets specifications into Stateflow. This approach is based on the method of transforming Petri nets to Stateflow explained in Chapter 3. The models created by this approach are concentrating on the internal structures and closer to hardware implementations, since circuits could be built with the same PN specifications by using direct translation techniques [BY02]. The datapaths require 1-hot input control variables, which also make models complex. A 3-slot *Signal* was used as the example to show that both the model in MATLAB and the hardware circuits implies the same properties. In investigating applications, internal structures are not necessary to know and complex models are not required if simple ones exist, therefore, the other approach was proposed for building simple models.

The second approach is building models directly from algorithms. This approach is focused on the behaviours of ACMs. With analysing algorithms, information exchange pairs between reader, writer and datapath could be found. Those pairs for referencing and updating control variables are simply modelled as comparisons and / or assignments. Other pairs are modelled with handshakes. After linking the handshakes and inserting the corresponding control variables, the controller is built. The datapath only depends on the number of slots in the memory. A 3-slot *Signal* shares the same memory structure with a 3-slot *Pool*. Furthermore, the datapaths of 3-slot ACMs are all the same. This applies to all the 2-slot and all the 4-slot

ACMs. The three types of memory models described in Chapter 4 gave all of the possibilities for ACM datapaths. As a result, the memory models can be taken as components in a library while modelling ACMs. A 2-slot *Channel*, a 3-slot *Signal* and a 4-slot *Pool* were modelled with this approach. The simulation results showed that these models work as they were expected to.

To investigate buffered ACMs, the same method was applied to build models. Buffered ACMs could be described in two ways: a global view and a modular design. The former one centralised the control path while the latter distributed it into identical cells. Increasing or decreasing the size of the buffer will result in the redesigning of the datapath for the global view model, and simply adding another cell for the modular design one. Because the algorithm for the latter is much more complex than that for the former, it is much easier to build a global view model in MATLAB. However, a modular design model is more suitable for hardware designs since redesigning always costs extra.

An RR-BB ACM was built in both the global view form and the modular design structure. Both of the two models showed exactly the same properties. These properties include: the writer writes data items to the buffer until the ACM is full, and, at this time, it does not respond to further requests until the state is changed. The reader reads the data items in the buffer in order until the ACM is empty, and it re-reads the latest item until a new one is available. It gives the best performance, best data continuity and average data latency, when the speed difference between the reader and the writer is small.

Additional cells normally increase the data continuity while decreasing the data latency. However, after the cells increase to a certain number, the additional cells will not improve the performance, and can even increase the data latency and worsen the data freshness when the writer is faster.

The investigation on the effect of including ACMs into a brushless DC motor system was proposed in Chapter 5. The inclusion of ACMs would introduce a delay and / or temporary block to the system. Delays could be compromised by adjusting the coefficients of controllers. However, temporary blocking may result in damage to the system. In the brushless DC motor system, this appeared as an increased current which could burn out the switching transistor in the PWM. Because *Pool* type ACMs would not introduce any blocking, they could be used as the components delivering data items without blocking between processes which execute at different speeds. In the example in this thesis, *Message*s played the same role as *Pool*s did. This is because, in this specific example, the write processes were always slower than the read ones, Messages would not introduce blocking either in this case. According to this fact, it could be said that what type of ACMs to be chosen is determined by the specifications of systems. If non-blocking is the only concern of a system, the choice of ACMs type is determined by the speed of the read and writer processes. If the writer is definitely faster than the reader, a *Signal* could be used to handle the communication. If the reader is definitely faster than the writer, a *Channel* could be used to handle the communication. If it is neither of the extreme cases, a *Pool* could be used.

6.2 Future Work

The potential for future research in the areas related to this thesis include several aspects.

The direct translation approach from Petri nets model to a Stateflow model proposed in Chapter 3 is still manual. Zhou [Zho03] developed a tool for converting Petri nets to Stateflows for linear connection in her Master's thesis. An STG [WB99, Yak92] format textual file is input as a Petri net model. A corresponding Simulink model is generated. However, only the linear connection was discussed in the thesis. Much more study on the complex connections needs to be done for fully automatic conversions. A successful tool for the Petri net to Stateflow conversion would also generate the MATLAB models of ACM automatically from a Petri net specification.

The completion of a tool for algorithm to MATLAB model translations would be an extended piece of work for ACM modelling. To do this, a standard format for ACM algorithms should be set up. Although the step by step modelling method has been developed, the formulated one still needs to be investigated. Combined with the support from the automatic ACM algorithm derivation, another piece of future work to be done, an ACM model generator could be implemented.

Only an RR-BB ACM, which only permits re-reading to occur, was discussed in investigating buffered ACMs. The over-writing strategy requires 2 data slots per cell rather than 1 [XHC+04]. This makes the algorithms and modelling more complex. Therefore, more investigations on different over-writing and re-reading strategies should be carried out. Furthermore, a complete library including both global view and modular design models for all the four types of buffered ACMs needs to be set up. The implementation of these into hardware is also an incomplete task.

In Chapter 5, the effects of inclusion of ACMs into a control system were studied. However, the only consideration of the example was non-blocking, and the write processes in the system were always slower than the read ones. Further investigation should be focused on the cases that the write process is faster and which there is no simple relationship between the rates of two processes. The systems may also allow a degree of blocking and non-freshness, etc. The investigation is still based on the experimental results. Theoretical system analysis methods when ACMs are embedded in also require further research.

Appendix A

Testbench for the Cadence Simulation

This appendix lists the Verilog codes of the testbench for simulation of the 3-slot *Signal* in Cadence.

```
// Verilog stimulus file.
// Please do not create a module in this file.
// Default verilog stimulus.
integer seed1, seed2, seed3;
real c, d;
parameter mean1=1978, mean2=1978;
reg [0:7] tmp[0:2];
integer n;
integer n0,n1,n2,n3;
initial #700$finish;                    // similation time is 700ns
// initialise the input data item, the requests and the seeds for
the distributions
initial
 begin
```

```
     Datain[0:7] = 8'b00000000;
     seed1=34;
     seed2=443;
     seed3=5654;
     clear = 1'b1;
     nDatain[0:7] = 8'b11111111;
     read_start = 1'b0;
     write_start = 1'b0;
     #1.11 clear=0;
    end
   always @ (Datain[0] or Datain[1] or Datain[2] or Datain[3] or
Datain[4] or Datain[5] or Datain[6] or Datain[7])
    begin
     nDatain[0:7]=~Datain[0:7];
    end
   always
    begin
     forever
      begin
       wait(write_done==0&&write_start==0)
              //the intervals in the write cycle
        begin c=$dist_exponential(seed1,mean1)/1000.0;
              // generate a new data item
              # c Datain[0:7]=Datain[0:7]+1;
              // request for a new write
              write_start=1;
        end
      end
```

```
 end
always
 begin
  forever
   begin
    wait(read_done==0&&read_start==0)
     begin
      // the intervals in the read cycle
      d=$dist_exponential(seed2,mean2)/1000.0;
      // request for a new read
      # d read_start=1;
     end
   end
 end
// release the read requests
always @ (posedge read_done)
  #0.01 read_start=0;
// release the write requests
always @ (posedge write_done)
  # 0.01 write_start=0;
```

Bibliography

[ASFR03] E. Allier, G. Sicard, L. Fesquet, and M. Renaudin. A new class of asynchronous A/D converters based on time quantization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 196–205. IEEE Computer Society Press, May 2003.

[Bai00] W. J. Bainbridge. *Asynchronous System-on-Chip Interconnect.* PhD thesis, Department of Computer Science, University of Manchester, March 2000.

[BE00] A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.

[BKR$^+$91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[Bla96] John Blaiklock. Case study 1: The motor controller - generic specification project deliverable 2. Technical report, Dept. of Electrical and Electronic Engineering., Univ. of Newcastle upon Tyne, UK, April 1996.

[BM91] Peter Beerel and Teresa Meng. Semi-modularity and self-diagnostic asynchronous control circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI*, pages 103–117. MIT Press, March 1991.

[Boh96] George A. Bohoris. Trend testing in reliability engineering. *International Journal of Quality and Reliability Management*, 13:45–54, March 1996.

[Bru02] Michael E. Brumbach. *Electronic Variable Speed Drives (2nd Edition)*. Thomson Delmar Learning, January 2002.

[BS88] C. H. (Kees) van Berkel and Ronald W. J. J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 157–162. IEEE Computer Society Press, 1988.

[BS89] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 262–265. IEEE Computer Society Press, November 1989.

[Bur04] R Burch. Monitoring and optimizing pid loop performance. In *ISA Annual Meeting, Houston*, 2004.

[BY02] A. Bystrov and A. Yakovlev. Asynchronous circuit synthesis by direct mapping: Interfacing to environment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 127–136, April 2002.

[Cha84] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, October 1984.

[CKK$^+$96] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In

XI Conference on Design of Integrated Circuits and Systems, Barcelona, November 1996.

[Cla00] Ian G. Clark. *A unified approach to the study of asynchronous communication mechanisms in real time systems.* PhD thesis, London University, King's College, May 2000.

[Dav77] René David. Modular design of asynchronous circuits defined by graphs. *IEEE Transactions on Computers*, 26(8):727–737, August 1977.

[Dav78] A. Davis. The architecture and system method of ddm-1: A recursively-structured data driven machine. In *Fifth Annual Symposium on Computer Architecture*, 1978.

[Dav97] A.C. Davies. Asynchronous communications between locally synchronous subsystems. In *Proc. Polish - Czech - Hungarian Workshop on Circuit Theory, Signal Processing and Applications*, pages 75–80, September 1997.

[DC01] A.C. Davies and I. G. Clark. Asynchronous communications without waiting: from the concept to the hardware. In *Applied Electronics 2001 conference, Plzen, Czech Republic*, pages 55–59, September 2001.

[DE95] Jörg Desel and Javier Esparza. *Free choice Petri nets.* Cambridge University Press, New York, NY, USA, 1995.

[Dea92] Mark E. Dean. *STRiP: A Self-Timed RISC Processor Architecture.* PhD thesis, Stanford University, 1992.

[DJ01] Jörg Desel and Gabriel Juhás. "what is a petri net?". In *Unifying Petri Nets*, pages 1–25, 2001.

[DN95] Al Davis and Steven M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[Ebe91] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[Esh05] R. Eshuis. Statecharting petri nets. In *Beta Working Paper (Int. rep. WP-153).*, 2005.

[FNT$^+$99] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[FPEN06] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems.* Pearson Education, Inc, 5 edition, 2006.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HBL$^+$99] J. M. Hoke, P. W. Bond, T. Lo, F. S. Pidala, and G. Steinbrueck. Self-timed interface for S/390 I/O subsystem interconnection. *IBM Journal of Research and Development*, 43(5/6):829–846, 1999.

[HC03] Fei Hao and Graeme Chester. Acms in matlab. In *14th UK Asynchronous Forum*, June 2003.

[HP02] N. Henderson and S Paynter. The formal classification and verification of simpson's 4-slot asynchronous communication mechanism. In *the International Symposium of Formal Methods Europe, FME 2002: Formal Methods - Getting IT Right*, 2002.

[Huf64] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.

[HXC$^+$04a] Fei Hao, Fei Xia, Graeme Chester, Alex Yakovlev, and Ian Clark. Matlab models of acms in control systems. In *1st International Conference on Informatics in Control, Automation and Robotics (ICINCO-2004)*, volume 3, pages 54–61, August 2004.

[HXC$^+$04b] Fei Hao, Fei Xia, Ian Clark, Alex Yakovlev, and Graeme Chester. Rr-bb algorithm models in matlab. In *15th UK Asynchronous Forum*, January 2004.

[HXCY04] Fei Hao, Fei Xia, Graeme Chester, and Alex Yakovlev. An acm application in broom balancer. In *1st UK Embedded Forum*, pages 176–183, October 2004.

[HYC$^+$03] Fei Hao, Alex Yakovlev, Graeme Chester, Fei Xia, Ian Clark, and Delong Shang. Implementation of a three-slot acm. In *Postgraduate Research Conference in Electronics, Photonics, Communications and Software*, April 2003.

[ITR05] *ITRS: http://www.itrs.net/Common/2005ITRS/Home2005.htm*, 2005.

[KAJ91] D. J. Kinniment, P.P. Acarnley, and A.G. Jack. An integrated circuit controller for brushless dc drives. In *European Power Electronics Conference*, pages 111–116, 1991.

[KEM03] Clinton Kelly, Virantha Ekanayake, and Rajit Manohar. SNAP: A sensor-network asynchronous processor. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 24–33. IEEE Computer Society Press, May 2003.

[KKAJ90] E. Kappos, D.J. Kinniment, P.P. Acarnley, and A.G. Jack. Design of an integrated circuit controller for brushless dc drives. In *Power Electronics and Variable-Speed Drives, 1991., Fourth International Conference on*, pages 336–341, July 1990.

[KKV87] Lefteris M. Kirousis, Evangelos Kranakis, and Paul M. B. Vitányi. Atomic multireader register. In Jan van Leeuwen, editor, *Distributed algorithms*, volume 312 of *Lecture Notes in Computer Science*, pages 278–296, Jul 1987.

[KYG00] David Kinniment, Alex Yakovlev, and Bo Gao. Synchronous and asynchronous A-D conversion. *IEEE Transactions on VLSI Systems*, 8(2):217–220, April 2000.

[L.86] Devroye L. *Nonuniform Random Variate Generation.* Springer Verlag, New York., 1986.

[Lam86] L. Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[Liu97] J. Liu. *Arithmetic and control components for an asynchronous micro-processor.* PhD thesis, Department of Computer Science, University of Manchester, 1997.

[LPI01] Pasi Liljeberg, Juha Plosila, and Jouni Isoaho. Asynchronous interface for locally clocked modules in ULSI systems. In *Proc. International Symposium on Circuits and Systems*, volume 4, pages 170–173, 2001.

[Mar90a] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[Mar90b] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[MATa] *Mathworks: MATLAB: http://www.mathworks.com/products/matlab/?BB=1.*

[MATb] *Mathworks: Migrating from MATRIXx to MathWorks Prodcuts: http://www.mathworks.com/services/consulting/areas/matrixx.html.*

[MATc] *National Instruments: MATRIXx: http://www.ni.com/matrixx/.*

[MB59] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[McC65] E. J. McCluskey. *Introduction to the Theory of Switching Circuits.* McGraw-Hill, New York, 1965.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mul62] David E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1962.

[Mul67] D. E. Muller. The general synthesis problem for asynchronous digital networks. In *Annual Symposium on Switching and Automata Theory*, New York, 1967.

[Mye01] Chris Myers. *Asynchronous Circuit Design.* John Wiley & Sons, 2001.

[Pee96] Ad M. G. Peeters. *Single-Rail Handshake Circuits.* PhD thesis, Eindhoven University of Technology, June 1996.

[Pet62] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *IFIP Congress*, pages 386–390, 1962.

[Pet73] C. A Petri. Concepts of net theory. *Mathematical foundations of concepts of computer science, High Tatras*, pages 137–146, 1973.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[PFF96] O. A. Petlin, C. Farnsworth, and S. B. Furber. Design for testability of an asynchronous adder. *Design and Test of Asynchronous Systems, IEE Colloquium on*, 1996.

[SBY03] D. Sokolov, A. Bystrov, and A. Yakovlev. STG optimisation in the direct mapping of asynchronous circuits. In *Proc. Design, Automation and Test in Europe (DATE).* IEEE Computer Society Press, March 2003.

[Sei71] Charles L. Seitz. *Graph Representations for Logical Machines.* PhD thesis, MIT Press, January 1971.

[Sei80] Charles L. Seitz. Ideas about arbiters. *Lambda*, 1(1, First Quarter):10–14, 1980.

[SF01] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, 2001.

[Sha03] Delong Shang. *Asynchronous Communication Circuits: Design, Test and Synthesis.* PhD thesis, School of EECE, University of Newcastle upon Tyne, April 2003.

[Sim90] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Computers and Digital Techniques*, 137(1):17–30, January 1990.

[Sim92] H. R. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings, Computers and Digital Techniques*, 139(1):35–49, January 1992.

[Sim94] Hugo R Simpson. Methodological and Notational Conventions in DORIS Real-Time Networks. IED Supporting Predictable Implementation of Requirements in Timing and Safety (SPRINTS) Deliverable, 1994.

[Sim03] Hugo R Simpson. Protocols for process interaction. *IEE Proc.-Comput. Digit. Tech.*, 150(3), May 2003.

[Sok06] D. Sokolov. *Automated synthesis of asynchronous circuit using direct mapping for control and data paths*. PhD thesis, Newcastle University, UK, January 2006.

[SSV96] N. Sayiner, H.V. Sorensen, and T.R. Viswanathan. A level-crossing sampling scheme for a/d conversion. *IEEE Transactions on Circuits and Systems II*, 43(4):335–339, April 1996.

[Sta] *Mathworks: Stateflow: http://www.mathworks.com/products/stateflow/?BB=1.*

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[SXY00a] D. Shang, F. Xia, and A. Yakovlev. Asynchronous circuit synthesis via direct translation. In *Proc. International Symposium on Circuits and Systems*, volume 3, pages 369–372, May 2000.

[SXY00b] D. Shang, F. Xia, and A. Yakovlev. An implementation of a three-slot asynchronous communication mechanism using self-timed circuits. In Alex Yakovlev and Reinder Nouta, editors, *Asynchronous Interfaces: Tools, Techniques, and Implementations*, pages 37–44, July 2000.

[Tro89] John Tromp. How to construct an atomic variable (extended abstract). In Jean-Claude Bermond, editor, *Distributed algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 292–302, Sept 1989.

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[Ung70] S. H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. In *Annual Symposium on Switching and Automata Theory*, pages 114–121. IEEE Computer Society Press, 1970.

[Ung71] Stephen H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, 20(12):1437–1444, December 1971.

[Var73] Victor Varshavsky. *Collective Behavior of Automata.* 1973.

[Ver88] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[VM96a] V. I. Varshavsky and V. B. Marakhovsky. Asynchronous control device design by net model behavior simulation. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets 1996*, volume 1091 of *Lecture Notes in Computer Science*, pages 497–515. Springer-Verlag, June 1996.

[VM96b] V. I. Varshavsky and V. B. Marakhovsky. Hardware support for discrete event coordination. In J. Billington and W. Reisig, editors, *Proc. of*

International Workshop on Discrete Event Systems(WODES'96), pages 332–340, August 1996.

[VMS95] Victor I. Varshavsky, Vyacheslav B. Marakhovsky, and Vadim V. Smolensky. Designing self-timed devices using the finite automaton model. *IEEE Design & Test of Computers*, 12(1):14–23, Spring 1995.

[WB99] R. Wollowski and J. Beister. Comprehensive causal specification of asynchronous circuit behaviour: a generalized STG. In *Proc. of the Workshop Hardware Design and Petri Nets (within the International Conference on Application and Theory of Petri Nets)*, pages 149–168, June 1999.

[WB00] R. Wollowski and J. Beister. Comprehensive causal specification of asynchronous controller and arbiter behaviour. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 3–32. Kluwer Academic Publishers, March 2000.

[XC99] F. Xia and I. Clark. Studying the three-slot asynchronous communication mechanism, 1999.

[XC00] F. Xia and I. Clark. Complementing role models with Petri nets in studying asynchronous data communications. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 33–50. Kluwer Academic Publishers, March 2000.

[XC02] Fei Xia and Ian Clark. Algorithms for signal and message asynchronous communication mechanisms and their analysis. *Fundam. Inf.*, 50(2):205–222, 2002.

[XHC$^+$04] Fei Xia, Fei Hao, Ian G. Clark, Alexandre Yakovlev, and E. Graeme Chester. Buffered asynchronous communication mechanisms. In *ACSD*, pages 36–46, 2004.

[XHC$^+$06] Fei Xia, Fei Hao, Ian G. Clark, Alex Yakovlev, and E. Graeme Chester. Buffered asynchronous communication mechanisms. *Fundam. Inform.*, 70(1-2):155–170, 2006.

[Xia00] Fei Xia. *Supporting the MASCOT method with Petri net techniques for real-time systems development.* PhD thesis, London University, King's College, January 2000.

[Xil] Xilinx. System generator for dsp. http://www.xilinx.com/ise/optional prod/system generator.htm.

[XYCS02] Fei Xia, Alex V. Yakovlev, Ian G. Clark, and Delong Shang. Data communication in systems with heterogeneous timing. *j-IEEE-MICRO*, 22(6):58–69, November/December 2002.

[XYS$^+$00] F. Xia, A. Yakovlev, D. Shang, A. Bystrov, A. Koelmans, and D. J. Kinniment. Asynchronous communication mechanisms using self-timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 150–159. IEEE Computer Society Press, April 2000.

[Yak92] Alexandre V. Yakovlev. On limitations and extensions of STG model for designing asynchronous control circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 396–400. IEEE Computer Society Press, October 1992.

[YDN92] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer*

Design (ICCD), pages 346–350. IEEE Computer Society Press, October 1992.

[YDS00] J.K. Yook, Tilbury D.M., and N.R Soparkar. Trading computation for bandwidth: reducing communications in distributed control systems using state estimators. In *Proceedings of the 2000 Japan-USA Symposium on Flexible Automation*, 2000.

[YK98] A. V. Yakovlev and A. M. Koelmans. Petri nets and digital hardware design. In *Lectures on Petri Nets II: Applications. Advances in Petri Nets*, volume 1492 of *Lecture Notes in Computer Science*, pages 154–236, 1998.

[YKKL94] A. Yakovlev, M. Kishinevsky, A. Kondratyev, and L. Lavagno. OR causality: modelling and hardware implementation. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 568–587, Zaragosa, Spain, June 1994. Springer-Verlag.

[YLSV96] Alexandre Yakovlev, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. *Formal Methods in System Design*, 9(3):139–188, 1996.

[YTS00] T. Yoneda, D. M. Tilbury, and N. R. Soparkar. A design methodology for distributed control systems to optimize performance in the presence of time delays. In *the American Control Conference*, pages 1959–1964, April 2000.

[YVMS95] A. Yakovlev, V. Varshavsky, V. Marakhovsky, and A. Semenov. Designing an asynchronous pipeline token ring interface. In *Asynchronous*

Design Methodologies, pages 32–41. IEEE Computer Society Press, May 1995.

[YX01] A. Yakovlev and F. Xia. Towards synthesis of asynchronous communication algorithms. In *Workshop on Synthesis of Concurrent Systems, 22nd International conference on application and theory of Petri nets*, pages 48–57. IEEE Computer Society Press, June 2001.

[YXS01] A. Yakovlev, F. Xia, and D. Shang. Synthesis and implementation of a signal-type asynchronous data communication mechanism. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 127–136. IEEE Computer Society Press, March 2001.

[Zho03] Ying Zhou. Petri net modelling in matlab. Master's thesis, School of EECE, Newcastle University, UK, June 2003.

[ZRG04] Jenny Zheng Zhou, Athula Rajapakse, and Aniruddha M. Gole. Effects of control systems time delay on the performance of direct harmonics elimination. In *IEEE Canadian Conference on Electrical and Computer Engineering*, pages 609–612, May 2004.