Formal Complexity-Oriented Performance-Critical Design and Verification Framework

Configurable Communication Systems Perspective

Suleiman Abu Kharmeh

A dissertation submitted to the ***University of Bristol***
in accordance with the requirements for award of the degree of
Doctor of Philosophy *in the* ***Faculty of Engineering***

August 2012

50565

Abstract

This thesis develops a formal framework for the specification, complexity analysis and verification of functional and performance requirements of configurable communication systems and protocols.

The main objective is demonstrating the applicability of the proposed framework for the modelling and verification of a realistic system. Design-for-Verification principles are demonstrated, such as the semantic analysis and decomposition of complex and intertwined requirements, and the subsequent composition of orthogonal functional units with manageable complexities. *Tock-CSP* was used to model those functional units and their interfaces. Analysis of the underlying state machines of the modelled system resulted in the identification of complexity and scalability issues. Then, through the development and application of formal complexity analysis techniques for state machines, modelling optimisations were possible. Complexity issues of the model-checker were also identified and resolved. Adoption challenges of formal methods were addressed by the development of suitable specification and verification interfaces. The properties of the configurable system and its ISA-*Oriented* interface were verified using various refinement models including the Tau Priority Model. Finally, the conformance of the ISA-*Oriented Specification* methodology to abstract specifications of selected communication protocols was also verified.

This thesis is the first to devise mathematical techniques for expressing and analysing the state-space complexity of formal models, the first to develop and use waveform visualisation for the analysis of timing specifications of formal models, and the first application of the newly released Tau Priority Model.

لِأُمِّي ثُمَّ لِأُمِّي

ثُمَّ لِأُمِّي ثُمَّ لأَبِي

وَآخِرُ دَعْوَاهُمْ أَنِ ٱلْحَمْدُ

لِلّٰهِ رَبِّ ٱلْعَالَمِينَ

Acknowledgements

I am greatly indebted to my parents Naeem and Fatemah for their everlasting help, dedication, and support. I am very proud and privileged to have them as my parents. All my respect and gratitude goes to both of you.

I am thankful to my supervisors Dr. Kerstin Eder and Professor David May for seeing me through this PhD. Their help is greatly appreciated. The unquestionable belief that they had in my abilities and motivation was absolutely vital in tough and challenging times. Nevertheless, they always managed to provide constructive, sceptical and critical suggestions, which were an invaluable source of challenges and motivation for improvement.

I am also very thankful to my examiners, Professor Dave Cliff, who bravely took the challenge of organising a three-examiner viva, Professor Michael Leuschel, whose advice and help throughout and after the viva was vital in highlighting and improving the material contained in this thesis, and Dr. Helen Treharne, whose constructive arguments and recommendations were also very helpful.

Special thanks goes to Professor Bill Roscoe of Oxford Computer Laboratory for genuinely helpful advice and also for granting access to the source code of their model-checker.

Special thanks also goes to Dr. Mike Barton for his help and support throughout my presence in Bristol. Thanks to Miss Sophie Benoit who is always very helpful and glowing with optimism. Thanks to my colleagues in the Computer Science department. In particular, Dr. Essam Ghadafi and Steven Kerrison for their help in proofreading some chapters of my thesis. Thanks to all the helpful staff of the Engineering Faculty. Finally, thanks to all the staff and students of University of Bristol who helped, advised and most importantly entertained my presence over the years.

Author's Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award.

Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such.

Any views expressed in the dissertation are those of the author.

Date: January 20, 2014

Signature: _____

Suleiman Abu Kharmeh

Contents

List of Figures

List of Tables

Nomenclature

I^2C Inter-Integrated Circuit Protocol

I^2S Inter-Integrated Circuit Sound Protocol

CSP_M Machine-Readable dialect of CSP

ABP Alternating Bit Protocol

API Application Programming Interface

ARC Adelaide Refinement Checker

ARM Advanced RISC Machines

ASIC Application Specific Integrated Circuit

BDD Binary Decision Diagrams

BEEM BEnchmarks for Explicit Model-Checkers

CAN Controller Area Network

CSP Communicating Sequential Processes

DDC Data Dependent Control

DDR Double Data Rate

DfV Design-for-Verification

DiVinE Distributed Verification Environment

FDR Failures-Divergence Refinement

FPGA Field-Programmable Gate Array

FSM Finite-State Machine

GRG Generalised Reduced Gradient algorithm

GUI Graphical User Interface

HDL Hardware Description Language

I/O Input/Output

ILP Integrated Layer Processing

ISA Instruction Set Architecture

ISM Indexed State Machine

ISO International Organization for Standardization

MII Media Independent Interface

NOP No-Operation

NRZ Non-Return-to-Zero

OMI Open Microprocessor Systems Initiative

OSI Open Systems Interconnection

PAT Process Analysis Toolkit

PHY Physical Layer Control

RAM Random Access Memory

RISC Reduced Instruction Set Computing

RZ Return-to-Zero

SAWP Stop-and-Wait Protocol

SBC Serialisation and Buffering Control

SoC System-on-Chip

SPI Serial Peripheral Interface

STC Synchronous and Timing Control

SWP Sliding Window Protocol

TCL Tool Command Language

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

VCD Value Change Dump

VCP Virtual Channel Processor

CHAPTER 1

Introduction

Communication interfaces are an important aspect of modern embedded hardware systems. They are becoming highly significant, especially in the context of the emerging multi-core and many-core platforms. System designers expect Input/Output (I/O) interfaces both internal and external to the System-on-Chip (SoC) to support a variety of existing protocols such as UART, SPI, CAN, MII, I^2C, I^2S, and USB [4–10] to mention a few. These interfaces are expected to implement a complex set of functional and timing aspects of the target protocols and thereby reduce the communication and processing demands on the processor or network-on-chip.

Normal practice would be to use fixed hardware modules to implement each of these interfaces. An alternative approach would be to use generic interfaces configured by software. This would allow these interfaces to implement a large and varying set of functions for each protocol, to be adaptable for any protocol changes over time or to accommodate new protocols.

Because the communication interface is able to communicate through many protocols, there is nothing that stops it from dynamically deciding which protocol it uses at run-time. This will be useful in the context of fault-tolerant systems where one interface could adapt its functions according to the system it is connected to and/or the environment it is interacting with.

Hardware designs of the configurable interfaces must meet a complex set of functional and performance requirements while maintaining configurability. The ad-hoc approach to achieving this goal would be to integrate all the known functionalities of identified protocols into a single hardware module. This also includes a speculative

addition of requirements, which might be required for the support of future communication protocols. This happens through the evolution of legacy interfaces with added functionality over time, which adds some unwanted "historical" complexity such as the case with the Universal Serial Bus (USB) protocol. This ad-hoc approach has a significant verification overhead, because all of the functional and performance requirements are intertwined in a single hardware specification and implementation.

In practice, a more structured and tractable approach would be to separate requirements into functionally-independent configurable hardware blocks. Different static constructions of these blocks could be made to meet the requirements of the needed communication protocol and thus constructing the configurable communication system. This communication system can then be dynamically configured to customise an I/O interface towards a particular communication function. A complete communication protocol is thus characterised by the static construction of the configurable hardware blocks coupled with a set of configurations of these blocks. This careful consideration of the design and verification issues at an early stage of the design cycle is called Design-for-Verification (DfV).

This DfV methodology provides the most tractable approach for the design of such a complex system by structuring the seemingly dependent, intertwined and complex design requirements into relatively simple blocks with specific requirements to be verified. However, the overall system resulting from the composition of those functional blocks is still complex and poses verification challenges. In particular, the nature of the complexity growth of the combined system and its state space as more functional blocks are introduced. An important issue is how the system complexity growth is related to the growth of the functional specifications of the system.

In practice, the complexity of the abstract functional specification, as well as the complexity of the proposed implementation of configurable communication blocks grows exponentially as additional blocks are added. Traditional simulation-based design and verification approaches are inherently intractable for verifying and analysing the expected complexity of the proposed system.

1.1 Background

Gregor von Bochmann [11] in his paper titled "Finite State Description of Communication Protocols" has suggested a method for specifying communication protocols in terms of a number of finite-state machines. A concept of "adjoint states" for achieving synchronisation between communication entities is discussed. Various "protocol

validation" techniques are suggested. The limitation of the approach is highlighted and shared by all finite state-based approaches, namely: the state-space explosion problem. Finally, the approach is demonstrated by the modelling and verification of the Alternating Bit Protocol (ABP). The shortcomings of this approach are the lack of any provision for automation through the use of modelling and verification tools. Such tools would be of great benefit for applying the approach to other problems and protocols.

In a thesis titled "A Better Way to Design Communication Protocols" Müffke [12] presented a framework for specifying communication protocols. The framework does not provide any practical results on how useful the approach might be for the verification of communication protocols. It does not address real-time aspects. It also falls short from providing or even describing an automation process (i.e. tools) for the suggested framework.

Böhm and Melham [13] have presented a refinement approach to designing and verifying communication protocols. It addresses on-chip protocols with a special focus on the AMBA Bus architecture by Advanced RISC Machine (ARM) [14]. The core design step and the following transformations are only relevant to the discussed protocol and much of the work requires manual proofs. Applying such a framework to other protocols would require considerable expertise and effort. It also does not address real-time aspects of protocol specification and verification.

In light of the previous work, the need for a standard and extendible framework for the modelling and verification of both functional and real-time aspects of communication protocols is apparent. Such a framework must have solid theoretical foundations. It must also be accessible to the wider hardware design community. This could be provided through an abstract specification interface that is similar to the well understood and widely implemented interfaces of Instruction Set Architectures (ISA).

1.2 Problem Statement

A formal specification and verification framework is considered most tractable for analysing multiple communication interfaces, as discussed earlier. This framework must be:

- capable of managing the inherent complexity expected of configurable communication systems and the wide range of possible communication interfaces;

- scalable to allow for larger units of communications and possibly higher performance levels; and
- extendible to enable the support of additional functional specifications and communication interfaces beyond the initially anticipated ones.

This overarching formal design and verification framework for the specification, implementation and verification of configurable communication systems is the main challenge addressed in this thesis. This overarching challenge is further broken down into smaller challenges and objectives as discussed in Section 1.3.

1.3 Research Questions and Objectives

The overall thesis topic is wide and furcating. For this reason, it has been divided into smaller identifiable research questions and objectives. This section lists those research questions and objectives.

Objective 1 *Refinement-Based Formal Modelling and Verification Framework*

What are the techniques and goals of the formal framework?

Refinement-based modelling and verification techniques proved the ideal methodology for this framework. The goal of the work presented is to provide a platform for the specification, implementation and verification of both well-established communication protocols as well as emerging ones. A modelling framework is needed to allow for accurately modelling communication protocols using the configurable blocks mentioned in Objective 2.

Simple high-level communication abstractions or properties are defined. These describe various aspects of a communication system. These abstractions can then be used as high-level specifications in the verification process of the framework and its configurations. They can also be used in the verification of other framework-independent abstract descriptions of the communication protocols in question.

Then, by using refinement techniques, a top-level abstract model of the system must be refined incrementally by adding implementation details. These increments are of the form of additional functional blocks.

This evolving modelling system must then be verified across (and proved to be a refinement of) the top-level abstract communication specification. Furthermore, not only the overall modelling system is verified across abstract communication models,

but most importantly (and using the same refinement-based model-checking techniques) specific sets of configurations of this system are proven to be a refinement of and equivalent to a specific communication protocol.

Objective 2 *Analysis of the Requirements of Communication Interfaces*

How would the requirements of the configurable communication system be defined? Are there any functional dependencies between those requirements and how would such dependencies be analysed in a verification framework?

A structured modelling approach was considered most tractable for managing the inherent complexity of the configurable communication system. This involves analysing the protocols and establishing their functional and performance requirements. The requirements must then be analysed for functional independence. This should lead to the specification of functionally independent configurable hardware blocks.

The hierarchical composition of these blocks, suitably configured, provides the overall functional and timing properties of a communication protocol.

Objective 3 *Empirical Analysis: Formal Specification and Modelling*

How would the configurable system be modelled? How would each unit be defined and configured? How would the units be composed into a fully functional and scalable communication system?

The generic I/O communication system should be modelled in a machine-readable format. The choice of modelling techniques should be aided by the suitability of those techniques to the configurable communication system in question, as well as the availability of associated model-checking tools. The produced models for the communication system should take into account the need to configure the system to meet functional and performance requirements of a selected set of communication protocols.

Due to the expected complexity of the proposed system, the models should be produced with scalability and automated model-checking in mind.

Objective 4 *Hardware Presentation and Visualisation Techniques*

What are the desirable modelling and verification user interfaces? What value would visual abstractions add to the overall verification framework?

Tools and extensions exist for model-checking CSP scripts. Predominantly, they all use a CSP_M front-end. Examples are the ProB model-checker [15], Adelaide Refinement Checker (ARC) [16], Process Analysis Toolkit (PAT) [17], but the most

well known CSP model-checker is the Failures-Divergence Refinement (FDR) model-checker [18]. However, there is a lack of support for visual hardware modelling abstractions in the formal CSP model-checking tools available; specifically in FDR.

It is important that the developed formal models, as well as their associated real-time aspects, are presented to hardware engineers using the relevant abstractions integrated into the hardware design tools. For these reasons, relevant hardware presentation and visualisation techniques and tools are needed. It would be ideal if such tools were an integral part of a model-checker.

Objective 5 *Complexity Analysis at Various Abstraction Levels*

How would the complexity, orthogonality and scalability of the system be analysed? How could this complexity analysis be used to reduce the complexity of the modelled system? How could the complexity analysis prove the orthogonality of different configurable units from a configuration parameter or data-type?

The success of the framework was hindered by the lack of support of formal analysis and model-checking tools available (namely FDR) for such a complex, furcating and broad framework. This was especially apparent when objective complexity analysis and management of the exponentially growing system was needed.

The investigation of viable system complexity metrics (e.g. state space, communication space, time and memory performance) is needed. This investigation involves the identification, extraction and analysis of those complexity metrics. The practical support of such an investigation through tool extensions is also needed.

Finally, it would be useful if the complexity and scalability of the FDR tool itself in handling an exponentially growing complex framework was analysed and subsequently optimised.

This lack of support for complexity and scalability analysis at many levels (e.g. modelling framework and the model-checking tools) as the system grows is seen as one of the main challenges facing the adoption of formal model-checking and verification techniques in general and CSP in particular.

Objective 6 *Empirical Analysis: Formal Verification through Model-Checking*

Which verification strategies would be best suited for functional and performance verification of the configurable system? How would specific configurations of the system be identified as a communication protocol's specifications? How would the conformance of communication protocols' configurations be verified with respect to independent communication protocols' specifications?

Selected example protocols (UART and SPI [4, 5]) are to be formally specified by configuring the generic communication system. Using the refinement models of CSP, proof is to be established to confirm that the configured instances of the hardware are indeed valid implementations of the respective abstract specifications of the protocols. This serves as an empirical demonstration of the ability of the framework to be used in modelling and verification of real-world communication protocols.

Objective 7 *Real-time Specification and Verification*

How would the real-time aspects of the configurable system be specified? How would those aspects be used when identifying the configurations of communication protocols? How would such *performance-aware* configurations be verified with abstract performance specifications of the communication protocols?

Real-time performance specifications are part of any real-time communication protocol. The verification of such specifications is essential for any communication system implementing and communicating using those protocols.

A methodology is needed for expressing high-level timing aspects of communication protocols in CSP. This methodology should be used for specifying performance aspects of communication interfaces. Subsequently those performance specifications should be model-checked with respect to the performance properties of the configured system.

The Tau Priority Model initially suggested by Ouaknine [19] has been recently implemented in FDR. Its availability could be essential to the verification of real-time aspects of communication interfaces.

1.4 Publications

Publication 1 Abu Kharmeh et al. [1] gives a brief progress update of the suggested platform. The paper was presented at the 10^{th} International Workshop on Automated Verification of Critical Systems. The paper focuses on the design aspect of a complex communication interface system. At that stage an initial design of the suggested system was introduced and individual properties and functional requirements were investigated.

Publication 2 Abu Kharmeh et al. [2] discusses the full modelling framework. The paper was presented at the 9^{th} International Conference on Formal Modelling and Analysis of Timed Systems. The paper couples *Property-Oriented Specification* techniques with ISA implementation techniques into a new specification methodology

called ISA-*Oriented Specification*. Then the modelling of communication protocols using ISA-*Oriented Specification* is discussed. Finally, the verification of specific functional and performance aspects of the protocols is discussed.

Publication 3 The complexity and scalability of the modelling techniques, the modelled framework and the model-checking tools are of great interest. In a full technical paper, Abu Kharmeh et al. [3] discuss such complexity issues. Objective definitions of *State-Space Complexity* and *State-Space Explosion* were provided by relating them to metrics of the underlying modelled system.

The paper discusses how other metrics could be as useful in analysing hardware complexity and model-checking such as the communication-space and its complexity. Complexity equations are defined and subsequently analysed for their asymptotic behaviour. This gives a clear indication on the scalability of the models. The paper also discusses the practical implication of producing the desired complexity metrics using professional model-checking tools. It uncovers inefficiencies in such tools.

Finally, the paper concludes with a detailed case study of the targeted communication system. It shows how optimisations were achieved by analysing both the detailed communication complexity formulae, as well as their asymptotic aspect. Such optimisations reduced the complexity of the modelled system, while maintaining the same functional behaviour.

[1] S. Abu Kharmeh *et al.*, "Formal Anal. of a Programmable Performance-Critical Processor Communication Interface," in *Proc. 10th AVoCS Int. Workshop*, 2010.

[2] S. Abu Kharmeh *et al.*, "A Design-for-Verification Framework for a Configurable Performance-Critical Communication Interface," in *Proc. 9th Int. Conf. Formal Modeling and Anal. of Timed Syst.* Springer, August 2011, pp. 335–351.

[3] S. Abu Kharmeh *et al.*, "Complexity of Hardware Design and Model-Checking: An Asymptotic Anal. of State Mach. Metrics," University of Bristol, Tech. Rep., 2012.

1.5 Thesis Structure

Figures 1.1 and 1.2 show a high-level outline of the modelling and evaluation chapters of the thesis. The top half of Figure 1.1 outlines Chapter 5, while the bottom half outlines Chapter 3. Figure 1.2 outlines Chapter 6 and how the communication protocols are abstractly defined and used in the verification process. The analysis of the relationship between the different abstraction levels depicted by the *Equivalence* arrows in Figures 1.1 and 1.2 is discussed in Chapter 7.

A brief description of each chapter follows.

Chapter 2 addresses Objective 2 and establishes an independent set of requirements to be used in the modelling stage of the communication system.

Chapter 3 presents a brief description of the CSP modelling language, followed by a brief presentation of the first attempt at modelling the communication system in Section 3.4, which partially addresses Objectives 3 and 7. Then, Section 3.5 demonstrates the capabilities of the State Machine Visualiser developed for analysing the compiled state machines graphically. This partially addresses Objective 4. The chapter concludes with a demonstration of the growing size of the developed models. This exposes the limitation of the capabilities of graph visualisation and basic analysis and the need for a higher-level of complexity analysis for the developed models.

Chapter 4 demonstrates a complexity analysis technique for state machine metrics as discussed in Objective 5. The technique is first demonstrated in Section 4.4 using simple processes from which the state-space and communication-space complexity formulae are developed. In practice, an iterative process was necessary for producing the complexity formulae, which involved the compilation of processes under many different configurations and the extraction of the metrics for those processes each time. For this purpose, the efficiency of the model-checker was hindering the success of the approach and hence required additional investigation. Section 4.5 demonstrates the investigations into the complexity of the model-checker and discusses the optimisation process along with the results. Section 4.6 demonstrates an automated process for developing the complexity formulae for the system models presented in Chapter 3. Section 4.7 demonstrates the asymptotic analysis of the complexity formulae and the use of such analysis in the development of more optimal specifications of the system.

Chapter 5 demonstrates the concrete communication system which concludes Objective 3. This includes the ISA-*Oriented Specification* methodology for configuring the communication system, the specification of the individual functional units and how those are constructed in parallel to build the overall communication system. The

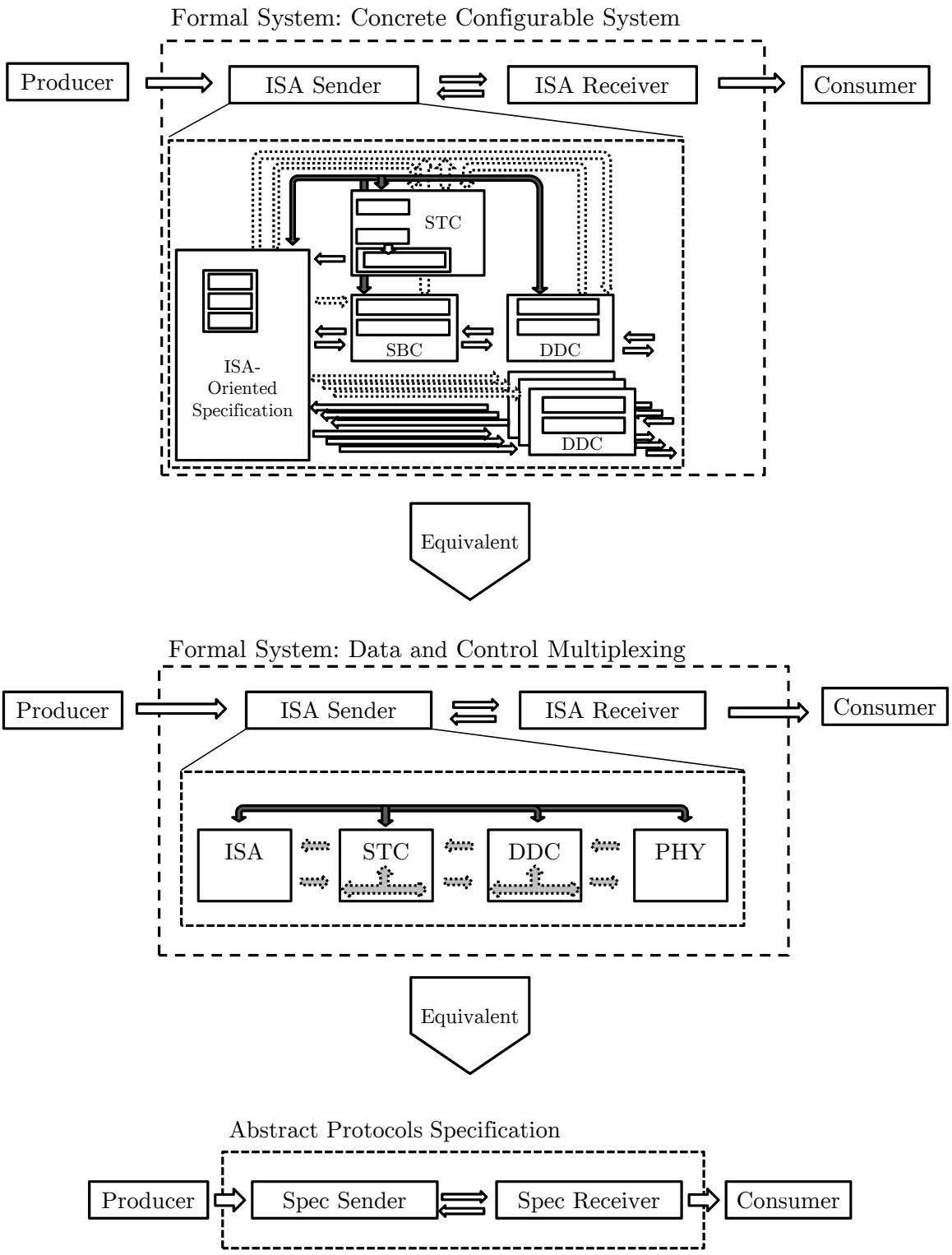


Figure 1.1 Modelling and Verification Block Diagram (Part 1)

modelling of the instruction set in Section 5.3.5 addresses configurability and data communication aspects of the system. It does not aim to model all the details of such

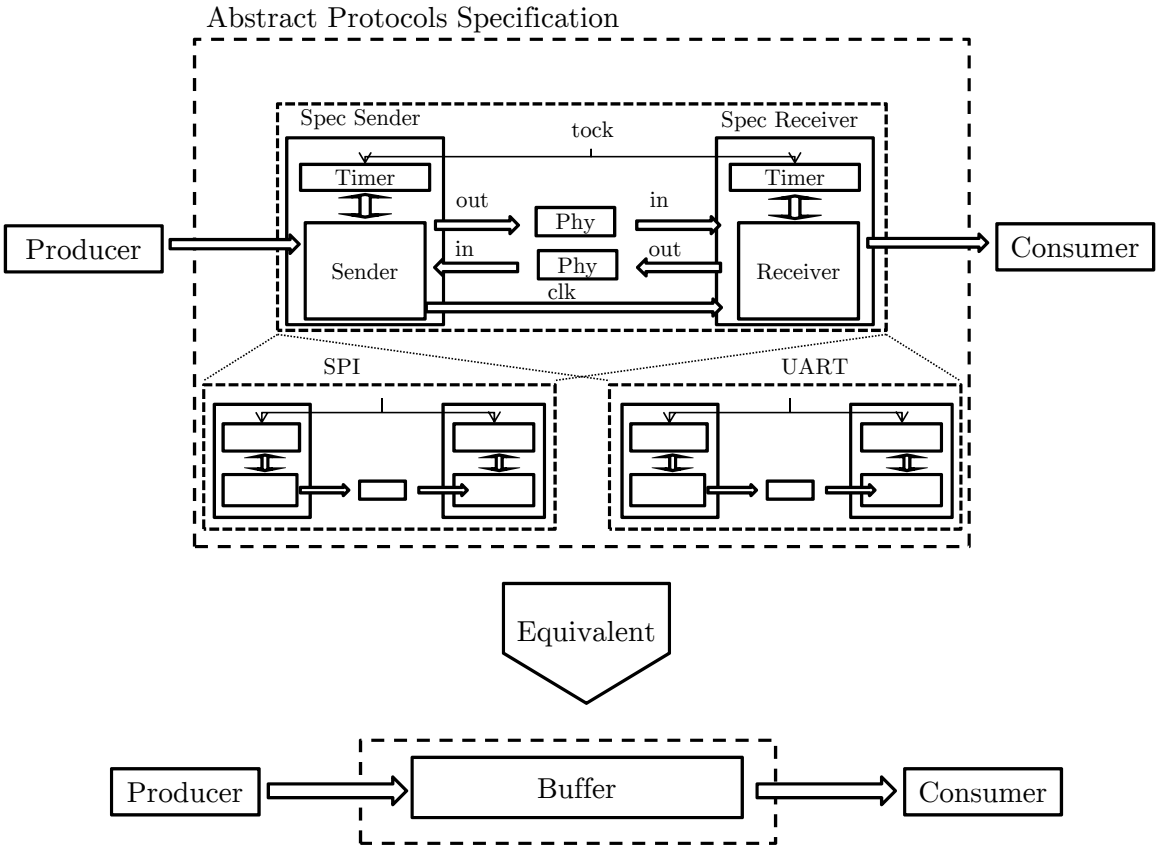


Figure 1.2 Modelling and Verification Block Diagram (Part 2)

an instruction set like the memory model. Such details are out of the scope of this thesis.

Chapter 6 demonstrates two independent specification approaches for communication protocols: an ISA-*Oriented Specification* using the instruction set modelled in Chapter 5 as well as a stand-alone abstract specification of selected communication protocols. By doing so, the chapter partially addresses Objective 6. The two sets of specifications will be instrumental in showcasing the value of the formal framework in Chapter 7.

In Chapter 7, the strength of the overall formal framework is demonstrated when a series of formal properties are proven. Section 7.3 performs basic timing and functional assertions of the communication system. Sections 7.4 and 7.5 check the validity of the ISA-*Oriented Specification*s and *Abstract Specification*s of the protocols separately. Section 7.6 and Section 7.7 demonstrate how individual protocol specifications could be used to construct a complex communication channel and how that channel could be functionally verified. Section 7.8 discusses how specific performance assertions could be carried out. Section 7.8 also discusses the *Tock-CSP Waveform*

Generator which concludes Objective 4. Finally, Section 7.9 demonstrates how the configured communication system conforms to the stand-alone specifications of selected protocols presented in Chapter 6. Through the various levels of functional and performance evaluation, Chapter 7 concludes Objective 6.

Finally, Chapter 8 summarises the thesis and presents the contributions and possible future work.

CHAPTER 2

Requirements Analysis

2.1 Motivation and Chapter Structure

This chapter addresses Objective 2: the analysis of the requirements of communication interfaces. First, the background of this chapter is presented in Section 2.2, then Section 2.3 presents a brief review for a number of communication protocols. This review is essential for constructing a set of requirements that are shared among communication protocols as discussed in Section 2.4. Next, the analysis of the functional independence of those requirements is discussed in Section 2.5. Once a set of orthogonal functional requirements is established, it can be used as a blueprint for the building blocks of the configurable communication system, as discussed in Section 2.6.

2.2 Background

While reasoning about complex communication systems and the need for a structural approach for specifying and modelling communication protocols, one cannot overlook the bigger picture of how complex systems in general are constructed and organised. Simon [20] discusses the organisation of complex systems from a relatively non-technical point of view. The work provides great insight on the commonalities between a very broad class of complex systems. It stresses the fact that most complex systems are constructed in a hierarchical manner. It also discusses briefly the mathematical background for this hierarchical construction. Most importantly, it discusses the "near-decomposability" theory which implies that one "can build a theory of the system at the level of dynamics that is observable, in ignorance of the detailed

structure of dynamics at the next level down, and ignore the very slow interactions at the next level up". Not only does this statement give a historical background to the refinement-based modelling techniques used today, but it also provides real evidence with examples from many complex systems that the best approach to structuring such systems is through hierarchical decomposition. Simon [20] also uses programming alphabets, languages and programs to illustrate these hierarchical decomposition concepts.

With respect to complex communication protocols, the most evident work to the standardisation and decomposition of protocols is the initiative of the International Organization for Standardization (ISO) called the Open Systems Interconnection (OSI) reference model [21]. This reference model provides a general hierarchical decomposition of any communication system for protocols designers on which many communication protocols standards are based. It is an abstract model with little consideration to hardware implementation details. Rather, it emphasises the separation of functional aspects with respect to their usage by software subsystems, such as applications and communication networks with each usage group having a dedicated functional layer. Clark and Tennenhouse [22] present architectural considerations relevant to the ISO model. The most important and relevant observations made by Clark and Tennenhouse [22] are:

- data manipulation costs more than transfer control operations;
- application data units are the natural pipelining units and they correspond to what applications want, not to the network technology of the day, which can and will change in the near future; and
- Integrated Layer Processing (ILP) allows applications to process their data incrementally and permits efficient implementation of data manipulations on RISC processors.

Though it is evident that the work performed by Clark and Tennenhouse [22] is at a rather higher level from the target level of abstraction in this thesis, it still provides interesting transferable insights. In particular, the performance implications from data processing functions, as opposed to only data transfer and control functions. It also presents interesting engineering approaches to integrating more than one functional aspect in the same implementation layer. While this might be interesting

from a performance point of view, it has to be performed with the utmost care, because it has implications on the principle of functional independence and hierarchical decomposition.

Finally, and probably most relevantly, the study by Furber and Spars [23, Chapter 2] is reviewed. Furber and Spars first describe a number of asynchronous communication protocols. They then describe the basic functional aspects of those protocols and a possible "solution space" of the different configurations of those protocols. "This solution space can be expressed as the Cartesian product of a number of options including:

$$\{\text{2-phase}, \text{4-phase}\} \times \{\text{bundled-data}, \text{dual-rail}, \text{1-of-n}, \ldots\} \times \{\text{push}, \text{pull}\}"$$

This study gives a good, but rather limited theoretical background on classifying communication protocols in terms of individual functional aspects and configurations. The total possible communication configurations as a product of the individual functional aspects of those protocols is also discussed. It does not, however, extend the scope of the study beyond the few legacy asynchronous communication protocols. Moreover, it does not address more sophisticated protocol requirements, where a potential overlap of the basic requirements might happen in which case the separation of the functional requirements into functionally independent configuration is not straightforward. Such issues will be discussed in this chapter. In addition, a number of recent communication protocols that are still being used in state-of-the-art hardware designs are analysed and the overall "solution space" is discussed.

2.3 Review of Communication Protocols Features

As part of the development of this framework, a number of communication protocols have been reviewed. This was performed by reviewing the relevant standards [4–9, 24]. In addition, implementation details were also found in application notes such as [25–27]. The review was aimed at establishing high-level functional and performance requirements of those protocols. A brief discussion of a selected set of those protocols follows.

- Universal Asynchronous Receiver/Transmitter (UART) [4] is a protocol that any configurable interface is expected to support. It is an asynchronous protocol and there is no external clock exchanged between devices. The implementing interface is expected to have its own time management, including an internal

clock which is used to sample the data. The protocol is serial in nature and ideally the interface must be able to collect a number of the incoming data bits and present them to the host as one unit possibly dropping control information, such as start-, stop- and parity-bit.

- Serial Peripheral Interface (SPI) [5] is a de facto standard protocol that is used between Application Specific Integrated Circuits (ASIC) and controller systems. Application notes such as [5] are usually a good source of information in the absence of an official standard. It is a four wire synchronous protocol with a clock signal driven by the master device and received by all slave devices attached to the bus. Two serial data signals are also exchanged. This allows devices to operate in full-duplex mode where a device can send and receive data at the same time. The 4^{th} exchanged signal is a chip select driven by the master. Since SPI is a de facto standard, there are no fixed data-rates that devices are expected to support. The various implementations reviewed supported data rates between 1 Mbps and 100 Mbps for a single channel SPI interface (i.e. 1 data signal in each direction).

- Media Independent Interface (MII) [7] is an interface used to communicate with Ethernet physical bus transceiver chips. Clocking rates vary between 10 Mbps in earlier versions of the protocol to 10 Gbps in most recent versions. 100 Mbps and 1 Gbps are the most widespread clocking rates these days. The interface is divided into three sub-interfaces: transmitter, receiver and management. The transmitter and receiver interfaces are similar in the fact that they both have four bit data buses along with three control signals: *Clock*, *Data Valid/Enable* and *Error*. The management interface is a two wire serial interface: clock and data. The clocking signals can be either driven by the controller or by the transceiver chip. The controller is expected to be able to serialise and de-serialise multiple data items, while dropping control information, such as the data preamble. Special care is needed to allow for serialisation and de-serialisation of data items with variable widths, especially at the end of data packets. Framing signals (*Data Valid/Enable*) are exchanged between devices which specify when the data bus carries valid data. In a receiver, the interface is expected to use such external signals to enable/disable sampling the data bus and the transmitter interface is expected to auto generate these signals.

- Controller Area Networks (CAN) [6] is a protocol standard used for implementing a network of communicating embedded controllers in the automotive

industry. The CAN bus is a two wire bus that allows for many controllers to communicate without the need of a bus arbitration or coordination device. When analysing the protocol in terms of low-level physical interface requirements, the synchronisation mechanism was identified as a challenging aspect of the specification. This is because it requires high resolution sampling and processing of the data bus by the controller. The synchronisation process divides each CAN bit into a number of time quanta. The maximum number of time quanta per bit is 25. Considering a bit-rate of 1 Mbps, then the possible sampling rate of the interface is 25 Mbps. Hartwich and Bassemir [28] discuss the synchronisation process in more detail. At such a high sample rate, a serialising buffer is expected to decouple the physical interface from the controller pipeline. The hard synchronisation requires devices on the bus to check for a specific transition on the data line and sample subsequent data values relative to that transition.

- Inter-Integrated Circuit (I^2C) [8] is a two wire protocol normally used to convey control information between a number of chips in an embedded system. Control information can range from volume control in an audio chip to reading physical interface settings such as port width, bandwidth, and protocol version in other controllers. It is a master/slave protocol: a master drives the clock signal and has main control over communications on the bus. Each device in the system can act as either a master or a slave at any particular point in time. The protocol allows for different clock speeds of the masters in the system through a clock synchronisation mechanism. It also allows for the coexistence of more than one master on the bus. There can only be one master in the system at any point in time. Masters use the bus arbitration mechanisms to claim ownership of the bus. Devices switch from master to slave when they lose the arbitration in case they are addressed by the winning master. The two wires are the clock line (SCL) and the data line (SDA). Communications along the SDA line is bidirectional. Information on the SCL flows unidirectionally in a single master set-up. However, in a multi-master configuration the SCL is also bidirectional. Both the SCL and SDA signals are pulled high by default when the bus is idle. Finally, clocking rates vary between 100 Kbps to 3.4 Mbps.

- Inter-Integrated Circuit Sound (I^2S) [9] is a serial communication protocol intended for the communication of audio data between controllers and audio devices. The controller can operate in master mode where it generates the clock,

or as a slave where it receives the clock from an external source (the audio device or an independent clock management hardware). The interface comprises of two serial data lines: digital-to-analog and analog-to-digital and two corresponding framing signals for left/right channel selection. A clock signal is also exchanged. When it is necessary to support higher data rates, it becomes clear that the controller needs additional hardware to support automatic clocking of the audio data using the externally provided clock. In such a scenario, a serialising buffer would also be necessary to decouple the relatively slow controller pipeline from the faster audio interface. In addition, there is a two wire management interface similar to the I^2C protocol discussed earlier.

- Handshake Protocols [23, Chapter 2] are a class of asynchronous communication protocols. They are generally implemented using handshake signals such as Request and Acknowledge. The handshake signals can be made to return-to-zero before each transaction, in which case the protocol is called 4-*phase*. If the return-to-zero mechanism is not enforced, the protocol is called 2-*phase*. They include:

  - Bundled-Data Protocols where two handshake signals are exchanged along with the data signals; and
  - Dual-Rail Protocols where only the Acknowledge signal is present while the request signal is encoded using the data signals.

  Other configurations of this set of protocols exist. Refer to [23] for more details.

2.4 Functional Requirements

From the review in Section 2.3, the following features are common in many protocols:

1. Raw: A most basic form of communication is observed which is not subject to any timing or data conditions. Lower levels of timing or data control are disabled. This form communication is asynchronous and happens instantaneously on the most abstract communication unit (1 bit port or wire). This definition is relative to the individual communication unit and other forms of control might be imposed at higher levels of abstraction such as ISA.

2. Clocking: A clocking mechanism to satisfy specific timing requirements of protocols is needed. It should be possible for this timing control to be based on an internally generated clock or an externally exchanged one. It is also evident that this clocking control mechanism should be able to interact directly and closely with the rest of the communication interface.

3. Time Dependent I/O: When implementing a real time function, it should be possible to perform specific communication operations at a specific time using a reference clock. The clock source might be the internal system clock, any multiplier or divider of that clock, or possibly an external clock source.

4. Data Dependent I/O: The ability of an external event to automatically trigger internal actions or responses in the interface. This action or response could be configured in advance but should subsequently finish automatically.

5. Direction: The ability to switch the function of an interface between input and output.

6. Buffering: This is useful for decoupling the physical interface from any further processing. This is particularly useful when the interface has higher clocking speeds than further processing units in which case it should be possible to enable the buffer to be controlled by external clocking mechanisms. This arrangement can only work when information flows into the system in bursts. The size of those bursts would control the amount of buffering that is needed while the average sustained data rate would control the ultimate speed of the overall system and subsequently the speed of any further processing performed on the data.

7. Serialisation: It is usually useful to have special hardware which enables the support of data items of various sizes. This is particularly the case when sections of the hardware pipeline are designed to manage data of sizes that are larger than the actual physical interface. This is evident in many of the reviewed protocols. In such cases, higher software abstractions would have the ability to process, store and deliver data items that are typically larger than the size of the physical interface. The serialisation hardware is then able to (de-)serialise the data from/to the physical interface. There are also cases where the physical hardware interface is larger than higher software abstractions and in such cases the serialisation process would also provide a useful conversion mechanism.

8. Hierarchy: In many protocols, the system is organised in a way to allow for devices to have higher control over operations and transactions performed on the bus. This becomes more important in the context of shared buses where more than two devices can be actively participating in a transaction. In some of the protocols above, a device with higher control priority is called Master. This device has higher priority of control over the communication interface compared to a Slave device. It is worth considering the case where devices have symmetric control over the communication interface. In such case, any device sharing the bus can start a communication transfer at any point in time. A more complex network of devices, which includes multiple levels of hierarchy could be considered.

9. Handshake: Some protocols have signals to indicate the validity of the exchanged data. These signals can either be accompanied with a clock to establish the timing of the exchanged signal and possibly enable multiple data transfers for each request. Other synchronisation mechanisms include the presence of an Acknowledge signal in the opposite direction.

10. Pull-up/down: This is where protocols expect the bus to have a default logical state when not driven by any attached device.

11. Invert: This is a low-level hardware requirement where a protocol expects its signals to be active for logical low values instead of logical high. For that reason, it would be useful if the interface would automatically address this requirement and provide a configurable option to enable this functionality.

12. Width: In some protocols (e.g. the MII protocol) a number of physical signals are grouped together to act as one unit. This is normally the case with the data bus. Interesting scenarios occur when the size of a single unit of transfer from another level of abstraction (Instruction Set, for example) is larger or smaller than the size of the underlying hardware unit. Another interesting situation is when all this is combined with some sort of buffering mechanism.

13. Return-to-Zero (RZ): Clocking and handshake mechanisms can use only the rising edge of a control signal to trigger data transfers and hence the control signal must Return-to-Zero after each transfer. An alternative mechanism would allow for both the rising and the falling edges of the control signal to be used for triggering data transfers. In such case the term becomes Non-Return-to-Zero (NRZ).

May [Chapter 15, 29] demonstrates an ad-hoc implementation of a configurable communication system addressing some of the requirements outlined above. There is no evidence of the use of formal methods by May in the design and verification of the system discussed in [29]. The framework demonstrated in this thesis provides a good proof-of-concept of a formal framework that could be used in the future for the design of complex configurable communication systems similar to [29].

2.5 Semantic Analysis of Requirements

The aim of this section is to analyse the requirements in terms of their orthogonality and group dependent design requirements together. This is achieved through the analysis of the dependencies between the various requirements. Subsequently, an orthogonal set of requirements is constructed. This orthogonal set will form the basis for configurable communication units, which could be used to construct a configurable communication system. Grouping interleaved and dependent requirements in a single functional unit is key to the success of orthogonal hardware design.

The proposed framework uses an ISA interface for configuring the associated functional units and establishing the communication stream. In such an ISA-based communication system and when additional timing and performance requirements are not taken into account, most of the requirements discussed in Section 2.4 can be addressed at a higher level of abstraction using ISA specifications and without any additional hardware units. However, the need for such additional hardware support becomes apparent when specific performance requirements are imposed.

2.5.1 Naming Conventions

Requirement 1 in Section 2.4 specifies the need for an asynchronous and unconditional form of communication from/to the physical interface. This has originally been thought of as a stand-alone functional requirement that would be modelled using a separate functional unit. However, after practical considerations, it turns out to be a naming convention for a specific *default* configuration of all the functional units in the system. For example, the functional unit handling timed data I/O would be inactive by default, hence all communications would be instantaneous and could be called *raw*.

2.5.2 Direction

The direction of the data transfers (Input or Output) in a communication interface is not seen as an independent requirement that could be modelled in an independent hardware module, but instead an aspect or configuration option that some modules use. Some requirements depend on this aspect such as Data Dependent I/O, Serialisation and Inversion. Other requirements are independent of this aspect such as clocking, timing and handshake signals.

2.5.3 Synchronous, Asynchronous and Hierarchical Communication

A few functional requirements that were initially considered orthogonal are the clocking mechanism used in synchronous communication protocols, the handshake signals used in asynchronous communication protocols and hierarchy. However, when those requirements were considered for orthogonality in the context of real-time systems, the situation became tricky.

2.5.3.1 Synchronisation and Timing Control

Real-time systems are inherently synchronous with system events, actions and responses all synchronising to a global clock which runs freely and infinitely. It is usually useful for an internal system clock or any division of that clock to be made available externally, as a means of synchronisation with another system. Sub-systems usually use different clocks in their corresponding clock-domains. A signal that crosses from one clock domain to another clock domain has to pass through a synchronisation boundary. An abstract look at the relative synchronicity of signals would interpret any signal on one side of a synchronisation boundary as being asynchronous to the other side and not bound by its clock. This of course includes the clock itself, in the case of synchronous communication protocols considered earlier. The system clock is typically regular with fixed timing specifications and runs infinitely. Real-time systems depend on this property in order to function in a timely manner. Asynchronous systems, on the other hand, are not bound by any clock signal and exchange data using some arrangement of control signals (request and acknowledge). The advantage of asynchronous communication protocols is their potential ability to outperform synchronous systems, because they can in theory run as fast as the physical hardware and electrical signals allow them to.

In some synchronous systems, the notion of clock-stretching is used to allow for extending the duration of clock cycles. This is useful for allowing either the sender or the receiver extra time for processing the communicated data. This technique is clearly used in the I^2C protocol mentioned earlier. When clock-stretching is employed by the receiver, the technique can be conceptually interpreted as some form of positive acknowledge to the reception of the data item. Asynchronous handshake protocols use a standalone acknowledge signal for this purpose. On the other hand, if clock stretching is employed by the sender to delay the next data transmission, it makes the clock signal equivalent to the physical request signal used in asynchronous handshake protocols. It is also possible for both the sender and receiver to use clock stretching, where the sender delays one edge of the clock (let us say the positive edge) and the receiver delays the negative edge. This way, a full handshake protocol is achieved. Hence, a stretched clock signal could be interpreted as either request, acknowledge or potentially both. In any case, the stretched clock signal becomes no longer regular and infinite.

Another way to look at the relation between synchronous and asynchronous communications is to assume an asynchronous protocol is configured to use only one handshake signal (say request) to specify the availability of new data items on a bus. If data transfers are regular and unacknowledged then this signal can simply be interpreted as the clock signal in any synchronous communication protocol (e.g. the MII protocol mentioned earlier). The fastest possible frequency of this request signal in a real-time system is equivalent to the internal clock of the system, assuming a new data item is available with each clock cycle. The same arguments can be used on the existence of an acknowledge signal which could be regular and equivalent to a clock going in the opposite direction with maximum frequency equivalent to the clock speed of the sub-system generating this acknowledge signal.

Higher-Levels of Clocking and Synchronisation could be envisaged through the presence of other data enveloping signals, as is the case with data valid signals for the MII protocol or the chip select signals in many other protocols such as SPI. Those signals could be interpreted as being higher-level synchronisation/handshake signals similar to the request or clock signals discussed earlier. Those signals are typically valid for a whole transaction (say a packet of Ethernet data) rather than only one data transfer which is the case in *lower level* clock and request signals.

Another interesting scenario is encountered in the I^2S communication protocol, where the channel-select signal can also be interpreted as being a higher-level synchronisation/handshake signal. This signal is particularly interesting because it splits

the flowing data bits into the left and right channel by assigning each channel a different logical value (either 0 or 1). This is similar to a higher-order Double Data Rate (DDR) technique used in the implementation of Random Access Memory (RAM).

However, higher levels of hierarchy are discussed here purely for philosophical reasons. This is because the signals involved typically have low bandwidth and additional hardware considerations for such signals usually have diminishing returns.

2.5.3.2 Transaction Initiation Hierarchy

Though it might not always be the case, control of the communication interface hierarchy is usually accomplished by controlling the synchronicity signals (i.e. the clock or the handshake signals).

The term Master/Slave is used in a number of protocols to indicate hierarchy. However, the terminology of Initiator, Responder and Symmetric shall be used instead for the following reasons:

- Master/Slave can indicate a number of other control mechanisms in addition to using the synchronisation signals to establish hierarchy;
- the lack of terminology to indicate Symmetric access to the bus; and
- the term Master/Slave is deemed unethical for indicating hierarchy.

As the names imply, Initiator is the *active* device that is capable of initiating a communication transaction and Responder is the *passive* device that is unable to initiate a transaction, but responds to transactions when one is initiated. Finally, a Symmetric device can both initiate transactions and respond to transactions if they are initiated.

Because transaction initiation is tightly coupled with synchronicity signals, this functionality is expected to either be a specific configuration of any timing unit, or possibly a subunit within a synchronicity functional unit.

2.5.4 Buffering, Serialisation and Shifting

Buffering is meant to handle situations where the two parties involved in the communication have different throughput capabilities. By providing an intermediate buffer to hold communicated data until the destination is able to process this data, the difference in capabilities of both sides can be managed and its effect on slowing down the interface is minimised.

Serialisation transforms the data from different sizes of interfaces. This is typically used to transform data from the internal system data units (typically 8, 16, 32 or 64 data bits) to the physical interface size, which is typically of smaller size (1 to 4 in most of the reviewed protocols above).

Though buffering and serialisation seem at first to be two orthogonal requirements, closer examination reveals the semantic dependency of serialisation on the existence of a buffer. For this reason it would not be possible to reason about any form of serialisation mechanism without the existence of some sort of buffer to hold the intermediate data value during the serialisation process. Hence, serialisation and buffering requirements are best modelled as a single entity.

2.5.5 Compositional Semantics

In this section the meaning of various possible compositions of the requirements is discussed. An exhaustive analysis of each possible arrangement of all the possible requirements is impossible in the scope of this thesis, as will be discussed later.

Let us consider the theoretical solution space resulting from all possible compositions of all functional units into a single configurable system: is this a permutations or combinations problem?

At first glance, one might expect that the order in which the functional requirements are composed together is irrelevant and hence assuming a combinations problem. However, some of the requirements operate on larger data structures such as the serialisation, while others only operate at lower levels next to the physical interface. Some requirements have no effect on the data values at all, such as *Time Dependent I/O*. State-of-the-art communication fabrics are typically positioned between an ISA controller and the actual hardware pins. This means that requirements operating on wider data structures would typically be positioned near the ISA controller, while the ones that operate on data values the width of the physical interface would operate nearer to that interface. In light of this, one can deduce that the order in which the functional units are arranged *is* important, resulting in a permutations problem.

However, because there are at least 13 requirements (some of which might be merged into a single functional unit as discussed in Sections 2.5.4 and 2.5.3), there are 4.8×10^8 possible permutations. As mentioned earlier, considering all the possible permutations of those functions is not practical. A structural and functional analysis of those requirements guided with formal analysis would help decide the exact structure of the communication system.

Once a suitable permutation of the functional requirements has been selected, the task of composing those requirements into a single comprehensible system becomes a necessity. Most requirements are independent of real-time or data values such as pull-up and invert functions. Hence, one could use both invert and pull-up in a single I/O operation, because neither functions conflict with the other function. However, this is not the case when dealing with Data Dependent I/O composed with Time Dependent I/O: is it possible to perform an I/O operation which depends on both the value of the data, as well as the real-time that the operation takes place? One approach is to provide an additional composability function to deal with this dilemma. This composability function would allow for the logical composition of two functional requirements or units. Using this composability function one could achieve Data Dependent I/O *AND* Time Dependent I/O on the same operation: the input operation will not take place until a specific time has passed and also the value of the data matches the value specified by the operation. It should be possible to use any of *AND*, *OR* or *XOR* as the composability function.

As more configurable options are provided, the chance of equivalent configurations increases. For example, a timed *AND* buffered operation could be used to achieve a variable size input (i.e. configurable size buffer) if the timer clock source is the same as the buffering clock. Another interesting composability scenario arises from the I^2C multi-master configuration, where the system could be configured to do both Time Dependent I/O *OR* Data Dependent I/O at the same time to watch for loss of arbitration.

A suitable encoding interface that supports composability might be useful for extra performance advantage. This interface would possibly merge multiple configuration operations into one: the configuration of each functional unit as well as configuring the composability function of those units.

Finally, a mechanism for ensuring all blocks involved in a compositional transaction have finished would be necessary to ensure all scheduled operations have finished and the configuration of subsequent operations may commence.

2.6 Orthogonal Functional Blocks

This section summarises the requirements in light of the above discussion and establishes an orthogonal set of functional units. The units themselves should be constructed in a hierarchical manner whereby multiple dependent functional requirements

are used to construct smaller hardware units. These functionally orthogonal units are then used to construct the top-level functional unit or system.

2.6.1 Data Dependent Control (DDC)

This control unit handles requests that depend on the value of the data in the I/O operation. The unit is positioned as close to the physical interface as possible to minimise the amount of traffic that would otherwise be involved when polling the interface for data. Its main concern is to trigger an event, once a specific value has been observed on the physical interface.

2.6.2 Synchronisation and Timing Control (STC)

This unit groups all the requirements that are concerned with synchronisation between two systems. This includes Timed I/O, Clocked I/O and Asynchronous I/O. It also contains configuration options for specifying transaction initiation hierarchy and other signal-specific options such as Return-to-Zero. This unit is attached to and has direct control of the physical interface. It also has a direct control interface from the ISA controller.

As discussed in Section 2.5, clocking, timing and hierarchical control are all interleaved requirements and hence are all addressed by the STC unit. A list of configurable options and signals follows.

- **Trigger Source** is a configurable option, which specifies the source of the signal used to increment the internal time counter, which subsequently initiates communication events on other physical interface signals. Typically, the source is regular and triggers infinitely which could be thought of as a reference clock (useful for implementing real-time functions). On the other hand, the trigger could come from an external source to the system, in which case it would be most relevant to the communication interface operations and might not be regular, depending on the I/O activity occurring on the interface. An activity could be controlled by external or internal data transfers, or both, as in the case of a handshake protocol, where the request signal initiating the next data transfer could not commence until an acknowledgement of the previous transfer has occurred and the next data item is ready for transfer.

- **Clock/Request** or **Higher-Level Framing/Acknowledge** configurable options could be used to enable the system to perform clocked or asynchronous communications.

- **Attached Data Signals** configures a set of data signals to be attached to this STC unit.

- **Return-to-Zero** configuration specifies whether the special clocking signals activate data transfer on one edge only (RZ) or on both edges (NRZ). The NRZ configuration could be useful in some protocols such as the DDR clocking signal and the I^2S framing signal.

- **Initiation** is a configurable option in the STC unit, which specifies whether the unit has Initiator, Responder or Symmetric access to the physical interface.

Finally, the direction of the control signals should be orthogonal to the direction of the attached data signals. For example, a clock could be input or output. Such a clock could be used to input or output data similar to the *Pull/Push* configuration option discussed in [23].

2.6.3 Shifting and Buffering Control (SBC)

As discussed earlier is Section 2.5.4, serialisation and buffering are best modelled in a single functional unit. This unit would have a control interface with the STC unit to enable the timely clocking of data in/out of the buffer. The buffer is concerned with holding data items. It would be useful to have other items buffered along with the data such as their input/output times and other control information.

Possible configurations to be implemented in the SBC unit follow.

- Direction: in which direction does the data flow? To or from the ISA interface.

- Size: what is the size of the buffer? A useful mechanism when it is necessary to read data values from the buffer that are smaller than the buffer full capacity. This affects the size of the ISA read/write interface.

- Attachment: which physical entity or entities is the unit attached to? Could be a single bit or multiple bits.

It should be possible for this unit to respond to an *End Transmission* command where the built-in buffer is either read by the ISA interface when in input mode, or flushed when the unit is in output mode.

2.7 Summary

This chapter reviewed a representative set of communication protocols to establish the common requirements and trends in state-of-the-art communication systems. In conjunction with previous work for classifying communication protocols [23] an analysis of the identified requirements was performed to establish their orthogonality. This provided the blueprint for the functional units to be used for building up the configurable communication system. This generic configurable system was later used for the modelling and verification of full communication protocols. This will form the benchmark for the modelling, analysis and verification framework discussed throughout the following chapters.

Many essential functional requirements would be captured by the functional units discussed earlier. The remaining requirements are either:

- **functionally distributed** over many units such as the *direction* of communication and the width of the physical channel;
- merely a **naming convention** and do not require additional hardware support such as the *Raw* I/O configuration, which is the default configuration for the functional units; or
- **trivial** such as invert and pull-up/down and can simply be implemented using an orthogonal functional unit in the communication pipeline.

The functionality of the overall hardware system grows exponentially with the addition of more functional blocks, while the design and verification efforts grow linearly. For example, consider if only two functional units were implemented in our system (STC and SBC), and each unit had only two configurations (for STC: Timed and Untimed, and for SBC: Buffered and Unbuffered). The overall functionality would be the Cartesian product of the functionality of the two units: (*Timed, Untimed*)$\times$(*Buffered, Unbuffered*), i.e. four functions in total, while the design and verification efforts are focused on two independent hardware units. The benefits become greater with the addition of more functional units to the system.

Finally, one must point out that the purpose of this chapter and all subsequent chapters is for a proof-of-concept of the requirements, techniques and methodologies that are essential for a formal modelling and verification framework. For this reason, the demonstrated formal models in subsequent chapters are not meant to be exhaustive.

CHAPTER 3

Data and Control Multiplexing Approach to Configurability

3.1 Motivation and Chapter Structure

Chapter 2 discussed a natural progression to the configurability evolution: the detailed analysis of selected communication protocols and the subsequent classification of their functional and timing aspects into functionally independent configurable hardware modules. It laid the theoretical foundations of the overall Design and Verification approach discussed in this thesis. This chapter presents the first attempt on addressing Objective 3: empirical analysis through formal specification and modelling.

First, Section 3.2 discusses the various implementation approaches to hardware configurability and the progression toward the systematic approach described in this thesis and first modelled in this chapter. Section 3.3 presents a brief introduction to the modelling language of choice: CSP along with the reasoning behind this choice.

Section 3.4 presents details of the multiplexing scheme, which attempts to connect the configurable hardware units using a general purpose data and control pipeline. Figure 3.1 highlights the intended function of the models presented in Section 3.4 in the design of the configurable communication system.

Then Section 3.5 partially addresses Objective 4 through the development of the automatic State Machine Visualiser for use with the FDR model-checker. Finally, Section 3.6 presents the results and conclusions of this chapter.

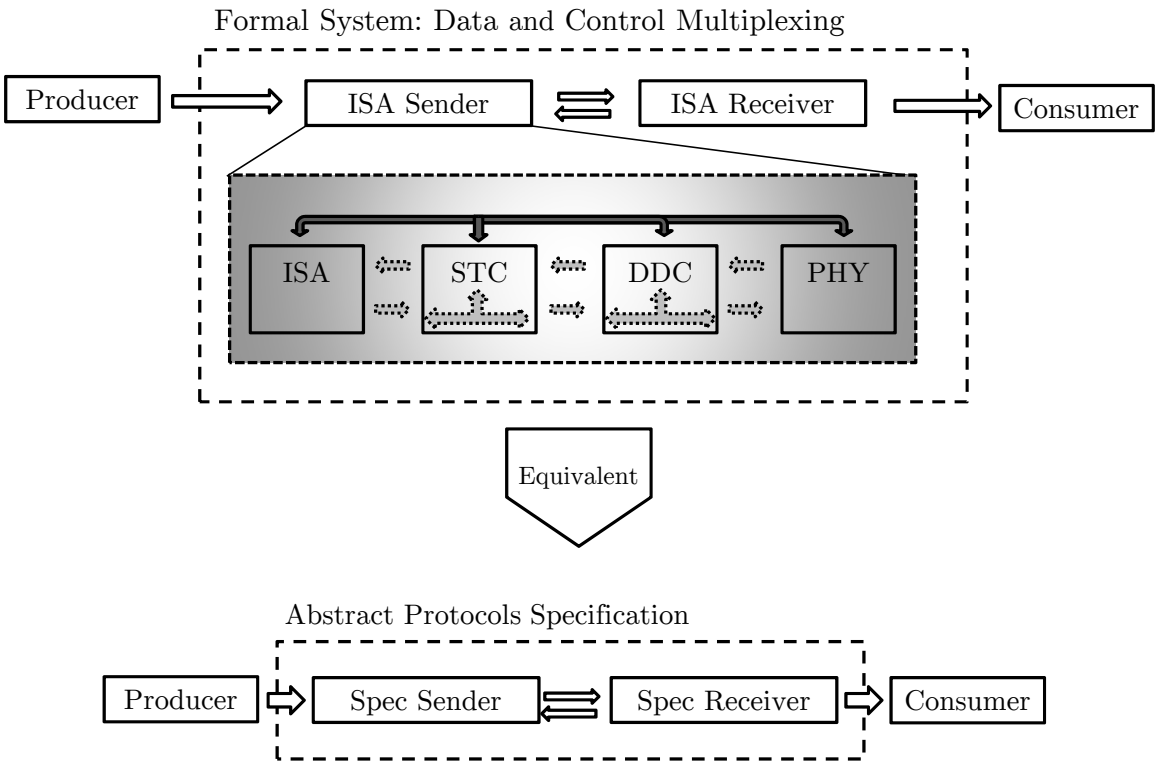


Figure 3.1 Outline of Data and Control Multiplexing Approach to Configurability

3.2 Background

Looking at the modelling and verification framework from a hardware implementation perspective, the notion of hardware configurability is quickly encountered for providing hardware that implements and supports multiple protocols. A review of different approaches to hardware configurability provides good background to the implementation directions on which the suggested modelling system can be based. The aim is to show the need for a generic and configurable system for the design and verification of such protocols.

Hardware protocols were traditionally supported using dedicated hardware interfaces with basic configurable options to accommodate the various options within the same protocol [30]. A fundamental functional change to the interface behaviour was not possible and supporting multiple interfaces would require considerable design and verification efforts.

The development of the Field-Programmable Gate Array (FPGA) [31] technology, on the other hand, introduced a true sense of configurability to hardware design. Nevertheless, this approach suffers from high power consumption and relatively low clocking speeds [32]. Based on the FPGA technology Horta et al. [33] suggested

a configurable architecture for a network application. This was a good example for providing a configurable network protocol infrastructure using FPGA technology. Finally, May et al. [34] suggested a different approach to configurability, which involves providing the user with a semi-configurable communication interface that has a set of configurability options. This approach proved useful for expressing communication protocols using ISA implementations.

The most obvious problem with the last two approaches is the grand scale of the verification problem. Both are not easily verifiable using traditional approaches. The fully configurable FPGA approach poses verification questions on many levels: the verification of the underlying FPGA fabric, the verification of the Hardware Description Language (HDL) specifications to be synthesised on the FPGA and finally the verification of any higher-level software-based configurability provided by the synthesised hardware. The semi-configurable approach, on the other hand, reduces the number of verification levels to possibly two: the verification of the actual communication hardware fabric and the verification of the software sequences used for configuring the hardware to implement a specific communication protocol. Both approaches suffer from the state-space explosion problem, resulting from the large number of possible configurations.

3.3 Communicating Sequential Processes

The Communicating Sequential Processes (CSP) formalism first described by Hoare in [35] was ideal for use in this empirical analysis. This is due to the availability of a machine-readable dialect of CSP (CSP_M) developed by Scattergood [36] and also a well-established Failures-Divergences Refinement (FDR) model-checker. FDR was initially the brainchild of INMOS Ltd. Earlier versions were developed through a collaboration between Formal Syst. (Europe) Ltd. [18] and INMOS.

CSP is also well suited because each functional block can be composed as an independent sequential process and the set of blocks on the form CSP processes synchronize over communication events. The fact that these blocks are designed and verified individually at block level gives rise to the DfV approach.

In this section, a brief introduction to CSP modelling constructs is presented. CSP has been devised as a modelling and verification language for computer systems involving concurrency. There are numerous texts in literature that provide more exhaustive theoretical and practical details about CSP. For example [35, 37–39] to

mention a few. The aim in this section is to provide a brief and comforting background to CSP for the unfamiliar reader.

3.3.1 Primitives

In CSP, a *Process* is defined as an independently existing object (or entity) which communicates, performs or engages in actions or events. *Event*s are visible atomic parts of the process behaviour, which offer uniform treatment of different physical events and also build up various styles of inter-process communication. The set of events which are considered relevant for a particular description of an object is called *Alphabet*. An object or process can only execute or engage in events that are part of its alphabet and it is impossible for processes to execute any event outside their alphabets. The simplest behaviour of a process is doing nothing. Such behaviour is called the STOP process. When modelling systems in CSP, the convention is that events are usually written as words in lower-case letters, whereas words in upper-case letters denote process names such as the STOP process mentioned earlier.

Processes are constructed of sequences of events and other processes by means of *prefixing*. For example, let Q be a process name and let x, y and z be event names, then the process:

$$\mathrm{P} = x \rightarrow y \rightarrow z \rightarrow \mathrm{Q} \tag{3.1}$$

is a new process, which performs the events x then y then z then exhibits the behaviour of the process Q. It is not possible for two or more events to occur simultaneously.

It is possible to define processes recursively in terms of themselves. This could be useful when there is a cycle of events that a process executes many times. For example, one could define a CLOCK process, which executes clock events (*tock*) indefinitely:

$$\mathrm{CLOCK} = \mathit{tock} \rightarrow \mathrm{CLOCK} \tag{3.2}$$

It is also possible for recursion to exist between two or more processes, such an arrangement is called mutual recursion. For example:

$$\mathrm{Q} = a \rightarrow \mathrm{P} \qquad\qquad \mathrm{P} = b \rightarrow \mathrm{Q} \tag{3.3}$$

are two mutually recursive processes.

A choice of events could be available for processes to choose from at any point in time. This could be expressed using one of two choice operators.

1. An internal choice ($\sqcap$): the process can decide internally and non-deterministically which event to execute. In this case the environment has no effect on which choice of events the process decides to execute.

2. An external choice ($\Box$): the choice of events the process can execute depends on the ability of the environment to execute it too.

The environment of a process *are* processes that interact with that process at the point of the choice.

In Eqn. 3.4, the process P is defined as an infinite sequence of external choice from any events in the set $\mathrm{A} = \{a, b, c\}$:

$$\mathrm{P} = a \rightarrow \mathrm{P} \ \Box \ b \rightarrow \mathrm{P} \ \Box \ c \rightarrow \mathrm{P} \tag{3.4}$$

The choice can be made from a set of events using *menu* choice: if A is a set of events and, for each event x in that set, $\mathrm{Q}(x)$ defines the behaviour of the process Q that is specific to the event x, then one can rewrite the process in Eqn. 3.4 as a *menu* choice as follows:

$$\mathrm{P} = \Box \ \forall x \in \mathrm{A} \Rightarrow \mathrm{Q}(x) \qquad\qquad \mathrm{Q}(x) = x \rightarrow P \tag{3.5}$$

It is useful sometimes to define processes at many abstraction levels, where details might be relevant in some contexts, but not in others. Abstracting away some events the process can perform that are not of interest to the observer could be performed using hiding or concealment. A process P, which is defined as the exact behaviour of process Q, except that the set of events X is hidden could be written as:

$$\mathrm{P} = \mathrm{Q} \setminus \mathrm{X} \tag{3.6}$$

Another feature of CSP is the notion of communication channels. For example, a process could be involved in a simple *input* event with another process, which could also be interpreted as a simple channel of communication with that process. In fact, all events in $\mathrm{CSP_M}$ are defined as channels. However, the notion of communication channels becomes more interesting when the channels could have additional dimension(s), in which they could communicate (possibly complex) values from a set of defined events. Those are called complex channels. For example, a complex channel

called *input*, which is able to communicate a single digit integer can be used in a process definition as follows:

$$\mathrm{S} = \{0\,..\,9\} \tag{3.7}$$

$$\mathrm{P} = \Box\, \forall\, x \in \mathrm{S} \Rightarrow \mathit{input}?x \rightarrow \mathit{compute} \rightarrow \mathit{output}!x \rightarrow \mathrm{P} \tag{3.8}$$

where P receives a data item ($\mathit{input}!x$), performs some internal processing and communications actions based on the received data (*compute*) and subsequently transmits the data item unchanged ($\mathit{output}!x$).

Let us think of the process P above to represent a simple communication system. One could abstract away (or hide) all internal system events, leaving only the reception and transmission events $\mathrm{P}' = \mathrm{P} \setminus \mathit{compute}$. This is useful for verifying that the overall system behaviour is equivalent to a buffer (with some additional details, such as the depth or width).

3.3.2 Concurrency

CSP is ultimately a process algebra concerned with establishing the mathematical relationships governing the interactions between sequential processes. For this reason, there is a wealth of mechanisms and notations by which interactions between processes could be specified. A brief list of concurrency operators is introduced in this section.

Let us assume the existence of two processes P and Q that need to run concurrently under many configurations.

When the two processes to be run concurrently do not have the same alphabet, they can be joined using the $_{\mathrm{X}}\|_{\mathrm{Y}}$ operator:

$$\mathrm{P}\ {}_{\mathrm{X}}\|_{\mathrm{Y}}\ \mathrm{Q} \tag{3.9}$$

where X is the alphabet set of process P and Y is the alphabet set of Q. In that case, the two processes would synchronise over events that exist in both X and Y sets.

On the other hand, when the design imposes the need for synchronisation over a single set of events (for example, the union of the alphabets of both processes), then the $\underset{\mathrm{Z}}{\|}$ operator can be used, in which case the two processes only synchronise over events specified in the Z set:

$$\mathrm{P} \underset{\mathrm{Z}}{\|} \mathrm{Q} \tag{3.10}$$

Another operator which is useful in the context of communication systems is the pipe (or chaining) operator ($\gg$). It implies running the two processes that are used to construct the pipe concurrently. It means that the two processes are connected together and synchronise over events in the communication channel used in the connection. When used abstractly (as in P $\gg$ Q), it means that the *right* channel of P is connected to the *left* channel of Q.

The interleave operator ($|||$) is used to run processes in parallel where they operate without any direct synchronisation with each other:

$$\mathrm{P} \;|||\; \mathrm{Q} \tag{3.11}$$

The chaining operator ($\gg$) is used to *chain* two processes in parallel (P $\gg$ Q) where all events output by P on an arbitrary channel (say *out* or *left*) are simultaneously input by Q on another arbitrary channel (say *in* or *right*) and all such communications are hidden from their common environment.

One could define an *alphabetised chaining* operator where the chaining operator is used in conjunction with the alphabetised parallel operator ($\underset{Z}{\|}$) to specify the arbitrary *input* and *output* channels used by the chaining operator as follows:

$$AlphaChain = \mathrm{P} \underset{left \gg right}{\|} \mathrm{Q} \tag{3.12}$$

As discussed earlier, the introduction to CSP discussed in this section is intended to provide a brief and comforting background. For full CSP background the reader is referred to [35].

Finally, CSP is inherently a mathematical description language for process algebra and hence it is usually convenient to use standard mathematical notation when describing a CSP process. In addition, Mazur [40] provides a summary of all the CSP symbols that can be used in modelling CSP processes. The notation used for describing CSP models in this chapter (and in the whole of this thesis) uses a combination of the Symbol Macros for CSP described in [40] as well as standard algebraic notation to achieve best readability.

3.3.3 Timed CSP

Real-time modelling is an important aspect of communication protocols design. It enables the verification of various performance aspects of the design at an early stage in the design cycle, in addition to verifying other functional aspects under the same framework. The topic of real-time modelling and verification in general is a broad

topic and a background review of all related material is beyond the scope of this thesis. A brief review of work related to real-time modelling in CSP is presented here. The focus is on the practical aspects using the FDR model-checker.

Roscoe and Reed [41] are believed to have developed the first approach for modelling timed systems in CSP. Subsequently, Schneider in his book *Concurrent and Real-time Systems* [39] discussed *Timed CSP* in detail.

Ouaknine [19] discovered close connections between *Timed CSP* and a discrete time version called tock-CSP. In tock-CSP the passage of time is represented by an external event (usually called *tock*), which is a global event that happens regularly and infinitely.

Traditionally the *tick* event is normally associated with clock actions rather than *tock*. However, *tick* is used in CSP to denote successful termination of a process using the CSP symbol ✓ or a process named SKIP. Hence, *tock* was chosen to represent clock events.

The use of an external CSP event to represent clock events changes the semantics of some CSP constructs and models including the *failures/divergences* and *traces* model. See [19] for further details. Furthermore, in the book titled *Understanding Concurrent Systems*, Roscoe [Chapters 14 and 15, 38] focused on the practical aspects of tock-CSP using case studies and automated model checking using FDR. The case studies included the modelling and verification of an algorithm called the Bully Algorithm which is used in distributed systems for dynamically selecting a coordinator.

In an earlier case study, Seidel [42] modelled and verified the Peripheral Interconnect Bus developed by Open Microprocessor Initiative (OMI) [24]. Seidel showed, through the use of automated model checking, that two different implementations of the protocol are compatible with the higher-level specification. This gave rise to the question of whether both of these implementations are regarded as acceptable, according to the standard, because they are mutually incompatible.

The author has made various attempts to verify the results of that study using a number of FDR releases and different refinement models. Those attempts were all unsuccessful. The author has now been made aware that Seidel has used an experimental implementation of the Tau Priority Model which was under development by Formal Syst. (Europe) Ltd. [18] at the time.

This chapter is dedicated to the *modelling* of the configurable communication system, hence the brief discussion of tock-CSP here is intended as a background to the modelling aspects of tock-CSP.

All aspects and constructs of CSP discussed in this section are intended as an introduction to the modelling aspects of CSP, including concurrent aspects in Section 3.3.2 and real-time aspects in Section 3.3.3. The refinement models used in the analysis of different properties of CSP processes are an important part of CSP. Those refinement models are introduced later in Chapter 7, when Formal Verification using Refinement Model-Checking is discussed. In particular, Section 7.2.1 presents a brief review of those models. The use of those models for the verification of real-time properties of processes is discussed in more detail in Section 7.2.2.

3.4 Data and Control Multiplexing

An early approach for providing configurability was considered, which was based on a shared pipeline of multiplexed data and control information. This was considered as a way to avoid many high-level race conditions that could result from separating the control and data pipelines. A block diagram for this approach is presented in Figure 3.2.

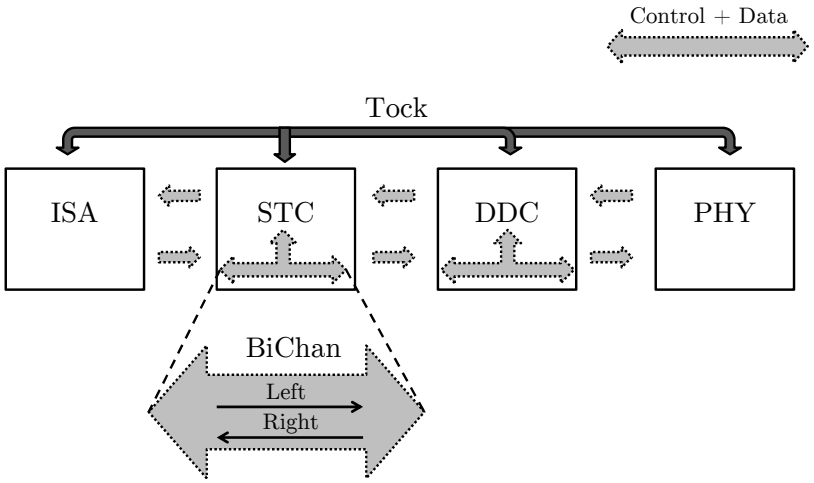


Figure 3.2 Multiplexing Data and Control System Block Diagram

The idea behind the pipeline of functional units in Figure 3.2 is to unify the interfaces for all the functional units (now CSP processes), whereby each one would have two communication interfaces, one for communicating with each adjacent functional unit in a linear formation.

The approach relies on a bidirectional, non-blocking communication channel as a building block for the pipeline. Eqn. 3.13 shows the CSP description of this channel. It uses two blocking unidirectional CSP channels. All functional blocks are chained

together using the CSP chaining operator [35]. The order in which those functional blocks are organised is fixed, as discussed in Section 2.5.5.

The *BiChan* allows for associated processes to select and respond to communicated tokens of interest. Unwanted tokens pass through the pipeline unchanged.

This bidirectional, non-blocking channel is defined in Eqn. 3.13.

$$
\begin{aligned}
&BiChan(left,\, right,\, lT,\, rT,\, Proc,\, \ldots) = && (3.13)\\
&\qquad (\Box\, \forall\, x \in rT \Rightarrow right.x?rD \rightarrow && (3.14)\\
&\qquad\quad BiCR(left,\, right,\, lT,\, rT,\, x,\, rD,\, Proc,\, \ldots)) && (3.15)\\
&\qquad \Box\, (\Box\, \forall\, y \in lT \Rightarrow left.y?lD \rightarrow && (3.16)\\
&\qquad\quad BiCL(left,\, right,\, lT,\, rT,\, y,\, lD,\, Proc,\, \ldots)) && (3.17)
\end{aligned}
$$

$$
\begin{aligned}
&BiCL(left,\, right,\, lT,\, rT,\, y,\, lD,\, Proc,\, \ldots) = && (3.18)\\
&\qquad right.y!lD \rightarrow tock \rightarrow Proc && (3.19)\\
&\qquad \Box\, tock \rightarrow BiCL(left,\, right,\, lT,\, rT,\, y,\, lD,\, Proc,\, \ldots) && (3.20)\\
&\qquad \Box\, (\Box\, \forall\, x \in rT \Rightarrow right.x?rD \rightarrow && (3.21)\\
&\qquad\quad BiCRL(left,\, right,\, lT,\, rT,\, x,\, rD,\, y,\, lD,\, Proc,\, \ldots)) && (3.22)
\end{aligned}
$$

$$
\begin{aligned}
&BiCR(left,\, right,\, lT,\, rT,\, x,\, rD,\, Proc,\, \ldots) = && (3.23)\\
&\qquad left.x!rD \rightarrow tock \rightarrow Proc && (3.24)\\
&\qquad \Box\, tock \rightarrow BiCR(lT,\, rT,\, x,\, rD,\, Proc,\, \ldots) && (3.25)\\
&\qquad \Box\, (\Box\, \forall\, y \in lT \Rightarrow left.y?lD \rightarrow && (3.26)\\
&\qquad\quad BiCRL(left,\, right,\, lT,\, rT,\, x,\, rD,\, y,\, lD,\, Proc,\, \ldots)) && (3.27)
\end{aligned}
$$

$$
\begin{aligned}
&BiCRL(lT, rT,\, x,\, rD,\, y,\, lD,\, Proc,\, \ldots) = && (3.28)\\
&\qquad right.y!lD \rightarrow tock \rightarrow BiCR(left,\, right,\, lT,\, rT,\, x,\, rD,\, Proc,\, \ldots) && (3.29)\\
&\quad \Box\, left.x!rD \rightarrow tock \rightarrow BiCL(left,\, right,\, lT,\, rT,\, y,\, lD,\, Proc,\, \ldots) && (3.30)\\
&\quad \Box\, tock \rightarrow BiCRL(left,\, right,\, lT,\, rT,\, x,\, rD,\, y,\, lD,\, Proc,\, \ldots) && (3.31)
\end{aligned}
$$

where:

- *left*: is the left channel;
- *right*: is the right channel;
- lT: are tokens the process can pass from left to right;
- rT: are tokens the process can pass from right to left;
- *Proc*: is the process associated with this particular channel instance; and
- $\ldots$: is used to pass any other state variables to the target process.

The *Proc* parameter is required so that control is returned to the associated process after a token has been successfully forwarded to the respective channel.

This *BiChan* process is then used in processes that make up the middle section of the pipeline. See Eqns. 3.44 and 3.55 for more details.

For demonstration purposes, a minimal but representative description of processes is presented in this section. This description is intended to demonstrate the concept of the Data and Control Multiplexing approach. The models discussed in this section are an earlier snapshot of the concrete communication system. They are the result of a large number of iterations through the complexity analysis procedure discussed in Chapter 4. For a full description of all the modelled functional units and all associated instructions, the reader is referred to Chapter 5.

Also, some of the details presented here were optimised at later stages, hence the functional units described here are not necessarily equivalent to the ones discussed in Chapter 5. For example, the physical layer functional unit (PHY process in Eqn. 3.61) was later optimised out and the state of the interface is stored in the DDC unit, as discussed in Section 5.3.2.

Eqn. 3.32 shows the implemented commands in this snapshot which are:

- C_{data}, basic data I/O operations to the PHY;
- C_{cond}, data dependent I/O command to the DDC;
- C_{tmd}, time dependent I/O command to the STC; and
- C_{stm}, reset the time counter in the STC.

Eqn. 3.33 defines the list of functional units that are available at this stage. Eqn. 3.35 defines the list of all possible data items that could be communicated in the system.

$$datatype\ \ Command = \mathrm{C}_{data} \,|\, \mathrm{C}_{cond} \,|\, \mathrm{C}_{tmd} \,|\, \mathrm{C}_{stm} \tag{3.32}$$

$$datatype\ \ Block = \mathrm{B}_{isa} \,|\, \mathrm{B}_{cond} \,|\, \mathrm{B}_{raw} \,|\, \mathrm{B}_{tmd} \tag{3.33}$$

$$DataSize = \mathrm{DS} = 3 \tag{3.34}$$

$$DataWord = \mathrm{DW} = \{0 \,..\, (2^{\mathrm{DS}} - 1)\} \tag{3.35}$$

$$Boolean = \{0 \,..\, 1\} \tag{3.36}$$

$$ddc = 0 \tag{3.37}$$

$$stc = 1 \tag{3.38}$$

$$channel \quad tock \tag{3.39}$$

$$inc(a, limit) = (a + 1) \bmod limit \tag{3.40}$$

In light of the above definitions, the definitions of the STC (Eqn. 3.44), DDC (Eqn. 3.55), PHY (Eqn. 3.61) and ISA (Eqn. 3.67) processes follow:

$$channel \quad lSTC,\, rSTC \,:\, Block.Command.DataWord \tag{3.41}$$

$$lSTCThro = \{\mathrm{B}_{cond} \,.\, \mathrm{C}_{cond}\} \tag{3.42}$$

$$rSTCThro = \{\mathrm{B}_{isa} \,.\, \mathrm{C}_{cond}, \mathrm{B}_{isa} \,.\, \mathrm{C}_{data}\} \tag{3.43}$$

$$\mathrm{STC} = \mathrm{STC}\prime(0) \tag{3.44}$$

$$\mathrm{STC}\prime(oTime) = tock \rightarrow \mathrm{STC}\prime(inc(oTime, 2^{\mathrm{DS}})) \tag{3.45}$$

$$\Box\ lSTC.\, \mathrm{B}_{tmd} \,.\, \mathrm{C}_{stm}?time \rightarrow tock \rightarrow \mathrm{STC}\prime(inc(time, 2^{\mathrm{DS}}) \tag{3.46}$$

$$\Box\ lSTC.\, \mathrm{B}_{tmd} \,.\, \mathrm{C}_{tmd}?time \rightarrow \tag{3.47}$$

$$\begin{cases} rSTC.\, \mathrm{B}_{raw} \,.\, \mathrm{C}_{data}!stc \rightarrow \mathrm{STC}\prime(oTime) & \text{if oTime} > \text{time} \\ \mathrm{STC}_{check}(oTime, time) & \text{otherwise} \end{cases} \tag{3.48}$$

$$\Box\ BiChan(lSTC,\, rSTC,\, lSTCThro,\, rSTCThro,\, \mathrm{STC}\prime,\, \ldots) \tag{3.49}$$

$$\mathrm{STC}_{check}(oTime,\, time) = tock \rightarrow \tag{3.50}$$

$$\begin{cases} rSTC.\,\mathrm{B}_{raw}\,.\,\mathrm{C}_{data}!stc \rightarrow \\ \qquad\qquad \mathrm{STC}\prime(inc(oTime, 2^{\mathrm{DS}})) \end{cases} \quad \text{if (inc(oTime,}2^{\mathrm{DS}}\text{) = time)} \tag{3.51}$$

$$\mathrm{STC}_{check}(inc(oTime, 2^{\mathrm{DS}}), time) \quad \text{otherwise}$$

The STC$\prime$ process 3.45 shows an example of how the *BiChan* could be used to connect intermediate processes. The *lSTCThro* set defines the list of tokens that the STC process expects to receive from the *lSTC* channel and pass onto the *rSTC* channel. Equivalently, the *rSTCThro* set defines the list of tokens that the STC process expects to receive from the *rSTC* channel and pass onto the *lSTC* channel without any processing.

$$channel \quad lDDC,\, rDDC \;:\; Block.Command.DataWord \tag{3.52}$$

$$lDDCThro = \{\mathrm{B}_{raw}\,.\,\mathrm{C}_{data}\} \tag{3.53}$$

$$rDDCThro = \{\mathrm{B}_{isa}\,.\,\mathrm{C}_{data}\} \tag{3.54}$$

$$\mathrm{DDC} = lDDC.\,\mathrm{B}_{cond}\,.\,\mathrm{C}_{cond}?cond \rightarrow tock \rightarrow \mathrm{DDC}_{check}(cond) \tag{3.55}$$

$$\Box\; BiChan(lDDC,\, rDDC,\, lDDCThro,\, rDDCThro,\, \mathrm{DDC},\, \ldots) \tag{3.56}$$

$$\mathrm{DDC}_{check}(cond) = rDDC.\,\mathrm{B}_{raw}\,.\,\mathrm{C}_{data}!ddc \rightarrow \tag{3.57}$$

$$rDDC.\,\mathrm{B}_{cond}\,.\,\mathrm{C}_{data}?npv \rightarrow \tag{3.58}$$

$$\begin{cases} lDDC.\,\mathrm{B}_{isa}\,.\,\mathrm{C}_{cond}!npv \rightarrow \mathrm{DDC} & \text{if (npv = cond)} \\ tock \rightarrow \mathrm{DDC}_{check}(cond) & \text{otherwise} \end{cases} \tag{3.59}$$

The DDC process in Eqn. 3.55 is able to perform a conditional input by continually polling the PHY for the desired value. The instruction is finished when the right value is observed and subsequently communicated back to the ISA unit. The DDC unit also uses the *BiChan* mechanism to pass tokens not addressed to it as shown in Eqn. 3.56.

$$channel \quad lPHY, rPHY \; : \; Block.Command.DataWord \tag{3.60}$$

$$\mathrm{PHY} = \mathrm{PHY}\prime(0) \tag{3.61}$$

$$\mathrm{PHY}\prime(opv) = \tag{3.62}$$

$$\Box \, \forall \, npv \in Boolean \Rightarrow rPHY.\,\mathrm{B}_{raw}\,.\,\mathrm{C}_{data}?npv \rightarrow \mathrm{PHY}\prime(npv) \tag{3.63}$$

$$\Box \, lPHY.\,\mathrm{B}_{raw}\,.\,\mathrm{C}_{data}\,.ddc \rightarrow lPHY.\,\mathrm{B}_{cond}\,.\,\mathrm{C}_{data}!opv \rightarrow \mathrm{PHY}\prime(opv) \tag{3.64}$$

$$\Box \, lPHY.\,\mathrm{B}_{raw}\,.\,\mathrm{C}_{data}\,.stc \rightarrow lPHY.\,\mathrm{B}_{isa}\,.\,\mathrm{C}_{data}!opv \rightarrow \mathrm{PHY}\prime(opv) \tag{3.65}$$

The PHY process 3.61 is responsible for monitoring the state of an external communication wire and reporting it to requesting processes (DDC or STC in this case). In Eqn. 3.63, B_{raw} is the block address of type *Block*, C_{data} is the command of type *Command* and *npv* is the *n*ew *p*ort *v*alue of type *DataWord*, which is used to communicate any data of size DS. Notice that the *rPHY* channel is theoretically able to communicate any data value of size DS since it was defined like all other multiplexing channels: *Block.Command.DataWord*. In Eqn. 3.63, however, the *rPHY* channel is only intended to communicate a subset of those values represented by the *Boolean* set.

$$channel \quad rISA \; : \; Block.Command.DataWord \tag{3.66}$$

$$\mathrm{ISA} = (\sqcap \, \forall \, data \in Boolean \Rightarrow rISA.\,\mathrm{B}_{cond}\,.\,\mathrm{C}_{cond}!data \rightarrow \tag{3.67}$$

$$rISA.\,\mathrm{B}_{isa}\,.\,\mathrm{C}_{cond}?data \rightarrow \mathrm{ISA}) \tag{3.68}$$

$$\sqcap \, (\sqcap \, \forall \, time \in DataWord \Rightarrow rISA.\,\mathrm{B}_{tmd}\,.\,\mathrm{C}_{tmd}!time \rightarrow \tag{3.69}$$

$$rISA.\,\mathrm{B}_{isa}\,.\,\mathrm{C}_{data}?data \rightarrow \mathrm{ISA}) \tag{3.70}$$

$$\sqcap \, (\sqcap \, \forall \, time \in DataWord \Rightarrow rISA.\,\mathrm{B}_{tmd}\,.\,\mathrm{C}_{stm}!time \rightarrow \mathrm{ISA}) \tag{3.71}$$

An abstract instruction set is defined as the ability of the system to perform any sequence of instructions in the absence of a more concrete specification of this sequence. This idea will become clearer when the ISA-*Oriented Specification*s of a communication protocol are discussed in Chapter 6.

Finally, the top-level SYSTEM process can be defined as the linear chaining of the individual building blocks as in Eqn. 3.72.

$$\mathrm{SYSTEM} = \mathrm{ISA} \underset{rISA \gg lSTC}{\parallel} \mathrm{STC} \underset{rSTC \gg lDDC}{\parallel} \mathrm{DDC} \underset{rDDC \gg lPHY}{\parallel} \mathrm{PHY} \tag{3.72}$$

The study in this chapter was performed using a version of the system that only had three instructions implemented in the ISA process (Conditional Input, Timed Input and Time Setting). Even with this minimal set of functions modelled, the resulting state machine for the individual building blocks was very large. This is evident from Table 3.1 which shows the metrics for individual processes as well as the top-level system.

The metrics in Table 3.1 were extracted iteratively by changing the size of the *DataWord* set expressed in bits ($DataSize = \mathrm{DS} = \log_2(|DataWord|)$).

DS	ISA		STC		DDC		PHY		SYSTEM	
	States	Trans.	States	Trans.	States	Trans.	States	Trans.	States	Trans.
1	13	20	50	146	20	47	6	12	308	1300
2	17	32	252	892	46	141	12	32	1648	10256
3	25	56	1496	6104	122	473	24	96	9920	101440
4	41	104	10032	44848	370	1713	48	320	66304	1208576

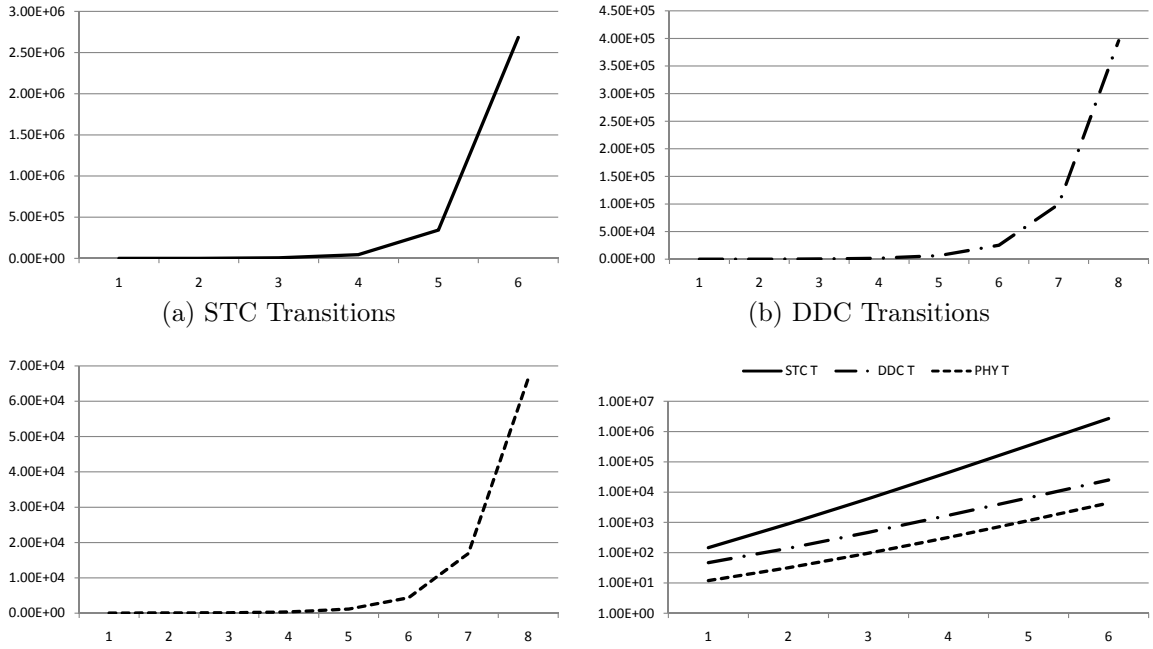
Table 3.1 Metrics of Data and Control Multiplexing System

Table 3.1 shows from this early stage that the number of possible states and transitions of the overall system is very large, even for small values of the communicated data-type (DS = 4).

When the number of transitions presented in Table 3.1 was plotted against DS, it became apparent that the relation is logarithmic. This is obvious from Figure 3.3d, where the number of transitions in various processes were plotted against DS on a logarithmic scale. Figure 3.3 also shows that the curves are always monotonically growing, at least for the gathered metrics. Looking at the CSP models in Eqns. 3.32 to 3.72, there is no reason to suspect otherwise− as the size of the DS variable grows, all related channel communications and associated processes grow.

Moreover, the DDC and PHY processes defined in Eqns. 3.55 and 3.61 respectively have been modelled in such a way that they are independent of the *DataWord* set and its associated configuration parameter (DS). Nevertheless, Figures 3.3b to 3.3d show that not only those processes are indeed dependent on DS, but they also show that the relation is logarithmic.

The need for further observability into the model-checker and its state machine representations is highlighted next. Such observability will hopefully uncover the reason behind the high level of dependency of the above processes and their state machines on DS.



(a) STC Transitions (b) DDC Transitions

(c) PHY Transitions (d) All, logarithmic scale

Figure 3.3 Number of Transitions vs. *DataSize*

At earlier stages of this study, the specification and verification of the system were aided by the use of the FDR model-checker [18] and its CSP_M scripting language [43]. However, as the study progressed, the Tool Command Language (TCL) interface of FDR proved useful in automatically constructing and performing model-checking tasks. It was noticed that this interface had limited documentation in the official manual of FDR, compared to the TCL/TK interface. Additional documentation of the TCL interface of FDR was found in *FDRExplorer* [44]. After extensive use of the command-line TCL interface, the need for further observability and higher levels of abstractions to represent state machines became apparent. Improving the observability and usability of model-checkers would help improve the adoption of the model-checking technology for the following reasons.

- Improving the maturity of the model-checking tools.
- Increasing the supporting evidence about the effectiveness of such verification methodology for verifying complex systems.
- Increasing the possibility of integrating model-checkers in the hardware design flow.

Woodcock et al. [45, page 31] reported that the second and third use of a formal technology and tool chain can lead to order-of-magnitude cost reductions. This is an example which highlights the need for improvements in the usability of the model-checking tools. Hoare and Misra [46] provided good insight into the challenges facing model-checking tools over the next 20-50 years. Both Bicarregui et al. [47] and Hoare and Misra [46] emphasised the need for greater efforts in the research and development of formal model-checking tools.

An automatic generation of graphical representations of state machines would enable the visual inspection of those state machines and the analysis of the associated processes. This would improve the usability of FDR, especially for hardware modelling purposes. A State Machine Visualiser for processes was developed as part of this framework. This is discussed in Section 3.5.

3.5 State Machine Visualiser

State machine description is an essential phase of any hardware design cycle. This is typically performed at an early stage. These state machines are then converted into another abstraction either manually or sometimes automatically if available tools support that transformation.

This is not the case for FDR, therefore design has to be manually converted into CSP description. FDR, on the other hand, uses the CSP description to compile each process into internal representations of state machines. This is evident from the *transition* list accessible through the TCL interface of FDR. As a first step in the verification process, it is important to confirm that these two transformations are accurate and that they both preserve the intended functional description of each process. This is also necessary because original state machines with a handful of transitions were compiled into Indexed State Machine (ISM) objects in FDR with at least an order of magnitude increase in the number of transitions.

Observability of the internal structure of FDR state machine objects was only available through the *transitions* function of the ISM objects. Because the transitions list represented a directed graph of the internal state machine, it should be possible to visually inspect this graph and possibly correlate it to the original hand-crafted one. For small processes, the graph can be manually sketched and a comparison be made. But as the system grows, and the number of transitions in the graph increases, the process becomes cumbersome, tedious, and error-prone. For this reason, an automatic

conversion from a transitions list to a graphical representation of the underlying state machine is required.

An open system for graph visualisation called *Graphviz* [48] was instrumental in converting ISM objects to a graphical form. It has a plain text interface language called *Dot* [49]. A tool called FDRlei has been developed as part of this framework which helped in the visualisation process. This involved the reconstruction of the ISM object, using the various TCL interface functions provided by FDR. Then an intermediate representation of the transition list using the Dot language file format is generated. Finally, using the *Graphviz* system, this intermediate representation is compiled into a Portable Document Format (PDF) file. Though it is possible to add the rendering specifications of states and transitions using the *Dot* language, this is currently left to be handled automatically by the *Graphviz* system.

This whole process was encapsulated in the *ShowGraph* function, which is part of the *Graph* object in FDRlei [1]. At the time of writing, only a minimal set of the *Dot* language was employed as proof of the concept. A more elaborate implementation with advanced graphical representations is beyond the scope of this thesis.

This visual inspection and analysis was useful in the development of small and medium size processes ranging between a handful of transitions to 10s or possibly 100s of transitions in some cases.

As an illustration of the *ShowGraph* function, let us consider the following example: A single-entry clocked buffer has two channels, an input channel (in), and an output channel (out). The number of clock cycles between the input and output operations is represented by $Delay$ and the total number of unique items that can be held in the buffer is represented by $DataItems$ in the following CSP description:

$$Delay = 1 \tag{3.73}$$

$$DataItems = \mathrm{DI} = 5 \tag{3.74}$$

$$Data = \{0 \,..\, (DataItems - 1)\} \tag{3.75}$$

$$channel \quad in,\, out : Data \tag{3.76}$$

$$channel \quad tock \tag{3.77}$$

$$Copy = in?x \rightarrow Copy'(Delay,\, x) \tag{3.78}$$

$$\Box\ tock \rightarrow Copy \tag{3.79}$$

[1] A *Linux(Intel) 64 bit* binary compilation of FDRlei is available online along with a brief description at `www.cs.bris.ac.uk/~kharmeh/thesis/fdrlei.tar`.

$$Copy'(\mathrm{N},\, x) = \begin{cases} tock \rightarrow Copy'(\mathrm{N}-1,\, x) & \text{if N} > 0 \\ out!x \rightarrow tock \rightarrow Copy & \text{otherwise} \end{cases} \tag{3.80}$$

Figure 3.4 illustrates the state machine drawing of the *Copy* process in Eqn. 3.78. It results from applying the *ShowGraph* function to the compiled ISM object in FDR-lei.

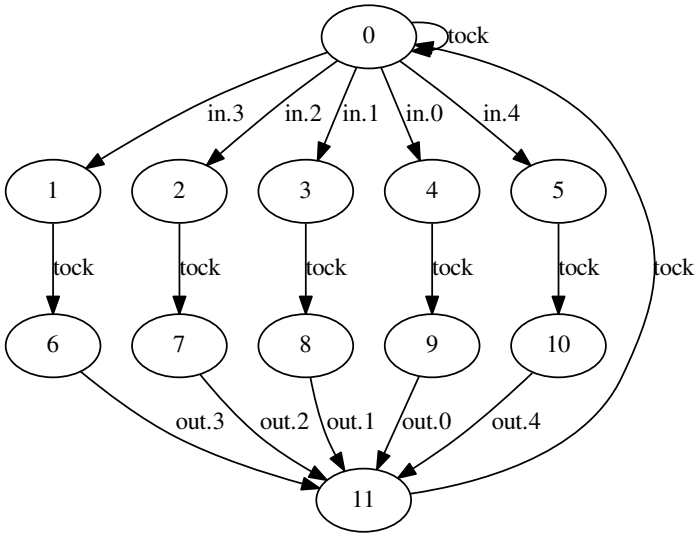


Figure 3.4 State Machine of the Copy Process

The ability to visualise CSP processes in the form of graphical representations of state machines provides the desired observability into FDR's transformation mechanisms. This gives the necessary confidence that the functionality specified earlier in the design cycle is correctly captured using CSP descriptions.

In addition, the visualisation of large processes with hundreds or possibly thousands of transitions was crucial in gaining a better understanding of how various design parameters affect the overall system. For example, if the *DataItems* held by the *Copy* process above were changed to 10, and the *Delay* parameter became 3, then the graph of the *Copy* process changes considerably as demonstrated in Figure 3.5.

By examining the automatically generated graphs, it was possible to see how each CSP construct was transformed into its state machine representation. Furthermore, it was possible to see how each process element or parameter (such as *Delay* in the *Copy* process or *BiChan* in the STC process) affects the overall state machine of that process. These visualisation capabilities helped in the modelling of lower-level processes with a small number of states and transitions. All the state machine graphs demonstrated throughout the thesis were generated using the *ShowGraph* function

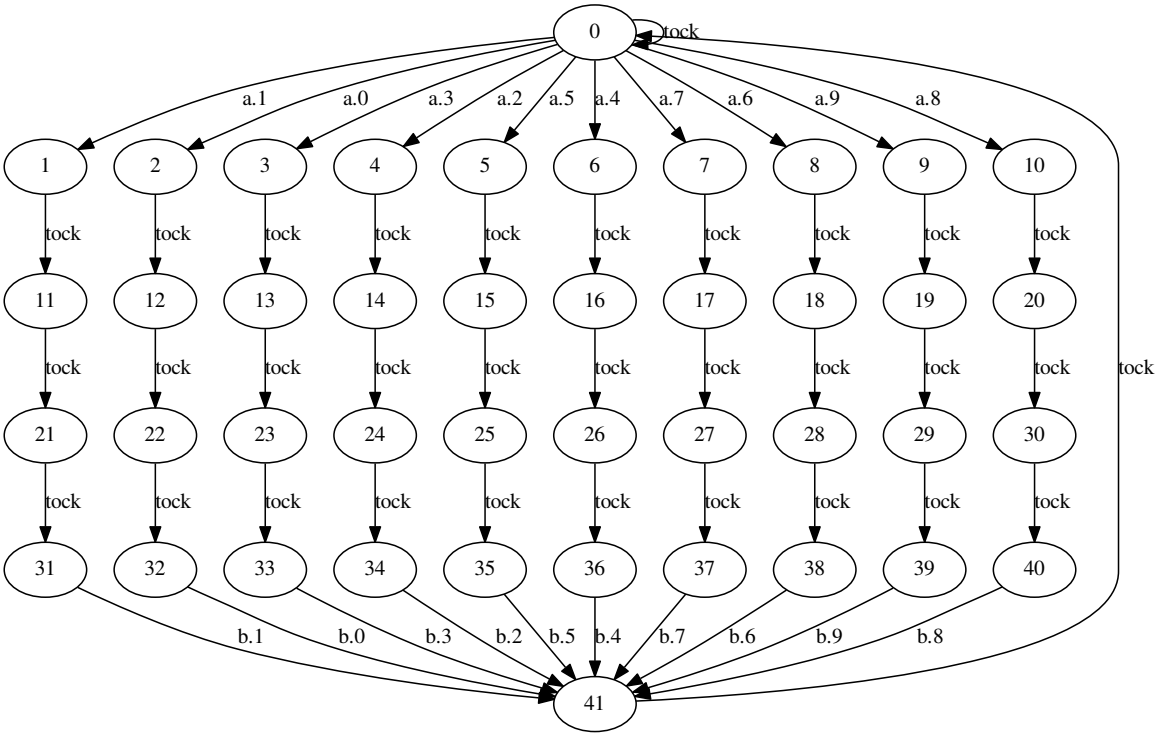


Figure 3.5 State Machine of the Modified Copy Process

discussed here. The contrivance of such a function and its subsequent use in demonstrating the power of the CSP specification and verification techniques is an important contribution of this section.

Finally, it is understood that an industrial release of FDR has built-in state machine visualisation capabilities. That release or any related documentation is not readily available, hence an objective comparison was not possible. Also, *FDRExplorer* [44] has a graph visualisation script, which translates the ISM transitions list into a *JGraph* [50] file format. Even though this *JGraph* format looks similar to the *Dot* format used by FDRlei, the *JGraph* version used is an old one and existing tools do not support it anymore. In addition, the *JGraph* tools are proprietary software, which is not readily available for academic research.

3.6 Results and Summary

The graph drawing functionality discussed in Section 3.5 helped in the introduction of a number of optimisations:

1. In initial attempts, *BiChan* allowed passage for any event $e \in \Sigma$. This allowed for the construction of a more dynamic pipeline, in which processes would con-

sume tokens of relevance and forward all other tokens. This makes the construction of the system more flexible, but the complexity of such an approach was simply unmanageable. In Eqn. 3.13, this was narrowed down to selected sets of interesting events (lT and rT).

2. Processes were initially designed to offer passing tokens using the *BiChan* mechanism at many points in the process execution cycle. This was optimised to allow tokens through the process only in the idle state.

Figures 3.6, 3.7, and 3.8 show the resultant state machines of the PHY, DDC, and STC processes, respectively. These have been generated with a very small data set ($DataSize = 2$).

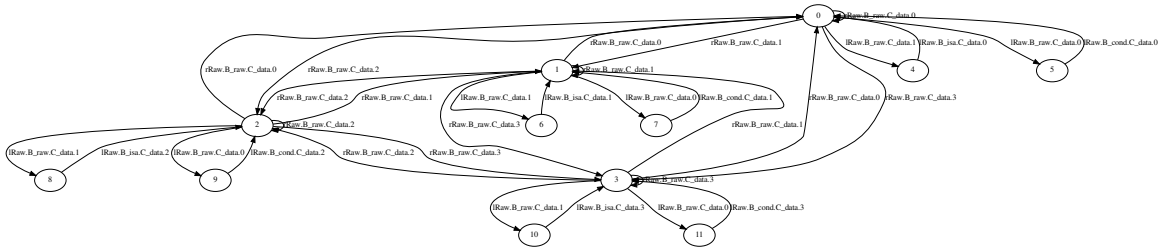


Figure 3.6 State Machine of the PHY Process [2]

[2]Figure is intended to be viewed online using the zoom function.

As discussed earlier, the PHY process has been modelled so that it only uses a subset of the *DataWord* set as defined in Eqn. 3.63. However, it is evident from Figure 3.6 that FDR has compiled the PHY process so that *all* values in the *DataWord* set are being used for communication and synchronisation on the $rPHY$ channel.

As the processes grew, it became clear that further analysis using metrics and state machines was becoming cumbersome and time-consuming and that the generated visualisations were simply incomprehensible to the naked eye. Figure 3.8 demonstrates that even for a relatively small state machine (≈ 250 states and ≈ 1000 transitions), visualisation might not be the best way for analysing the state machine and the associated process description. This, along with other observations made earlier about the Data and Control Multiplexing Pipeline highlighted the need for a better and more precise understanding of how the compiled state machines are related to the underlying CSP models and the associated data sets and definitions. This in turn has inspired the inception of the next chapter titled: *Complexity of Hardware Design and Model-Checking*.

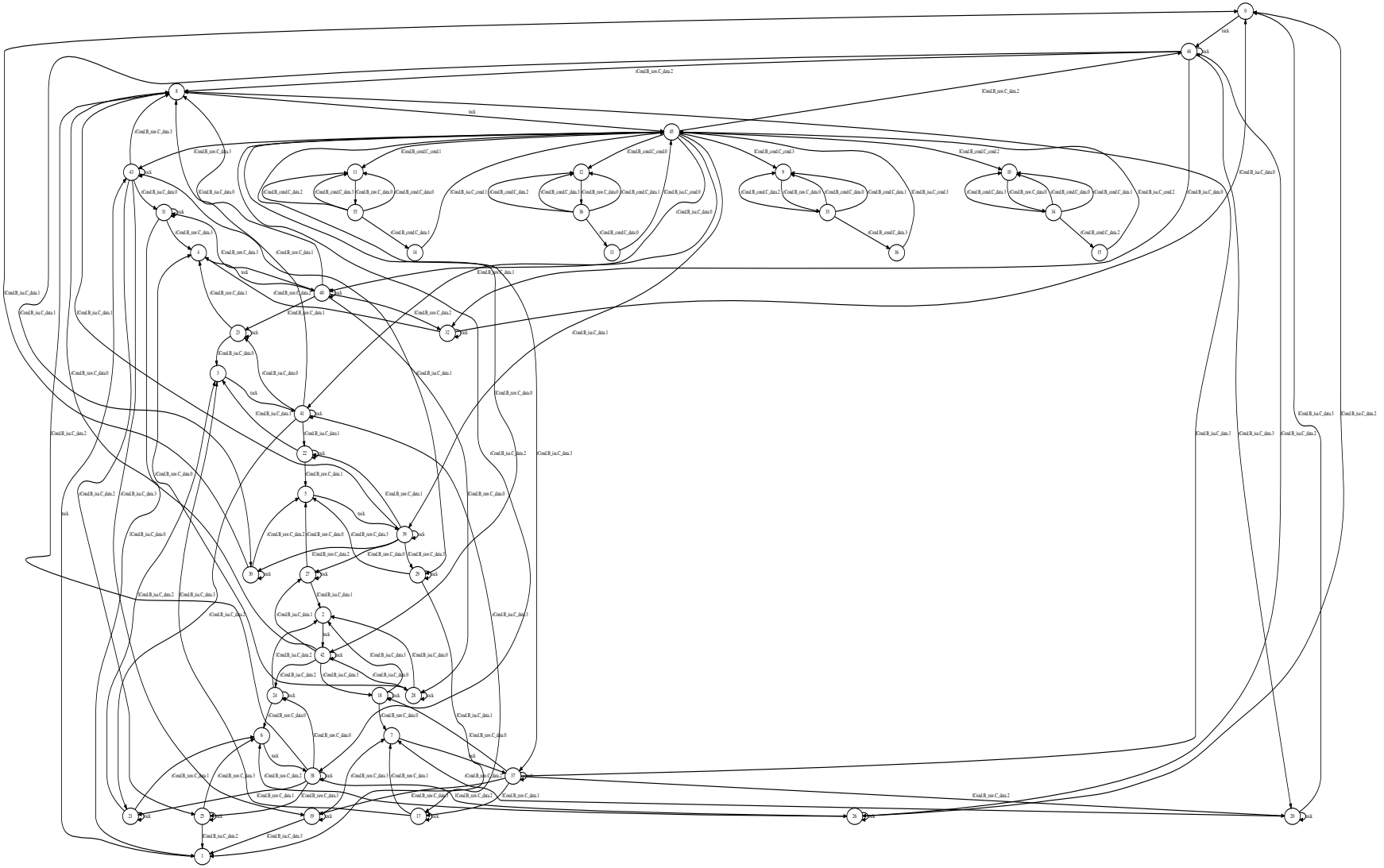


Figure 3.7 State Machine of the DDC Process [2]

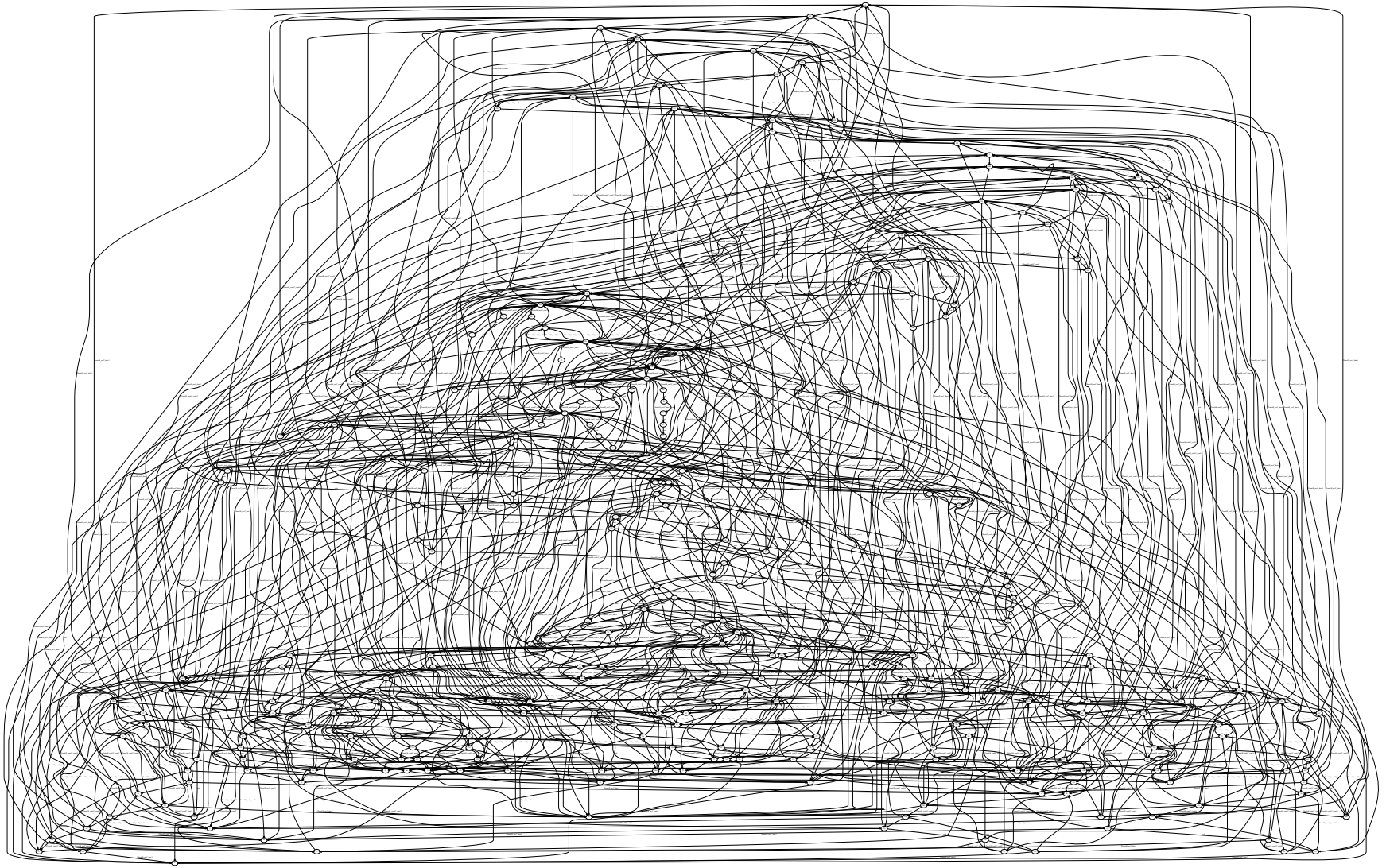


Figure 3.8 State Machine of the STC Process [2]

3.7 Future Work

State machine visualisation would benefit from state-space reduction algorithms. Leuschel and Turner [51] demonstrate two state-space reduction algorithms which help in the visualisation of state machines in the ProB model-checker. It would be interesting to compare the state machines and the associated visualisations generated by FDRlei with the ones generated by ProB.

Roscoe et al. [52] and Formal Syst. (Europe) Ltd. [18] present a few other compression algorithms that are implemented by FDR. It would also be interesting to visualise and compare the state machines generated by FDR before and after applying the compression algorithms.

CHAPTER 4

Complexity of Hardware Design and Model-Checking: Formal Analysis of State Machine Metrics

4.1 Motivation[1] and Chapter Structure

Objective 5 highlighted the need for the investigation of viable and objective hardware system complexity analysis mechanisms. The need for analysing the complexity of the model-checking tools was also highlighted. This is the main topic of this chapter. The chapter also addresses objective 6 partially: the need for an empirical analysis of the developed systems, which is discussed in Section 4.6.2.1.

In this chapter, objective definitions of *State-Space* and *Communication-Space Complexities* are presented in terms of underlying hardware system metrics. This should enable the quantification of the *State-Space Explosion* problem. In addition, it enables the identification and subsequent quantification of a *Communication Space Explosion* phenomenon.

This involved the analysis of the complexity and scalability of the model-checking tool itself (namely FDR) in handling exponentially growing complex systems.

A detailed case study is presented to show how the complexity formulae can be developed through the use of current model-checking tools coupled with automated equation solving techniques.

[1] Additional motivation for this chapter is omitted at the discretion of the author.

Abu Kharmeh et al. [3] presented a brief progress update which discussed some of the topics of this chapter.

First, Section 4.2 explains the complexity issues facing hardware specification and verification techniques. Section 4.3 provides previous work that identifies and addresses the complexity issues of model-checking and state-space explosion. It also discusses previous work that addresses the performance and benchmarking of the model-checking tools and algorithms. Then Section 4.4 describes how metrics of complied processes could be analysed in terms of specific configuration parameters and how this relationship could be expressed using complexity equations. Section 4.5 presents the challenges faced when extracting the complexity metrics from the FDR model-checker and the implemented optimisations to FDR to speed up this process. Section 4.6 presents a detailed complexity analysis of the data and control multiplexing approach to configurability presented in Chapter 3, using automated solving techniques for calculating the needed complexity equations. Section 4.7 describes how asymptotic analysis of the relevant complexity equations provides an insightful knowledge about the nature of the relationship between the analysed metric and the configuration parameter. This elevates the focus beyond the metrics and the detailed complexity equations. Finally, the chapter concludes with a summary in Section 4.8 and a discussion of possible future work in Section 4.9.

4.2 Introduction

Hardware designs must meet a complex set of functional and performance requirements. Above all, designs are increasingly required to support additional configurability options in order to meet the changing demands of system and application designers. Configurability adds to the ever-increasing complexity of hardware designs.

It is important to consider the design and verification complexity at an early stage of the design cycle. Doing so would provide a tractable approach for the design of complex systems by structuring the seemingly dependent, intertwined and complex design requirements into relatively simple blocks with specific requirements to be verified. However, the overall system resulting from the composition of these functional blocks still poses significant complexity and verification challenges. A major issue is the nature of the complexity growth of the combined system and its state space as the functional specification grows and more blocks are introduced.

The complexity of the abstract functional specification, as well as the complexity of the proposed implementation of the configurable hardware blocks are commonly

found to grow exponentially as additional blocks are added. Traditional simulation-based design and verification approaches are inherently intractable for verifying and analysing the complexity of state-of-the-art configurable hardware designs.

A formal specification and verification framework is best suited for analysing such hardware designs. This framework must be capable of managing the inherent complexity of such designs and the associated wide range of applications. This framework must also be scalable and extendible to enable the support of additional functional specification and systems beyond the initially anticipated ones.

This chapter focuses on complexity issues of the design and verification framework. In particular, the aim is to propose and demonstrate an objective and quantitative definition for Hardware Complexity. The focus here is on the complexity of hardware systems, when they are described as a collection of Communicating Sequential Processes (CSP) or state machines. Using asymptotic analysis of the resulting complexity definitions and formulae, it is possible to identify a configuration threshold at which the complexity of the system amounts to a *State-Space* or *Communication-Space Explosion*.

4.3 Background

Finite State Machines (FSM) are arguably the most universal abstraction for describing hardware systems. FSMs underpin many hardware specification, implementation and verification systems, including model-checking in general and the Failure-Divergences Refinements (FDR) model-checker [18]. It is therefore natural to identify and to quantify hardware complexity in terms of FSM complexity.

A well-known problem of using state machines for model-checking based verification is the State Space Explosion problem. Roscoe [Chapter 17, 38] explains this phenomenon. An objective and quantifiable way of measuring when state-space explosion is likely to happen is needed. Roscoe also discusses a number of different approaches to managing the state space and possibly avoiding this phenomenon. Lazic et al. [53] suggested that the approach is to induce the independence of a process from a design parameter or data-type according to a set of rules. By doing so, it is possible to devise mechanisms for managing the size of the examined data-type and hence the overall state space. This chapter instead addresses the dependence of a system on a data-type from a practical perspective by exploring the exact relationship between the complexity of the compiled state machines and selected design parameters

or data-types. This enables for better observability into the complexity of both the modelled systems and the model-checking abstractions being used.

In his thesis, Allis [54] defines the *State-Space Complexity* to be "the number of legal game positions reachable from the initial position of the game". However, this is simply a measurement of the state space.

Yao [55] explains *Communication Complexity* to be concerned with finding the minimum number of communication events that are exchanged between two parties in order for them both to compute a common function. Similar to Allis, Yao defines the complexity to be simply a statistical count or a *metric* of some entity (communication events in this case). Again, this is simply a measurement of the communication space and does not shed any light on its complexity.

4.3.1 Performance and Optimisation of Model Checking

For analysing the complexity of state machines and also as a measure of the scalability of the developed framework, benchmarking and statistical analysis of different metrics are sometimes a good starting point, as will be discussed in Section 4.5. Examples in literature on the scalability of different model-checking techniques and optimisations exist.

Roscoe et al. [52] describe modelling and compression methods using CSP and FDR to verify systems as large as 10^{1000} communicating processes. Though it serves as an insight on the possible capabilities of FDR, the approach only applies to the hierarchical construction of systems. Roscoe et al. [52] report that the system *perhaps* has $7^{10^{1000}}$ states.

Using a variety of systems and communication protocols to benchmark their FDR-Compliant validation tool, Leuschel and Fontaine [43] have chosen experiments with state machines of a maximum size of 10^5 states. That study was reported in 2008 and currently FDR can handle more, as will be demonstrated in Section 4.5.

Clarke et al. [56] describe another approach for managing the state space. Namely, they illustrate the Binary Decision Diagrams (BDD) algorithm for reducing the state space and consequently managing the state-space explosion problem. While BDD is a useful compression technique when implemented as part of a model-checker, it would not provide any information about the complexity of the models being checked and their dependencies or inefficiencies. The compressed and optimised state machines are of no use outside the model-checker, unlike the analysis technique proposed in this chapter, which would result in optimising the original models. Using BDD, Clarke

and Lerda [57] and Clarke et al. [58] reported state machine sizes of 10^{20} and 10^{120} states, respectively.

Other methods for managing the state space exist. Turner et al. [59] demonstrate the application of a method called Symmetry Reduced Model-Checking for the B language. Up to 231 speedup in the verification time on selected problems was reported. Exponential improvements to the state space were also reported.

Nalumasu and Gopalakrishnan [60] presented perhaps the most relevant benchmark, where the maximum number of states visited by two different model-checkers (SPIN and PV) was 5×10^6.

In a more recent development, Pelanek [61] discusses the Benchmarks for Explicit Model-Checkers (BEEM) initiative. This is an interesting development in the quest for an objective comparison to be made between various model-checking tools and the capabilities of each one. The online portal shows the benchmark results for up to 7.5×10^7 using the Distributed Verification Environment (DiVinE). Benchmark results are also available for the SPIN [62] model-checker.

In initial experimentation, FDR was able to verify various properties about systems as large as 8×10^6 states.

There is little evidence to compare the performance of different model-checking tools and algorithms in terms of time and memory required to perform verification checks (except for SPIN and DiVinE, as mentioned earlier).

The detailed benchmarking and optimisation analysis of FDR was essential to gauge its ability for analysing the complexity of the demonstrated configurable communication system, as discussed in Section 4.6.2.1. This was also a good indication of the ability of FDR for subsequently verifying functional and performance properties of the system, as discussed later in Chapter 7.

4.4 Hardware Complexity: From Metrics to Complexity Equations

The structured approach to designing a complex system mentioned earlier divides the state space of the complex system into more tractable functional units. The system specification that results from combining these functional units, however, is still complex in terms of its state space.

When using a formal specification, the structured approach exposes unforeseen configurations and interactions between the independent functional blocks. The modelling formalism (CSP in this case) might also introduce hidden complexity aspects of

the system not originally apparent or anticipated. This is apparent in model-checking when the system has to be enumerated and evaluated with respect to all possible values of a data-type for all possible functional blocks and subsequently for the full system. Also, all corner cases and design holes have to be considered and modelled for the system specification to be sound and viable for model-checking.

An essential part of formal modelling and verification is the evaluation of the system behaviour with respect to all possible interactions of all the processes of the system. Often, the complexity of the system is said to *explode* due to the factors described above. Model-checking tools face a great problem when they cannot predict the occurrence of a complexity explosion in advance. In such a case, tools could spend an unacceptable amount of time before they fail to perform the desired checks.

Metrics are increasingly being used to assess the complexity of a system. In light of the above discussion about the complexity of the verification and model-checking methodology, an obvious metric that might be used to assess the complexity is the number of states in the state space. This metric is usually accessible from the model-checker (as is the case with FDR). Other metrics to analyse the complexity of state machines, such as the complexity of the communication space are not so obvious and hence are not usually available by model-checkers.

4.4.1 State-Space Complexity

A hardware system is traditionally specified as (or compiled into as is the case with FDR) an FSM composed of states and transitions. When the size of the state machine becomes unmanageable by the model-checker, this scenario is considered to be a *State-Space Explosion*. However, there has not been a quantifiable approach for accurately predicting the occurrence of this phenomenon.

One way to address this problem is through parametrised verification and data *independence* analysis as suggested by Roscoe [38]. In systems where the number of building blocks is large and each block introduces new data-types, it becomes apparent that what is needed is to assess how *dependent* the system is on such data-types. Dependencies usually transcend individual building blocks and hence are apparent in overarching blocks and processes and ultimately in the top-level system. This affects the overall behaviour and complexity of the system and it becomes dependent on many data-types and configuration parameters of the individual functional blocks.

The specification of configurable systems normally depends on various configuration parameters and data-types. It is essential to find techniques to *measure* the dependence of the system on each of these data-type and configuration parameters.

Expressing the value of the state space and its relation to a specific configuration parameter would be best described using a formula or function. Such a formula can be called the *State-Space Complexity*. This complexity function will be useful for predicting the manageability of the verification problem, as will be demonstrated later in Section 4.6.2.1. Most importantly it will be possible to provide insight as to when the *State-Space Explosion* problem may occur and how to avoid or counteract its effects.

Ideally, a model-checking tool would give this information at an early stage of the verification task through static analysis of the machine-readable specification. This static analysis stage would fit between two stages of the compilation process of a model-checker, possibly as an additional phase to the semantic analysis stage. The specific value of each parameter in question could either be implicitly extracted from the model or provided explicitly. Then, by possibly interpolating previously provided benchmark results, the model-checker can accurately predict the time and memory requirements for performing the verification task in question and provide warnings about possible bottlenecks. Consequently, one can decide whether to continue with the verification task or not.

In the absence of tool support for producing the complexity formulae and the availability of the state count, an iterative process can be devised to establish how a configuration parameter affects the overall size of the state machine. The state space is then presented as a mathematical formula: a polynomial function or in particular cases an exponential function. This function represents how the state space changes as a specified parameter changes.

As an illustrative example, let us consider the *Copy* process presented in Eqn. 3.78. By varying the value of *DataItems* (or DI in short) between 1, 2, 3, 4 and 5, the corresponding state space was 4, 6, 8, 10 and 12 respectively. Hence, in this case, the *State-Space Complexity* function for the *Copy* process ($Copy_{SComplex}$) is:

$$Copy_{SComplex}(\mathrm{DI}) = 2 + 2 \times \mathrm{DI} \tag{4.1}$$

Using the above complexity function, the state space can be accurately predicted for any value of DI.

For now, the state-space analysis will be restricted to polynomial complexity functions of degree n:

$$SComplex(x) = c_0 + c_1 \times x + c_2 \times x^2 + \ldots + c_n \times x^n \tag{4.2}$$

In Section 4.4.3, an exponential form of the complexity function will be discussed.

It is not usually straightforward to find the coefficients of the complexity function and an iterative process using an equation solver proved useful in the absence of the support from formal modelling tools. This will be demonstrated in the case study in Section 4.6.2.1.

4.4.2 Communication-Space Complexity

An important aspect for quantifying the complexity of CSP models is their communication complexity. By analogy with the state space and the *State-Space Complexity* it is possible to define the communication space and its associated *Communication-Space Complexity*.

It is also possible for a specification to exhibit *Communication-Space Explosion* which is analogous to *State-Space Explosion*. It gives rise to the problem of predicting when the total number of communications between agents in a communication system grows out of control (possibly exponentially) as the parameters or inputs for the system increase. This prediction is performed using the *Communication-Space Complexity* formulae described later in this section.

The *communication space* of a CSP process shall be defined as the total number of edges in the state space. This can be extracted from FDR as the total number of *transitions* in the compiled FSM. This metric indicates how well connected the FSM graph is. Communication space is a rather interesting metric because two FSMs with the same state space can have different communication spaces, as demonstrated next.

Let us consider the following adjustment to the *Copy* process:

$$CopyNew = in?x \rightarrow Copy''(Delay,\, x) \tag{4.3}$$

$$\Box\ tock \rightarrow CopyNew \tag{4.4}$$

$$Copy''(\mathrm{N},\, x) = \begin{cases} \left.\begin{array}{l} tock \rightarrow Copy''(\mathrm{N}-1,\, x) \\ \Box\ tock \rightarrow Copy''(\mathrm{N}-1,\, (x+1) \bmod \mathrm{DI}) \\ \Box\ tock \rightarrow Copy''(\mathrm{N}-1,\, (x-1) \bmod \mathrm{DI}) \end{array}\right\} & \text{if } \mathrm{N} > 0 \\ out!x \rightarrow tock \rightarrow CopyNew & \text{otherwise} \end{cases} \tag{4.5}$$

The *CopyNew* process now optionally modifies data items by incrementing or decrementing their values with each clock cycle. Figure 4.1 shows the state machine of the *CopyNew* process.

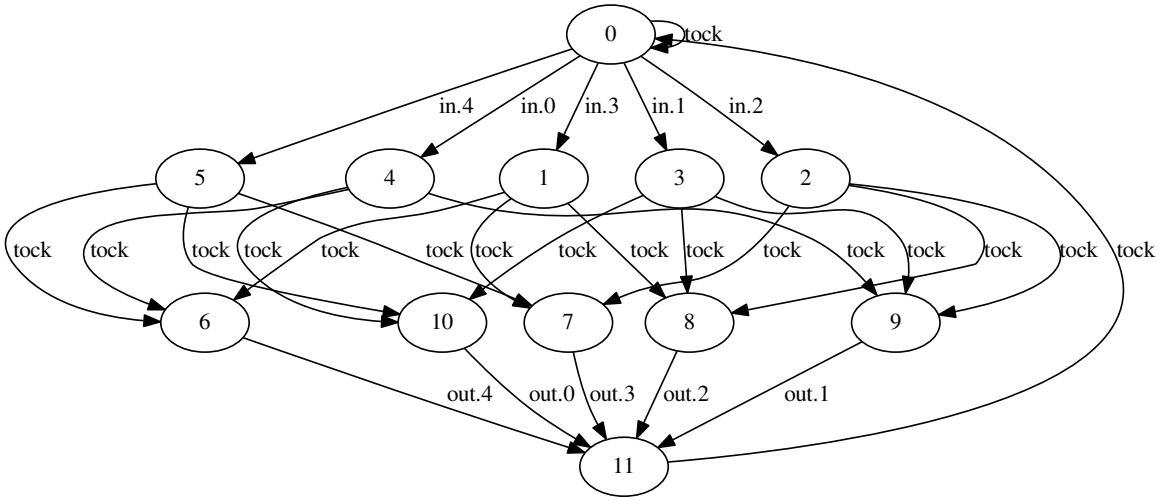


Figure 4.1 CopyNew Process with $Delay = 1$ and $DataItems = 5$

Two state machines with the same *State-Space Complexity* do not necessarily have the same *Communication-Space Complexity*, as can be evident from Figure 4.1 when compared to Figure 3.4. The transitions count of both *Copy* and *CopyNew* processes were extracted by varying DI. These are presented in Table 4.1.

DataItems (di)	Transitions ($Copy$)	Transitions ($CopyNew$)
1	5	5
2	8	8
3	11	11
4	14	22
5	17	27
6	20	32

Table 4.1 Communication-Space Change

By analysing the dependency of the *Transitions* metric on the value of *DataItems* (DI), the *Communication-Space Complexity* function for the *Copy* process was found to be:

$$Copy_{CComplex}(\mathrm{DI}) = 3 \times \mathrm{DI} + 2 \tag{4.6}$$

For *CopyNew*, the complexity function becomes:

$$CopyNew_{CComplex}(\mathrm{DI}) = \begin{cases} 3 \times \mathrm{DI} + 2 & \text{if DI} \leq 3 \\ 5 \times \mathrm{DI} + 2 & \text{otherwise} \end{cases} \tag{4.7}$$

As will be discussed in Section 4.4.3, the focus is on the asymptotic behaviour of the complexity function (similar to *Big O* notation) and hence the *Communication-Space Complexity* function of *CopyNew* for DI ≤ 3 can be ignored, in which case the function becomes a standard polynomial similar to Eqn. 4.2.

The complexity formulae above show that two processes with the same state space can have different communication spaces. Hence, the obvious state-space metric that is typically used for assessing hardware complexity is not always sufficient and one should also consider the communication space when assessing how complex a hardware system is.

As before, the communication complexity function can be generalised as a standard polynomial of degree n.

The notion of taking the hardware complexity analysis beyond simple statistics by relating those statistics to underlying hardware system inputs, outputs and configurations is important. By doing so, one can understand and hopefully manage the complexity of the system in an informed manner. The *State-Space Complexity* and *Communication-Space Complexity* are two aspects that reflect how complex a state machine is. In the model-checking environment these are crucial to:

- seeing precisely how processes are dependent on each data-type which helps to avoid unnecessary complexities and dependencies introduced by unwanted functional complications;
- helping engineers and scientists to understand how the CSP description is related to the underlying state machine objects and how to use CSP to produce optimal designs and state machines;
- enabling the model-checker to statically estimate the complexity of the verification problem. This could be translated into an estimate of the run-time and memory requirements of the problem. Subsequently a decision (possibly assisted by the user) can be made whether to proceed with the task or not;
- enabling better benchmarking of model-checking abstractions, algorithms and technologies. For this to be possible, standards are needed for various aspects of the model-checking technology, from modelling interfaces and languages through complexity analysis and benchmarking; and
- helping engineers and scientists to understand how model-checkers work in general.

Other aspects related to a specific model-checker might not be so obvious, but could be informative as to the complexity of the model-checking algorithm. For example, the acceptance and refusal sets which represent some of the state variables computed by FDR for performing the required verification tasks might also shed some light on the complexity of the model-checking algorithm, but not necessarily on the complexity of the actual models being checked. Different model-checkers typically have other factors of interest.

However, because abstract state machines of the form of states and transitions are the common mathematical (or formal) model underlying all model-checkers and also hardware design and implementation techniques, analysing their complexity is seen as the one and only aspect that could be considered universal for analysing hardware complexity.

4.4.3 Asymptotic Analysis of Space Complexity

When modelling digital systems in CSP, such as the one discussed in Section 4.6.2.1 or a Compiler, such as the one described by Roscoe [Chapter 18, 38], it is very plausible to express the system complexity in terms of the size of a data-type or the number of shared variables. Suppose the system complexity is to be analysed in terms of a data-type called *Integer*. Let S be the size of the *Integer* data-type in bits. Typically, model-checking algorithms would verify the system with respect to all possible values of each variable in the system. In that case, the system would be dependent on the total number of possible values that can be represented in such a data-type. But because S is the total number of bits an *Integer* can have, then the total number of values represented by an *Integer* is 2^{S}. Hence the complexity polynomial becomes of exponential nature:

$$ExpComplex(\mathrm{S}) = c_0 + c_1 \times 2^{\mathrm{S}} + c_2 \times 2^{2\times\mathrm{S}} + \ldots + c_n \times 2^{n\times\mathrm{S}} \tag{4.8}$$

Then the asymptotic complexity of Eqn. 4.8 would be $ExpComplex(\mathrm{S}) \in O(2^{n\times\mathrm{S}})$.

Because the above asymptotic complexity of Eqn. 4.8 is of exponential nature, it is only practical to analyse and verify the system for small values of S. This will become apparent in Section 4.6.2.1. In such cases, it might also be worth considering the whole complexity function in Eqn. 4.8 instead of its asymptotic behaviour only. This is because the sum $c_0 + c_1 \times 2^{\mathrm{S}} + c_2 \times 2^{2\times\mathrm{S}} + \ldots + c_{n-1} \times 2^{(n-1)\times\mathrm{S}}$ might be larger than the asymptotic aspect itself ($c_n \times 2^{n\times\mathrm{S}}$) for small values of S.

Both the full complexity function for small n, as well as the asymptotic aspect will be analysed in the case study in Section 4.6.2.1.

The size of the state space and communication space is an objective, abstract and stand-alone metric that is useful for analysing state machine complexity. Using such analyses, conclusions could be made about the run-time, memory requirements or possibly other dependent, but subjective, aspects. These conclusions could be made about different implementation and verification techniques, in addition to model-checking. In the analysis of hardware systems, other aspects of relevance include chip-area and power consumption of the underlying hardware. It would be very interesting to reflect on such aspects, possibly by extending the above complexity analysis to include multiple configuration parameters or possibly different metrics.

4.5 Complexity Metrics In Practice

In order to analyse the complexity of a process, various communications between FDR (instantiated as a Tool Command Language (TCL) server) and FDRlei take place. These communications involve compiling and explicating a process to establish its transitions. The transitions are then used for finding the state-space and communication-space metrics of the process. This procedure is then repeated iteratively by changing the configuration parameter under scrutiny according to a set of prefixed values. Finally, the generated metrics are processed by an equation solver to establish the coefficients of Eqn. 4.8. Further information about the process of driving the equation solver are discussed in Section 4.6.

The iterative process of producing the needed metrics was proving costly as process sizes grew (beyond 3×10^5 states). It was observed that the time spent by FDR was growing unexpectedly large for such relatively small systems. This necessitated further investigation to identify the reasons behind the performance degradation experienced by FDR. This included performance profiling and usage model analysis of internal data structures as follows.

First FDR was built with profiling enabled. Then, by using TCL scripting to build the refinement checks step by step, an implementation process and a specification process were compiled into two ISM objects. Then those two processes were used to construct the object describing the check or assertion to be carried out. In FDR, this is called a *hypothesis* object [18]. Finally, the check is performed using the *assert* function which is part of the *hypothesis* object.

These performance experiments revealed that most of the CPU time was spent in this *assert* function. In other words, the time spent in the compilation of the CSP script into the needed ISM objects and the construction of the *hypothesis* object was negligible. The experiments also revealed that most of the time was spent in the *fdr2tix* front end application and the *state2* server of the FDR tools chain had minimum performance overhead.

The profiling results revealed that even for reasonable size systems (i.e. processes with 5×10^7 transitions) the performance profile of *fdr2tix* was alarming.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds    calls    name
 86.08   1711.08  1711.08  13937496   Set::add_singleton(...)
  3.03   1771.37    60.29  98459995   BTree::insert(...)
  1.26   1796.36    24.99         2   Supercombinator::tabulate_data()
  1.01   1816.35    19.99 160730614   BTree::insert_non_full(...)
  0.90   1834.24    17.89 104979264   BTree::find(...)
  0.90   1852.09    17.85     54673   DeltaCompressor::compress(...)
  0.82   1868.35    16.26  66526528   Supercombinator::walk_afters(...) const
  0.50   1878.29     9.94      3849   std::__introsort_loop(...)
  0.41   1886.44     8.15  64620525   Supercombinator::calc_initials() const
```

(a) Before optimisation

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds     calls   name
 20.91     48.31    48.31  98459995   BTree::insert(...)
  7.16     64.85    16.54 160730614   BTree::insert_non_full(...)
  7.02     81.06    16.21     54673   DeltaCompressor::compress(...)
  6.86     96.91    15.86 104979264   BTree::find(...)
  6.71    112.40    15.49         2   Supercombinator::tabulate_data()
  6.34    127.04    14.64  66526528   Supercombinator::walk_afters(...) const
  3.55    135.25     8.21      3849   std::__introsort_loop(...)
  3.14    142.50     7.25  64620525   Supercombinator::calc_initials() const
  2.41    148.06     5.56     75593   DeltaCompressor::uncompress(...)
```

(b) After optimisation

Figure 4.2 FDR Performance Profile

Figure 4.2a shows an extract of the information obtained by profiling the *fdr2tix* executable: the part of FDR tools chain which was the most CPU-intensive during the verification of an assertion of the following form:

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \text{IMPLEMENTATION} \setminus (\Sigma \setminus \{tock\}) \tag{4.9}$$

where:

$$\text{TOCKS} = tock \rightarrow \text{TOCKS} \tag{4.10}$$

and the IMPLEMENTATION process had about 5×10^7 transitions. More details about CSP assertions will be discussed in Chapter 7.

Figure 4.2a shows that 86% of the time was spent in a single function namely:

$$Set::add\_singleton(\ldots) \tag{4.11}$$

Further source-code analysis revealed that the *Set*::*add_singleton* function in Eqn. 4.11 is a generic set creation and manipulation algorithm. It was used by the process explication procedure: a procedure that enumerates the process by tabulating the transition system rather than deducing the operational semantics *on the fly*. This is supposed to make subsequent manipulation faster [18]. However, this procedure assumed that the enumerated states form an unordered set. Each time a new state was added to the set, a search was performed to find its position in the set before adding it. The complexity of this search and place algorithm for adding a single item was $O(\mathrm{N}^3)$, where N is the cardinality of the set being expanded. Practically, the relative performance of this algorithm to the whole verification process increased drastically as the size of the set increased, as evident from Figure 4.2a.

For this reason an optimised set creation and manipulation mechanism was implemented and integrated into FDR. It takes into account that the enumerated set of states representing a state machine is an ordered one. By keeping a pointer to the last item in the set, the complexity of the whole expansion process becomes $O(1)$ and the overhead of expanding the set (i.e. the *Set*::*add_singleton* function) disappears completely from the execution profile, as evident in Figure 4.2b.

Professor A. W. Roscoe of Oxford University's Computing Laboratory (current maintainers of FDR) has been made aware of this possible optimisation to FDR. Professor Roscoe reported in April 2012 that they "are presently planning FDR3: intended to be a complete rewrite".

The implemented optimisation resulted in vast performance improvements as demonstrated by Table 4.2.

The change was *tested* using a regression of 372 different assertions and the results were consistent with the unoptimised version. The regression consisted of 62 functional and performance checks performed on the communication system being developed. These checks were performed iteratively using a set of 6 different configurations. Table 4.2 shows a summary of the real-time execution results for 4 of those iterations along with the size of the IMPLEMENTATION process for each iteration.

Those improvements are presented here for the following two reasons.

Implementation		Before	After	Improvement
States	Transitions	(Sec)	(Sec)	(Factor)
353264	1842544	100	38	2.6
1669088	8887008	1842	249	7.4
9154496	49591744	452002	2358	191.7
57302912	320169014	46 years[2]	9 hours	45323.0

Table 4.2 FDR Performance Optimisation Summary

[2]Corresponds to predicted rather than real performance figure.

1. To highlight the fact that the case study in Section 4.6.2.1 was greatly facilitated by those optimisations. This is because it involved a large number of iterative compilation and metrics extraction.

2. To give strong evidence that time and space performance metrics extracted from the model-checker are not objective for assessing the complexity of the formal model.

As evident from Table 4.2, the impact of the optimisations increases rapidly as the size of the system increases. These optimisations enabled FDR to model-check much larger systems than was previously possible.

Finally, the performance improvements made to FDR were also very useful in Chapter 7, where functional and performance refinement checks were performed on multiple large-scale processes.

4.5.1 Future Work

Future optimisation work includes:

1. Further analysis and optimisations of large sets. In particular the structure of these sets and their unsorted expansion/access scenarios. The optimisation discussed in this section avoids the use of the *Set*::*add_singleton* function in a specific case. However, the function is still available and is being used in other cases. It might be worth optimising it for the general case of an unordered set expansion.

2. Analysis of the performance impact of the TCL interface. It is predicted that performance gain would result from integrating all the functionality offered by FDR and the FDRlei plug-in into a single compiled application, as suggested

by Prechelt [63]. Doing so would eliminate the need to go through many layers of function calls and programming abstractions (C++ application, TCL scripts, Linux system calls, ...) to reach the *fdr2tix* executable.

4.6 Automated Solving of Complexity Equations

As discussed in Section 4.4, the ability to assess the complexity of CSP processes through analysing the complexity of the associated state machines is essential for producing efficient and verifiable models. This has been made clear in Chapter 3 and also earlier in this chapter.

In the absence of a built-in solution in the model-checker, an alternative approach to developing the complexity polynomials would involve the iterative collection of the desired metrics (i.e. number of states or number of transitions) from the model-checker by varying the desired definition or variable (i.e. DS in Eqn. 3.44 or DI in Eqn. 3.78) and subsequently establishing the mathematical relation between the collected metrics and the associated variable.

Establishing the relation could be performed manually by searching for the c_0, c_1, c_2, ..., c_n coefficients of Eqn. 4.8 for simple state machines like the ones in Section 4.1. Another mechanism to finding those coefficients would involve fitting a curve to the experimental data (state machine metrics in this case) using automated curve fitting algorithms. This section describes the use of such algorithms for finding the *exact* solution to the complexity equations of the CSP processes presented in Chapter 3.

4.6.1 Background

Optimisation or curve fitting algorithms describe a field of mathematics with a set of algorithms that are used to relate collected experimental data to a mathematical formula. This is done through the optimisation of a target formula using a set of constraints and target variables [64].

The specific algorithm used depends largely on the analysed data and the associated mathematical formula. For example, the simplex algorithm is suitable for solving linear problems. The field of solving optimisation problems, where the objective function or target formula is not linear is called nonlinear programming (NLP). Several algorithms can be used in solving nonlinear problems, including the Gauss-Newton, the Marquardt-Levenberg, the Nelder-Mead and the steepest descent method [65]. Lasdon et al. [66], on the other-hand, describe a robust, efficient and easy-to-use NLP algorithm called the Generalised Reduced Gradient algorithm (GRG). Lasdon

et al. focus on software implementation, rather than on the mathematical properties of the algorithm. An exponential formula such as Eqn. 4.8 is a typical nonlinear problem that could be solved using the GRG algorithm.

A well established optimisation software is the *Excel Solver*, in which D. Fylstra *et al.* [67] implement a number of optimisation algorithms including the *Simplex LP*, GRG and *Evolutionary* methods, some of which are only available under commercial licence. The design and use of the *Excel Solver* along with its interface is well documented by Fylstra et al. [68]. This section discusses the experiments performed in solving the state machine complexity formulae using the *Excel Solver*, which is bundled with *Microsoft Excel*. The discussion will address the practical aspects of applying the algorithm to solving the required complexity formulae including the options used, some observations and implemented modifications to the algorithm.

4.6.2 Equation Solver Using the GRG Algorithm

Fylstra et al. [68] explain curve fitting as the act of expressing the experimental data as a mathematical equation in the form $y = f(x)$, where x is the *independent* variable which is controlled by the experimenter; y is the *dependent* variable which is measured; and f is the function, which includes one or more parameters used to describe the data.

In the case of the complexity equation described in Section 4.4 and the models discussed in Section 3.4, the *DataSize* or DS has been chosen as the independent variable for the analysis. This is due to the reasons already discussed in Section 3.6. As the models presented were refined further, other independent variables of interest came to exist. The discussion in this chapter shall be limited to the analysis of the DS independent variable.

The state machine metrics considered for analysis as dependent variables were:

1. Events (E): total number of unique events.
2. States (S): total number of states.
3. Transitions (T): total number of all transitions.

The spreadsheets were constructed by initialising all c_0, c_1, c_2, ..., c_n in Eqn. 4.8 using initial estimates and a sufficiently large n. In analysing the complexity of the models discussed in this chapter, values of $n > 3$ were never observed, hence $n = 5$ was deemed sufficiently large.

```
Sub SolveOnce(ChangeRange)

  SolverReset
  SolverOptions MaxTime:=100,iterations:=1000,Precision:=0.000001, _
    AssumeLinear:=False, StepThru:=False, Estimates:=1, _
    Derivatives:=1, SearchOption:=1, IntTolerance:=5, _
    Scaling:=False, Convergence:=0.0001, AssumeNonNeg:=False
  SolverAdd CellRef:=Range(Cells(F, FS), Cells(F, n + 1)), _
    Relation:=3, FormulaText:="0"
  SolverOk SetCell:=Cells(D, SS), MaxMinVal:=2, ValueOf:="0", _
    ByChange:=ChangeRange
  SolverSolve True

End Sub
```

Listing 4.1 Standard GRG Routine

In a typical solving scenario, the initial c_i values would be estimated before starting the GRG algorithm. However, since the initialisation procedure was to be used as part of an automated process to solve all the complexity equations for all the desired processes, all c_i values were initialised to 1.

By varying DS according to the collected data, the formulae for the predicted metrics were constructed and the initial values of the predicted metrics were calculated.

Once all the initial values had been calculated, the error differences for all values could be calculated and the target cell for optimisation could be defined as the sum of error squares SS for all data points:

$$\mathrm{SS} = \sum_{i=1}^{\mathrm{DS}_{max}} (Metric_m - Metric_p)^2 \tag{4.12}$$

where $Metric_m$ is the *measured* value for the metric as reported by FDR and $Metric_p$ is the predicted value for the metric using the complexity formula.

Bowen and Jerman [69] describe the initialisation process in detail. This initialisation process has been implemented as a *Visual Basic* macro, which is attached in Appendix A.

Various options to the solving algorithm have been considered. The solving routine along with the constraints used are defined in Listing 4.1 where:

- D is the error differences row;
- SS is the column number of the values calculated by Eqn. 4.12;
- F is the row number of where the factors (c_i) are defined;

- FS is the starting column for the factors; and
- n is the total number of factors.

In Listing 4.1, command lines ending with _ are continued on the next line. By default, all n factors are allowed to be changed by the GRG algorithm, hence:

$$ChangeRange = Range(Cells(\mathrm{F}, \mathrm{FS}), Cells(\mathrm{F}, n+1)) \tag{4.13}$$

Finally, the routine defines constraints on the c_i factors and the predicted metrics to be positive at all times.

It was observed that the best results were obtained by setting all c_i factors to zero. See Appendix A for full *Visual Basic* code including the initialisation procedure.

4.6.2.1 Case Study: Complexity of the Multiplexing Approach

In this section, the GRG algorithm discussed earlier is used to develop and analyse the complexity formulae of the processes presented in Section 3.4.

The independent variable used is DS: the logarithm of the size of the *DataWord* set in Eqn. 3.35 to the base 2.

Table 4.3 shows the results of applying the GRG solving procedure for all the essential processes in Section 3.4. The table shows the metrics as obtained from FDR, the predicted metrics using the solution obtained from the *Solver*, and the error values squared $(Metric_m - Metric_p)^2$. The table also shows the optimised SS results which were the target for optimisation.

Table 4.4 shows the factors of the complexity equations to all the processes in Table 4.3.

It is clear from Table 4.3 that, only in a few cases, the solver was able to find the exact complexity formulae and hence the optimised SS result was negligible. For example, the optimised SS for the STC process was between $1.42E$-05 and $4.98E$-04. In all other cases, the optimised SS was considered not negligible and hence required further investigation. In the case of the transitions solution of the DDC process, the result is alarming, since the SS optimisation target was considerably large ($1.37E$+04).

Looking at Table 4.4, it is clear that some of the produced multiplicands have very small (and negligible) values. For example, almost all the values for the $2^{4\times \mathrm{DS}}$ multiplicand had values smaller than $1E$-6. In those cases, the complexity equations are assumed to be of order less than 4, but the GRG algorithm is unable to reach that conclusion and hence it always produces a small residual value for the $2^{4\times \mathrm{DS}}$ multiplicand of the complexity equation. This was the motivation for the next section.

Table 4.3 Predicted Metrics and Error Using Standard GRG Solver

DataSize (DS)			1	2	3	4	5	6	7	8	SS
ISA	Events	Measured	12.00	20.00	36.00	68.00	132.00	260.00	516.00	1028.00	
		Predicted	8.26	16.51	33.04	66.14	132.72	270.08	605.52	2221.24	
		Error^2	14.02	12.15	8.75	3.44	0.52	101.55	8013.48	1.4E+06	3.89E+01
	States	Measured	13.00	17.00	25.00	41.00	73.00	137.00	265.00	521.00	
		Predicted	4.65	9.29	18.59	37.20	74.50	149.38	300.26	606.53	
		Error^2	69.79	59.40	41.08	14.41	2.26	153.27	1243.37	7315.16	1.87E+02
	Trans.	Measured	20.00	32.00	56.00	104.00	200.00	392.00	776.00	1544.00	
		Predicted	12.52	25.04	50.10	100.30	201.43	413.12	977.52	4323.77	
		Error^2	55.98	48.45	34.85	13.72	2.05	446.26	40608.78	7.7E+06	1.55E+02
STC	Events	Measured	20.00	36.00	68.00	132.00					
		Predicted	20.00	36.00	68.00	132.00					
		Error^2	0.00	0.00	0.00	0.00					1.42E-05
	States	Measured	50.00	252.00	1496.00	10032.00					
		Predicted	50.00	252.01	1496.02	10032.00					
		Error^2	0.00	0.00	0.00	0.00					4.98E-04
	Trans.	Measured	146.00	892.00	6104.00	44848.00					
		Predicted	146.01	892.01	6103.99	44848.00					
		Error^2	0.00	0.00	0.00	0.00					1.16E-04

Table 4.3 Continued ↓

DataSize (DS)			1	2	3	4	5	6	7	8	SS
DDC	Events	Measured	17.00	31.00	59.00	115.00	227.00	451.00			
		Predicted	14.14	28.28	56.58	113.27	227.64	469.87			
		Error^2	8.19	7.40	5.87	2.99	0.41	356.13			2.49E+01
	States	Measured	20.00	46.00	122.00	370.00	1250.00	4546.00			
		Predicted	18.62	45.09	121.84	370.61	1249.96	4546.97			
		Error^2	1.90	0.82	0.03	0.37	0.00	0.95			3.12E+00
	Trans.	Measured	47.00	141.00	473.00	1713.00	6497.00	25281.00			
		Predicted	25.48	101.94	407.76	1631.03	6524.11	26096.45			
		Error^2	462.90	1525.74	4256.65	6719.41	735.06	664954.86			1.37E+04
PHY	Events	Measured	10.00	16.00	28.00	52.00	100.00	196.00	388.00	772.00	
		Predicted	6.27	12.54	25.08	50.21	100.70	204.24	447.42	1492.23	
		Error^2	13.93	11.99	8.50	3.20	0.49	67.82	3531.18	518730.48	3.81E+01
	States	Measured	6.00	12.00	24.00	48.00	96.00	192.00	384.00	768.00	
		Predicted	5.97	11.95	23.90	47.82	96.02	198.15	493.32	2533.95	
		Error^2	0.00	0.00	0.01	0.03	0.00	37.83	11950.48	3.1E+06	4.71E-02
	Trans.	Measured	12.00	32.00	96.00	320.00	1152.00	4352.00	16896.00	66560.00	
		Predicted	4.53	18.11	72.46	289.82	1159.30	4637.19	18548.76	74195.03	
		Error^2	55.82	192.82	554.32	910.57	53.25	81332.97	2.7E+06	5.8E+07	1.77E+03

Table 4.3 Continued ↓

DataSize (DS)			1	2	3	4	5	6	7	8	SS
SYSTEM	Events	Measured	5.00	7.00	11.00	19.00					
		Predicted	4.93	6.94	10.96	19.01					
		Error^2	0.01	0.00	0.00	0.00					1.02E-02
	States	Measured	308.00	1648.00	9920.00	66304.00					
		Predicted	308.00	1648.00	9920.00	66304.00					
		Error^2	0.00	0.00	0.00	0.00					5.07E-06
	Trans.	Measured	1300.00	10256.00	101440.00	1.2E+06					
		Predicted	1321.58	10245.77	101442.21	1.2E+06					
		Error^2	465.90	104.58	4.90	0.01					5.75E+02

Table 4.3 Predicted Metrics and Error Using Standard GRG Solver

Multiplicand		1	2^{DS}	$2^{2\times DS}$	$2^{3\times DS}$	$2^{4\times DS}$
ISA	Events	5.47E-04	4.13E+00	3.62E-04	0.00E+00	2.66E-07
	States	3.42E-04	2.32E+00	1.83E-04		
	Transitions	8.29E-04	6.26E+00	5.24E-04	0.00E+00	6.26E-07
STC	Events	4.00E+00	8.00E+00	0.00E+00	0.00E+00	1.93E-07
	States	0.00E+00	3.00E+00	7.00E+00	2.00E+00	1.60E-07
	Transitions	0.00E+00	3.01E+00	1.50E+01	1.00E+01	4.13E-07
DDC	Events	9.15E-04	7.07E+00	5.03E-04	0.00E+00	9.23E-07
	States	8.95E-02	7.28E+00	9.92E-01	4.15E-08	1.07E-06
	Transitions	8.03E-09	2.17E-07	6.37E+00		
PHY	Events	4.24E-04	3.13E+00	2.63E-04	0.00E+00	1.57E-07
	States	3.85E-04	2.99E+00	4.49E-08	3.94E-07	4.10E-07
	Transitions	1.46E-09	3.88E-08	1.13E+00		
SYSTEM	Events	2.92E+00	1.01E+00	3.28E-05	8.13E-07	
	States	0.00E+00	0.00E+00	5.10E+01	1.30E+01	2.83E-06
	Transitions	0.00E+00	3.99E+01	9.38E+01	8.25E+01	1.29E+01

Table 4.4 Complexity Factors Using Standard GRG Solver

The asymptotic complexity of solving a single complexity equation, as demonstrated in this section, is equal to the complexity of the GRG algorithm. The complexity of the initialisation procedure is of $O(\mathrm{DS} \times n)$, which is negligible.

Using a computer with 8 GB RAM and an Intel Core *i7* running at 1.6 GHz, it took 0.45 of a second to run the GRG initialisation and solving procedure once. Running the procedure 15 times for producing all the results in Tables 4.3 and 4.4 took 4.59 seconds.

4.6.3 Modified Iterative GRG Algorithm

The generic GRG solving routine in Listing 4.1 was used for solving all the observed complexity metrics and equations. For this reason, the generic complexity formulae used by that routine was of order 5, which was deemed sufficiently large as explained earlier. However, it was noticed that for many of the solved equations the highest order multiplicand to the complexity equations produced by the GRG algorithm always had a very small and negligible value. This meant that the associated equation must be of lower order and that the *ChangeRange* to the GRG Solver in Listing 4.1 must be adjusted accordingly.

```
Sub RoundAndSolve()

  Dim iToRound  As Double

  For c = n + (FS - 1) To FS Step -1

    Call SolveOnce(Range(Cells(F, FS), Cells(F, c)))

    iToRound = ActiveSheet.Cells(F, c).Value
    Cells(F, c).Select
    ActiveCell.FormulaR1C1 = Round(iToRound, 0)

  Next c

End Sub
```

Listing 4.2 Iterative GRG Routine

It was also noticed that many of the produced multiplicands had a small fraction below or above an integer. This has resulted in similar fractions in the associated predicted metrics. Practically, this cannot be the case, since the *measured* metrics were always integers.

For these reasons, an iterative *RoundAndSolve* routine was implemented as illustrated in Listing 4.2. It uses the *SolveOnce* GRG routine defined in Listing 4.1 iteratively as follows: the highest order factor is rounded to integer value before another iteration of the GRG procedure was started. In the next iteration, the factor rounded in the previous iteration would be fixed and not allowed to be modified by the GRG algorithm. In the case of a highest order multiplicand with a very small value, this rounding operation would reduce it to zero and subsequently the order of the associated complexity equation is reduced.

The iterative operation is performed a number of times that is equal to the order of the analysed complexity equation (number of multiplicands, n in this case). All the multiplicands by the end of this procedure must have integer values.

4.6.3.1 Updated Complexity Results of the Case Study

Using the iterative GRG algorithm in Listing 4.2, the analysis of the complexity equations for all the processes modelled in Section 3.4 was repeated except this time with more decisive results.

Similar to Table 4.3, Table 4.5 shows the updated results: the metrics as obtained from FDR, the predicted metrics using the iterative procedure above and the error values squared. Table 4.5 also shows the final SS values for all the analysed complexity

equations, and this time the algorithm worked *perfectly* and all error values were zero. In practice, the modified algorithm was able to produce the same results for all the analysed processes using only four measured values of the metric. For this reason Table 4.5 displays results for up to DS = 4, in comparison with Table 4.3, which displayed the results for up to DS = 8.

From the practical experiments performed it was observed that the algorithm was able to produce the complexity factors of an equation of complexity $O(2^{3\times \mathrm{DS}})$, using a minimum of three data points. Additional data points did not affect the calculated complexity equation. Similarly, the algorithm was able to find a solution (optimising SS to zero) for an equation of complexity $O(2^{4\times \mathrm{DS}})$, using a minimum of four data points. An example of this are the transitions metrics of the SYSTEM process demonstrated in Table 4.6, which were the metrics with the highest complexity in this case study.

Another observation is the fact that the algorithm was able to produce precise complexity equations, since the underlying models were all monotonically and exponentially growing as DS grew. For precise results, it was also noticed that the growth should be uniform according to one exponential formula and there should be no special cases where the processes behaved differently for specific DS values or ranges.

Table 4.6 shows the multiplicands of all the complexity equations for all the analysed processes. One can now assess the complexity results with the confidence that it truly represents the underlying state machines of the CSP processes.

The final *State-Space Complexity* and *Communication-Space Complexity* formulae for selected processes (PHY, DDC and the top-level SYSTEM process) as extracted from Table 4.6 are shown in Eqns. 4.14 to 4.19.

$$\mathrm{DDC}_{SComplex}(\mathrm{DS}) = 2 + 7 \times 2^{\mathrm{DS}} + 1 \times 2^{2\times \mathrm{DS}} \tag{4.14}$$

$$\mathrm{DDC}_{CComplex}(\mathrm{DS}) = 1 + 11 \times 2^{\mathrm{DS}} + 6 \times 2^{2\times \mathrm{DS}} \tag{4.15}$$

$$\mathrm{PHY}_{SComplex}(\mathrm{DS}) = 3 \times 2^{\mathrm{DS}} \tag{4.16}$$

$$\mathrm{PHY}_{CComplex}(\mathrm{DS}) = 4 \times 2^{\mathrm{DS}} + 1 \times 2^{2\times \mathrm{DS}} \tag{4.17}$$

$$\mathrm{SYSTEM}_{SComplex}(\mathrm{DS}) = 51 \times 2^{2\times \mathrm{DS}} + 13 \times 2^{3\times \mathrm{DS}} \tag{4.18}$$

$$\mathrm{SYSTEM}_{CComplex}(\mathrm{DS}) = 113 \times 2^{2\times \mathrm{DS}} + 80 \times 2^{3\times \mathrm{DS}} + 13 \times 2^{4\times \mathrm{DS}} \tag{4.19}$$

The complexity of the iterative GRG algorithm described in Listing 4.2 is:

$$n \times Complexity(GRG) \tag{4.20}$$

Table 4.5 Predicted Metrics and Error Using Iterative GRG Solver

DataSize (DS)			1	2	3	4	SS
ISA	Events	Measured	12	20	36	68	
		Predicted	12	20	36	68	
		Error[2]	0	0	0	0	0
	States	Measured	13	17	25	41	
		Predicted	13	17	25	41	
		Error[2]	0	0	0	0	0
	Trans.	Measured	20	32	56	104	
		Predicted	20	32	56	104	
		Error[2]	0	0	0	0	0
STC	Events	Measured	20	36	68	132	
		Predicted	20	36	68	132	
		Error[2]	0	0	0	0	0
	States	Measured	50	252	1496	10032	
		Predicted	50	252	1496	10032	
		Error[2]	0	0	0	0	0
	Trans.	Measured	146	892	6104	44848	
		Predicted	146	892	6104	44848	
		Error[2]	0	0	0	0	0
DDC	Events	Measured	17	31	59	115	
		Predicted	17	31	59	115	
		Error[2]	0	0	0	0	0
	States	Measured	20	46	122	370	
		Predicted	20	46	122	370	
		Error[2]	0	0	0	0	0
	Trans.	Measured	47	141	473	1713	
		Predicted	47	141	473	1713	
		Error[2]	0	0	0	0	0

Table 4.5 Continued ↓

DataSize (DS)			1	2	3	4	SS
PHY	Events	Measured	10	16	28	52	
		Predicted	10	16	28	52	
		Error^2	0	0	0	0	0
	States	Measured	6	12	24	48	
		Predicted	6	12	24	48	
		Error^2	0	0	0	0	0
	Trans.	Measured	12	32	96	320	
		Predicted	12	32	96	320	
		Error^2	0	0	0	0	0
SYSTEM	Events	Measured	5	7	11	19	
		Predicted	5	7	11	19	
		Error^2	0	0	0	0	0
	States	Measured	308	1648	9920	66304	
		Predicted	308	1648	9920	66304	
		Error^2	0	0	0	0	0
	Trans.	Measured	1300	10256	101440	1208576	
		Predicted	1300	10256	101440	1208576	
		Error^2	0	0	0	0	0

Table 4.5 Predicted Metrics and Error Using Iterative GRG Solver

where n is the number of multiplicands or factors in the complexity equation. The complexity of the initialisation procedure is still of $O(\mathrm{DS} \times n)$, which has negligible asymptotic effects.

Setting $n = 5$ and using the same computer as in the previous section (8 GB RAM and an Intel Core $i7$ running at 1.6 GHz), it took 1.59 seconds to run the iterative GRG routine once. Running the routine 15 times for producing all the results in Tables 4.5 and 4.6 took 21.84 seconds.

4.7 Evaluation

It is now possible to perform an abstract complexity review of both the building blocks as well as the top-level system using the respective complexity formulae. These formulae and their asymptotic behaviour allow the reflection on the complexity semantics of different CSP operators and their effect on the complexity of the system. Moreover,

Multiplicand		1	2^{DS}	$2^{2\times DS}$	$2^{3\times DS}$	$2^{4\times DS}$
ISA	Events	4	4			
	States	9	2			
	Transitions	8	6			
STC	Events	4	8			
	States	0	3	7	2	
	Transitions	0	3	15	10	
DDC	Events	3	7			
	States	2	7	1		
	Transitions	1	11	6		
PHY	Events	4	3			
	States	0	3			
	Transitions	0	4	1		
SYSTEM	Events	3	1			
	States	0	0	51	13	
	Transitions	0	0	113	80	13

Table 4.6 Complexity Factors Using Iterative GRG Solver

it is also possible to perform targeted optimisations to reduce the complexity of the associated hardware models.

The asymptotic behaviour of the building blocks as well as the top-level SYSTEM process is presented in Table 4.7. The table shows that the *Communication-Space Complexity* is not necessarily the same as the *State-Space Complexity*; the PHY process is an example of this observation.

Process	ISA	STC	DDC	PHY	SYSTEM
$SComplex(\mathrm{DS}) \in$	$O(2^{1\times DS})$	$O(2^{3\times DS})$	$O(2^{2\times DS})$	$O(2^{1\times DS})$	$O(2^{3\times DS})$
$CComplex(\mathrm{DS}) \in$	$O(2^{1\times DS})$	$O(2^{3\times DS})$	$O(2^{2\times DS})$	$O(2^{2\times DS})$	$O(2^{4\times DS})$

Table 4.7 Asymptotic Aspect of the Complexity Formulae

The analysis of the CSP specifications in light of the above complexity formulae reveals some interesting observations about the semantics of some CSP operators with respect to the associated state machine complexity. For example, the STC process had at most three state variables of *DataWord* type. This is expected to be the reason behind its asymptotic complexity (both *SComplex* and *CComplex*) being $O(2^{3\times n})$.

Both Table 4.7 and Eqns. 4.16 to 4.17 show that even though the associated CSP model of the PHY process is using a *Boolean* subset of the *DataWord* set for basic data

communication, the resultant state machine is still dependent on the whole *DataWord* set and its associated DS variable. This is predicted to have a negative effect on the complexity of the DDC and STC processes as well. For example, Eqns. 4.14 to 4.15 show that the DDC process is even more dependent on DS than the PHY process, even though this is not obvious from the CSP description.

Finally, Eqns. 4.18 to 4.19 show precisely how the top-level SYSTEM process will grow as the value of DS grows. The equations show that the SYSTEM will grow to about 2×10^8 states and 6×10^{10} transitions using only 8 bit data-type. This is both informative and alarming, taking into account the minimal set of modelled functions.

The complexity analysis of the modules was very useful. It exposed the overhead of using a single and global data-type to communicate time, control and data information. This is evident by the strong dependence of each process analysed on that data-type.

4.7.1 Complexity-Based Targeted Optimisation

Now let us look closely at the PHY process in Eqn. 3.61. Its communication complexity in Table 4.7 is $O(2^{2\times \mathrm{DS}})$, while its predicted complexity is $O(1)$ and it should be totally independent of the DS variable. Its state variable (*opValue*) stores the state of the port and must be limited to *Boolean* values since it is only changed in Eqn. 3.63. However, by looking at Figure 3.6 closely, it can be seen that all values from the set *DataWord* are being used to update the state variable *opValue* through the statement:

$$\Box \, \forall \, npv \in Boolean \Rightarrow rPHY.\mathrm{B}_{raw}.\mathrm{C}_{data}?npv \rightarrow \mathrm{PHY}\prime(npv) \tag{4.21}$$

This means that the condition $\forall \, npv \in Boolean$ has no effect on limiting the communications on the associated $rPHY.\mathrm{B}_{raw}.\mathrm{C}_{data}$ channel and the "?" input operator still allowed the full *DataWord* set to be input.

By changing the "?" input operator to the standard "." ("dot") operator in the input statement (Eqn. 3.63), the intended behaviour is observed and communications on the $rPHY.\mathrm{B}_{raw}.\mathrm{C}_{data}$ channel are now limited to the *Boolean* subset of the *DataWord* set as follows:

$$\Box \, \forall \, npv \in Boolean \Rightarrow rPHY.\mathrm{B}_{raw}.\mathrm{C}_{data}.npv \rightarrow \mathrm{PHY}\prime(npv) \tag{4.22}$$

This observed behaviour of the "?" input operator is thought to be an issue with FDR and its compilation process, rather than with the CSP description of Eqn. 3.63.

Table 4.8 shows the updated measured metrics only. This is because, similar to the results in Table 4.5, the metrics are identical to the predicted metrics calculated by the iterative GRG algorithm and all error values were zero.

DataSize (DS)		1	2	3	4
ISA	Events	12	20	36	68
	States	13	17	25	41
	Transitions	20	32	56	104
STC	Events	20	36	68	132
	States	50	252	1496	10032
	Transitions	146	892	6104	44848
DDC	Events	17	31	59	115
	States	20	46	122	370
	Transitions	47	141	473	1713
PHY	Events	10	10	10	10
	States	6	6	6	6
	Transitions	12	12	12	12
SYSTEM	Events	5	5	5	5
	States	308	776	2192	6944
	Transitions	1300	3352	9712	31456

Table 4.8 Metrics After Complexity Targeted Optimisations

Table 4.9 shows the updated multiplicands to all the complexity equations and Table 4.10 shows the updated asymptotic behaviour of all the processes.

As evident from both Tables 4.9 and 4.10, a small change, such as the selection of two seemingly equivalent CSP operators, has reduced the complexity of the PHY process to $O(1)$. This in turn had the desired effect on the complexity of the top-level SYSTEM. Its state and communication complexities have now been reduced to $O(2^{2\times \mathrm{DS}})$.

The examples and results presented in this section are only intended to demonstrate the usefulness of applying innovative complexity analysis to CSP processes and associated state machines. The procedure has been applied to various other bottlenecks in the system. For example, similar to the PHY process, the DDC process as demonstrated in Eqn. 3.55 was also expected to be independent of the DS variable. Also, the STC process has an unexpectedly large complexity which is even greater than the complexity of the top-level SYSTEM, as can be seen in Table 4.9. This issue has been addressed through fine-tuning the communication sets on the *lDDC*,

Multiplicand		1	2^{DS}	$2^{2\times DS}$	$2^{3\times DS}$
ISA	Events	4	4		
	States	9	2		
	Transitions	8	6		
STC	Events	4	8		
	States	0	3	7	2
	Transitions	0	3	15	10
DDC	Events	3	7		
	States	2	7	1	
	Transitions	1	11	6	
PHY	Events	10			
	States	6			
	Transitions	12			
SYSTEM	Events	5			
	States	0	114	20	
	Transitions	0	462	94	

Table 4.9 Factors After Complexity Targeted Optimisations

Process	ISA	STC	DDC	PHY	SYSTEM
$SComplex(\text{DS}) \in$	$O(2^{1\times DS})$	$O(2^{3\times DS})$	$O(2^{2\times DS})$	$O(1)$	$O(2^{2\times DS})$
$CComplex(\text{DS}) \in$	$O(2^{1\times DS})$	$O(2^{3\times DS})$	$O(2^{2\times DS})$	$O(1)$	$O(2^{2\times DS})$

Table 4.10 Asymptotic Aspect of the Optimised Complexity Formulae

rDDC, *lSTC* and *rSTC* channels. The details of this step were omitted for brevity since they did not affect the complexity of the top-level SYSTEM process.

It has been observed that the complexity of the top-level SYSTEM process is not necessarily larger than the complexity of its most complex building block as evident from Table 4.7: the complexity of the STC process is $O(2^{3\times DS})$, while the complexity of the SYSTEM process is $O(2^{2\times DS})$.

Table 4.11 demonstrates the improvements to the metrics of the top-level system process. The improvements increase drastically as DS increases. An improvement of 3842% for the transitions metrics when DS = 4 is indeed a significant improvement.

The improvements shown in Table 4.11 were further analysed by relating them mathematically to DS using the iterative GRG algorithm described earlier. Eqn. 4.23 shows that the improvements in the number of transitions are increasing exponentially as the data size increases.

Data Size	SYSTEM before		SYSTEM after		Improvements(%)	
	States	Transitions	States	Transitions	States	Transitions
1	308	1300	308	1300	0	0
2	1648	10256	776	3352	212	305
3	9920	101440	2192	9712	452	1044
4	66304	1208576	6944	31456	954	3842

Table 4.11 Improvements to Metrics of Optimised SYSTEM

$$TransitionsImprovement_{\%}(\mathrm{DS}) = 16 \times 2^{\mathrm{DS}} + 14 \times 2^{2\times \mathrm{DS}} \tag{4.23}$$

This fact can also be evident in the change to the communication complexity of the SYSTEM process, which was of $O(2^{2\times \mathrm{DS}})$. This has great impact on the scalability of the system.

Let us now assume that the maximum number of transitions that FDR can currently handle is 5×10^8 transitions. This is a realistic assumption, since the largest observed process during the development of this framework had about 3.2×10^8 transitions as shown in Table 4.2, which needed about 9 hours to verify. Figure 4.3 demonstrates this aforementioned assumption along with the expected growth in the number of transitions for the SYSTEM process before and after optimisation. The demonstrated growth is according to the developed complexity formulae in Tables 4.6 and 4.9.

Figure 4.3 shows that FDR is able to verify an unoptimised version of the SYSTEM with a maximum DS value of 6. This figure almost doubles to 11 after optimisation. The figures discussed here are for illustration purposes, since the ultimate goal from performing these complexity analyses is to improve the scalability of the system and address any complexity overhead added by the modelling language, the architecture of the modelled system or the verification tools.

4.8 Summary and Conclusions

The conclusions of this chapter are twofold: first, the development of objective complexity analysis techniques for analysing the complexity of various modelling approaches. Second, and through the use of the case study, this chapter provides great insight on the complexity of the multiplexing approach that was presented in Chapter 3.

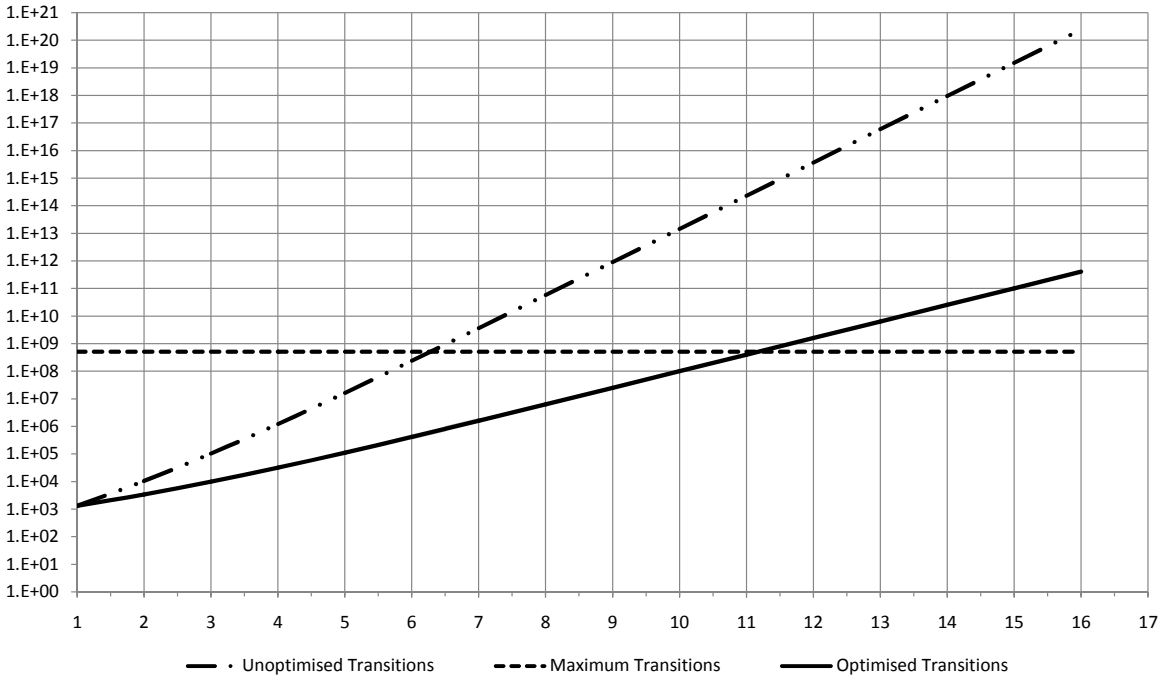


Figure 4.3 Transitions vs. *DataSize* for the SYSTEM Process

4.8.1 Complexity Analysis Conclusions

The techniques described in this chapter enable the complexity of state machines and formal models to be analysed in terms of configuration parameters, state variables, or data-types. These are highly valuable techniques in the design of efficient hardware architectures, which includes the design of functionally independent hardware blocks. It is also essential for managing the space explosion problem.

By using the right analysis tools, it may be possible to transform a non-optimal design with unrealistic complexity explicitly to a much simpler design with more optimal state space. Furthermore, by establishing properties and checks about the optimal design, similar properties can be concluded about the non-optimal one. For example, if the non-optimal design was $\in O(\mathrm{S}^{10})$ and the optimal one was $\in O(\mathrm{S}^{5})$, one can achieve the verification of the non-optimal design by verifying the optimal one.

Finally, as it turned out, analysing the complexity of the compiled state machines is a useful mechanism for objective analysis of the modelling and verification formalism itself and any associated compilation or compression algorithms.

4.8.2 Multiplexing Approach Conclusions

Through the complexity analysis of the case study discussed in Section 4.6.2.1, the following observations were made.

1. Functional blocks in the pipeline have to deal with control and data information not addressed to them, in which case they forward them onto the next block in the pipeline.

2. Through the use of the proposed complexity analysis techniques, optimisations were possible to the complexity of various blocks by introducing the *Boolean* data-type for specific instances of data and control communications.

3. The design approach involves analysing which tokens are produced, consumed or propagated by each functional block for accurate and optimal system specification. This process was laborious and time consuming.

4. The enforcement of all functional blocks to use the same generic interface introduces unnecessary latencies to control and data information.

5. The hardware implementation implications are also great: large multiplexers and token passing controllers have to be implemented and customised for each functional block. Even with the most optimal design and implementation, the resulting hardware is still expected to be very inefficient.

The above observations lead to the conclusion that the data and control multiplexing architecture adds significant modelling, verification and possibly implementation overhead and hence a different architecture for modelling the communication system was needed. For these reasons, Chapter 5 introduces a more dynamic approach for modelling the communication system, where various styles of control and data communications are first considered. Subsequently, the concrete configurable communication system is demonstrated.

4.9 Future Work

Future work includes a comprehensive analysis of complexity semantics of both CSP and the associated machine-readable interface for FDR. This should result in modifications to the existing constructs and interfaces or the provision of new ones that are optimised from a complexity perspective.

A static complexity analyser of CSP models producing complexity formulae and asymptotic analysis of the models would be a useful tool for managing complexity and designing optimal hardware architectures. It could also serve as another level of formal verification, where the intended behaviour of the models as captured by the modelling language and the verification tools conforms to the intended complexity model.

Finally, this chapter presents the analysis of a communication system with a handful of state variables, which are at most shared between 5 processes. Roscoe [Chapter 18, 38], on the other hand, discusses *Shared-Variable Programs* in which variables are shared between multiple threads in a parallel system. Those programs would benefit from an empirical analysis similar to that which has been presented in this chapter. Such an analysis would show how dependent the programs are on the shared variables and how realistic the proposed model-checking solution is.

CHAPTER 5

Hierarchically Controlled Concrete Configurable Communication System

5.1 Motivation and Chapter Structure

The approach taken for providing configurability of the hardware blocks is based on an ISA control system [70]. A central process has control over the system building blocks using simple instructions for conveying control information, as well as for transferring data to and from the control system. This chapter explores further design options for modelling the configurable communication system.

For modelling the Concrete Configurable Communication System (C^3S), the technique of separating functional requirements into functionally independent configurable hardware blocks was employed. Though this technique was discussed in detail in Chapter 2 and first modelled in Chapter 3, it was fine-tuned here through the complexity analysis technique presented in Chapter 4 in order to reduce the verification efforts.

First, Section 5.2 discusses how the data and control multiplexing presented in Chapter 3 was further optimised through many refinements to result in the hierarchically controlled system. A formal description of the configurable blocks, as well as how those blocks could be used to construct the top-level system is demonstrated in Section 5.3 along with the ISA interface. Section 5.4 discusses the use of the configurable building blocks in exploring the construction of a system that meets the needs of the targeted communication protocols. This is done through statically instantiating and connecting the desired configurable blocks synchronising on the appropriate control and data channels.

Figure 5.1 highlights the models discussed in Sections 5.3 and 5.4 and their position in the design and verification of the configurable communication system.

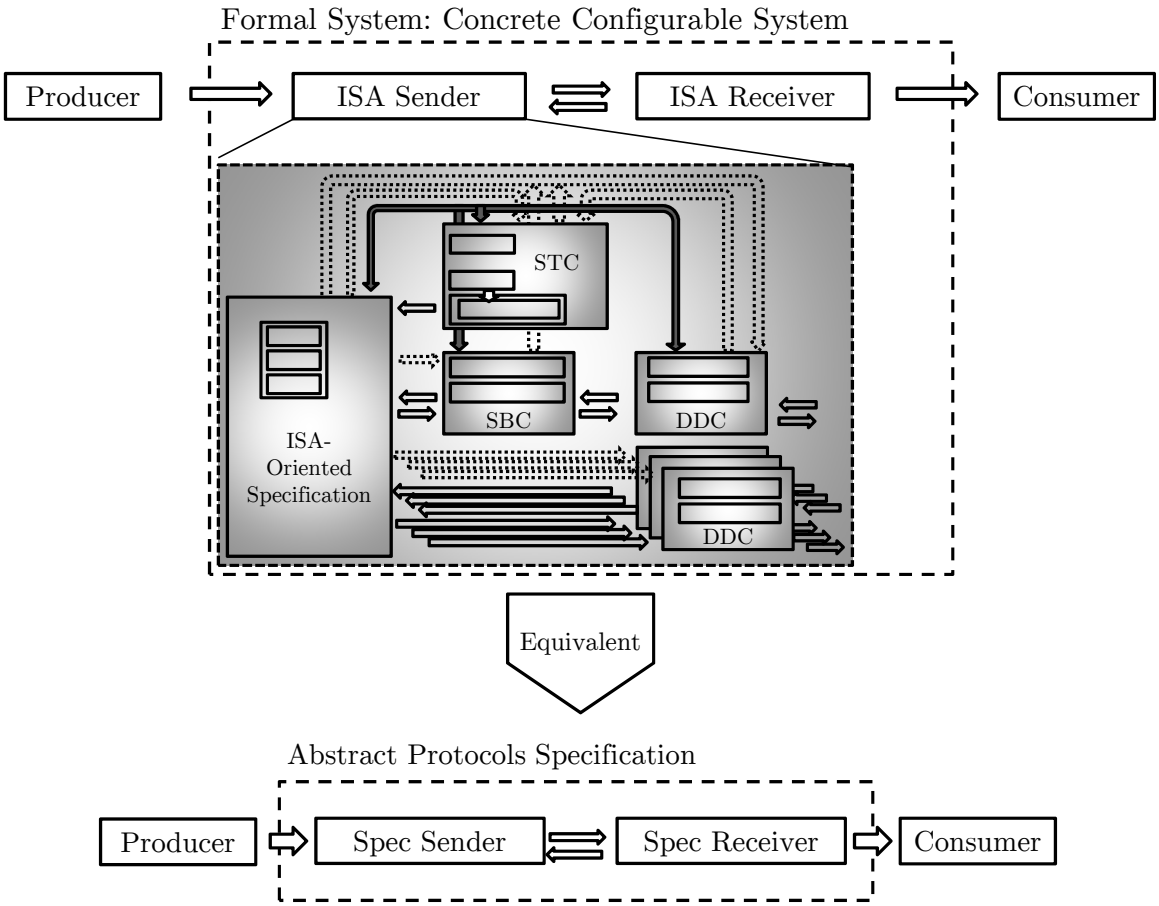


Figure 5.1 Outline of the Concrete Configurable Communication System

Finally, Section 5.5 presents the complexity analysis of the system and its components through identifying the statistical metrics and solving the corresponding complexity formulae. Asymptotic analysis of the complexity formulae and the effect of different configuration parameters on such formulae is also discussed. The chapter concludes with the summary and future work in Sections 5.6 and 5.7, respectively.

As mentioned already in Section 3.3, the notation used for describing the CSP models in this thesis uses a combination of the Symbol Macros for CSP described by Mazur [40], as well as standard algebraic notation to achieve best readability. The full machine-readable CSP (CSP_M) scripts of all the models discussed in this chapter are available online at: `www.cs.bris.ac.uk/~kharmeh/thesis/CSP.tar`. Scattergood and Armstrong [71] provide a summary of the CSP_M interface used throughout those scripts.

5.2 Hierarchical Control of the Functional Blocks

Because the data and control multiplexing approach described in Chapter 3 was proving too complex and cumbersome to design and verify, other approaches were explored until a final hierarchical approach was considered most practical.

First, a hybrid model that combines central and distributed control mechanisms of the functional blocks was considered. This has evolved through the consideration of two different architectural approaches.

1. Central Control: a single process modelling the instruction set of an embedded system has direct control channels to each functional block in the system.

2. Distributed Control: each functional block has direct control over subsequent functional blocks in the pipeline.

The reasoning behind the two control considerations is discussed briefly in the following sections.

5.2.1 Central Control

This was required in order to deliver control commands directly from the ISA process to control blocks further down the pipeline of configurable blocks. This is usually needed because the ISA is responsible for configuring the functional units in the system and for initiating any I/O operation. The introduced control channels are demonstrated in Figure 5.2.

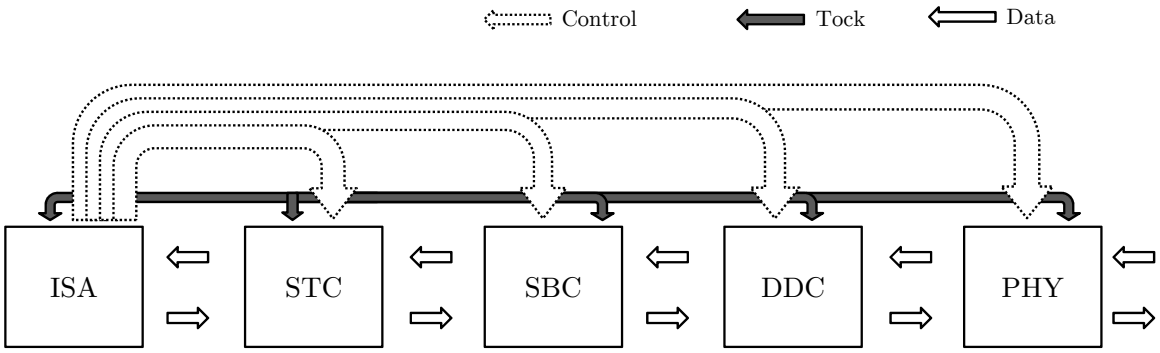


Figure 5.2 Central Control of Configurable Units

Section 2.5.5 considered the compositional semantics of the functional blocks. In order for the architecture demonstrated in Figure 5.2 to effectively address commands and configurations composed of the interaction of more than one functional block, the system needed further refinement.

Let us consider the compositional semantics of the STC block and the PHY block when executing a *Timed Input* operation. While the STC block is responsible for the correct timing of the executed operation, the actual data input from the environment is provided by the PHY block. Using the system in Figure 5.2, this operation can only be achieved through a *2-phase* transaction centrally controlled by the ISA process as follows: the first phase starts by the ISA issuing a timing command to the STC and is complete when the correct time has elapsed and the STC block notifies the ISA of such completion. The ISA then issues the second phase of the transaction, which resembles a raw input operation performed by the PHY block.

This has many drawbacks including efficiency, atomicity and timing correctness of such transactions. A better approach for modelling compositional transactions is demonstrated in Figure 5.3.

5.2.2 Distributed Control

This was required to enable the modelling of operations involving multiple configurable blocks as an atomic operation from the ISA point of view. It offers direct control channels from functional blocks to subsequent functional blocks in the pipeline, as demonstrated in Figure 5.3.

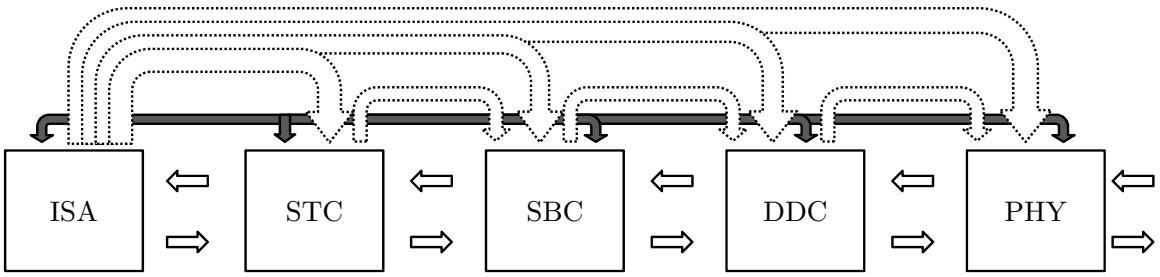


Figure 5.3 Additional Distributed Control

The ISA process issues the commands directly to the first block in the composition chain, which in turn can complete the operation and respond back to the ISA. Alternatively, such a block can issue further commands to the next block in the composition chain.

In the case of *Timed Input* for example, the STC block can wait until the time of the input is met. Then it triggers the input to the PHY block, which can then send the data back to the ISA block.

Hypothetically, there is a scenario in which commands have to be propagated along a chain if the source block and the destination block are not directly connected.

The brute force alternative would be a large matrix of communication channels, where each process has direct links to all other processes in the system.

5.2.3 Hierarchical Control

This was considered the most tractable approach. It presents a more dynamic mechanism for connecting the configurable blocks where the two connectivity abstractions mentioned earlier (*Central Control* and *Distributed Control*) are taken into account. In addition, the observations made in Section 2.5 argue that more than one requirement could have functional dependencies and hence could be better addressed in a hierarchically constructed system: processes that address similar or dependent functional aspects are grouped together into one larger block with a single interface to the ISA process. For example, a register file is an integral part of the ISA process and no other process in the system should have access to it. In Section 5.3, a set of functional requirements are selected and modelled in their respective functional blocks. The communication channels connecting the different blocks demonstrated later in Figure 5.4 were developed through employing the different connectivity mechanisms discussed.

5.3 The Concrete Configurable Communication System

In this section, a concrete formal definition of the modelled functional blocks is presented. The CSP models of each block are discussed briefly, as well as the overall system. At this point, the concrete system has the following configurable blocks: STC, SBC and DDC. The direction of data flow is modelled as a configuration option in the relevant blocks and not as a separate configurable block, as discussed in Section 2.5. The state of the physical interface (denoted by the PHY block in Figure 5.3) is integrated into the DDC functional block for implementation efficiency considerations. Also modelled is an ISA controlling process to allow for the configuration and communication with the functional blocks.

5.3.1 Preliminaries

First, preliminary functions, definitions and CSP constructs are defined. These are shared between both the configurable communication system modelled in this chapter and the communication protocols modelled in Chapter 6.

Those are defined as follows:

$$inc(a, limit) = (a + 1) \bmod limit \tag{5.1}$$

$$add(a, b, limit) = (a + b) \bmod limit \tag{5.2}$$

$$sright(word, bit) = \begin{cases} (word/2) + 2^{\mathrm{DS}-1} & \text{if } bit = 1 \\ (word/2) & \text{otherwise} \end{cases} \tag{5.3}$$

$$boolean = Data = \{0, 1\} \tag{5.4}$$

$$DataSize = \mathrm{DS} = 2 \tag{5.5}$$

$$DataWord = \mathrm{DW} = \{0 \,..\, (2^{\mathrm{DS}} - 1)\} \tag{5.6}$$

$$TimeSize = \mathrm{TS} = 3 \tag{5.7}$$

$$TimeWord = \mathrm{TW} = \{0 \,..\, (2^{\mathrm{TS}} - 1)\} \tag{5.8}$$

$$channel \quad produce, consume : DataWord \tag{5.9}$$

$$channel \quad tock \tag{5.10}$$

where:

- *inc(a, limit)*: increments a by 1 and finds the modulo of the result by *limit*;
- *add(a, b, limit)*: adds a and b then finds the modulo of the result by *limit*;
- *sright(word, bit)*: shifts *bit* into *word* from the right;
- *boolean*: defines the basic communication data-type;
- *DataSize*: or DS is the system internal data word size which is used by some registers and variables. It is also the unit of communication for the modelled protocols in Chapter 6;
- *DataWord*: or DW is the communication system internal data word unit which is *DataSize* bits wide;
- *TimeSize*: or TS is the size of the time data-type;
- *TimeWord*: or TW is the time data-type;
- *produce*: is a virtual generator of data that can output *DataWord*s infinitely;
- *consume*: is a virtual consumer of data that can input *DataWord*s infinitely; and
- *tock*: is an event used to represent the passage of time.

The *produce* and *consume* are virtual channels that are mainly used for functional verification of the communication protocols. For the modelling part, they are assumed to be an infinite source/sink of data items to be transmitted/received.

In addition, the following process definitions are made available:

$$\text{TOCK}(0) = \text{SKIP} \tag{5.11}$$

$$\text{TOCK}(n) = tock \rightarrow \text{TOCK}(n-1) \tag{5.12}$$

The TOCK process in Eqns. 5.11 and 5.12 is used to model the passage of n clock cycles by a timed process.

In the CSP specification that follows, all blocks are assumed to have an arbitrary number of control channels as per their functional requirements. The data interface is modelled as two separate channels (one for input and one for output) on each side of the configurable block. For example, the DDC block has two channels that connects it to the environment (*ddcLeftIn* and *ddcLeftOut*) and two channels that connects it to other internal blocks (*ddcRightIn* and *ddcRightOut*).

5.3.2 Data Dependent Control (DDC)

Let us now consider a data dependent input performed in the DDC block and how this could be implemented in the presence of a PHY process. When the ISA block initiates a data dependent input (i.e. an input operation which does not complete until a specific value appears on the physical interface), the DDC process can execute it using two possible approaches.

1. Polling: the DDC process keeps polling the PHY every clock cycle for input until the right value is observed at which time it notifies the ISA.

2. Event based: the PHY could interrupt the DDC when the correct value is observed, which is more efficient than polling.

Implementing the second alternative makes the DDC block redundant and hence it may be bypassed, in which case the response can go directly to the ISA process. This means that the PHY block is now implementing data dependent I/O operations. For this reason, the PHY process functions including keeping track of the physical state of the communication bus were integrated into the DDC process. This process also modelled *raw* input and output operations.

The physical interface, on the other hand, is modelled as signal change events received and sent by the DDC block. The state of the physical interface is kept internally within this block. A signal change event might be sent by this module as a result of an output operation. Alternatively, it can be received as a result of an external change to the physical interface triggered by another entity connected to the bus.

The DDC block is defined as follows:

$$\mathrm{DDC} = \mathrm{DDC}\prime(1) \tag{5.13}$$

$$\mathrm{DDC}\prime(data) = tock \rightarrow \mathrm{DDC}\prime(data) \tag{5.14}$$

$$\Box\ ddcRightIn?new \rightarrow \tag{5.15}$$

$$(Internal(new)\ \Box\ tock \rightarrow \mathrm{DDC}\prime(new)) \tag{5.16}$$

$$\Box\ Internal(data) \tag{5.17}$$

$$Internal(data) = ddcCtrl.\,\mathrm{RAWO} \rightarrow \mathrm{DDC}_{O2}(data) \tag{5.18}$$

$$\Box\ ddcCtrl.\,\mathrm{COND} \rightarrow \mathrm{DDC}_{I2}(data) \tag{5.19}$$

$$\Box\ ddcCtrl?\,\mathrm{RAWI} \rightarrow \mathrm{DDC}_{\mathrm{IFINISH}}(data) \tag{5.20}$$

$$\mathrm{DDC}_{O2}(data) = stcCtrlDdc!\,\mathrm{STMP} \rightarrow ddcLeftIn?new \rightarrow \tag{5.21}$$

$$\begin{cases} ddcRightOut!new \rightarrow tock \rightarrow \mathrm{DDC}\prime(new) & \text{if new}\neq\text{data} \\ tock \rightarrow \mathrm{DDC}\prime(data) & \text{otherwise} \end{cases} \tag{5.22}$$

$$\mathrm{DDC}_{I2}(data) = ddcLeftIn?cond \rightarrow \tag{5.23}$$

$$\begin{cases} \mathrm{DDC}_{\mathrm{IFINISH}}(data) & \text{if data=cond} \\ \mathrm{DDC}_{I3} & \text{otherwise} \end{cases} \tag{5.24}$$

$$\mathrm{DDC}_{I3}(cond) = tock \rightarrow \mathrm{DDC}_{I3}(cond) \tag{5.25}$$

$$\Box\ ddcRightIn?new \rightarrow \tag{5.26}$$

$$\begin{cases} \mathrm{DDC}_{\mathrm{IFINISH}}(new) & \text{if new=cond} \\ tock \rightarrow \mathrm{DDC}_{I3}(cond) & \text{otherwise} \end{cases} \tag{5.27}$$

$$\mathrm{DDC}_{\mathrm{IFINISH}}(data) = stcCtrlDdc!\,\mathrm{STMP} \rightarrow ddcLeftOut!data \rightarrow \tag{5.28}$$

$$tock \rightarrow \mathrm{DDC}\prime(data) \tag{5.29}$$

The following is a brief description of the functions provided by the DDC block. A top-level process ($\mathrm{DDC}\prime$) is defined to respond to interaction commands with the

outside world, I/O commands from the ISA block, as well as I/O commands from the STC block. Commands from ISA and also from STC are all triggered on the same control channel (*ddcCtrl*).

- RAWI: Raw input command, which the DDC block can receive from the STC block, once the correct time of the operation has elapsed. RAWI can also be initiated from the ISA block. The DDC block responds to the ISA with the physical state of the interface.
- RAWO: Raw output command which, depending on the physical state of the interface that is recorded in this module, might trigger an external event that is visible to the environment.
- COND: Conditional input command (or data dependent I/O) in which the ISA block requires notification when the value of the physical interface equals the value specified by the ISA process. The DDC block would only respond to ISA once the value of the physical interface matches the requested value.

The DDC block needs to execute and synchronise on the timing event (*tock*) because it is responsible for triggering the timestamps at the exact clock cycle an operation has occurred. Hence it was modelled using tock-CSP similar to all other blocks: all commands and responses need to be valid from a timing point of view and the top-level processes need to conform to timing consistency checks, as will be discussed in Chapter 7. In addition to the standard timing consistency checks, logical timing consistency refers to the temporal relation between different real-time commands. For these reasons, process DDC′ in Eqn. 5.14 offers the *internal* choice twice.

The DDC process is meant to register the state of the physical interface and act like a latch for when the value of the external interface changes. For this reason, a RAWI command presents this registered value to the ISA interface. In the case where the physical interface changes its value in the same cycle the ISA requests an input, the DDC responds with the newly registered value.

On the assumption that the value registered in the DDC block is the actual value of the physical interface, an output operation would only result in a change event when the output value is different from the currently registered value of the interface. This is specified in Eqn. 5.22.

Internally, a DDC unit is accessible through three channels: a control channel (*ddcCtrl*), and data input and output channels (*ddcRightIn* and *ddcRightOut* respectively). Those could be defined using the following sequence:

$$ddcPort = \langle ddcCtrl, ddcRightIn, ddcRightOut \rangle \tag{5.30}$$

In FDR, $head(seq)$ returns the first item in a sequence, while $tail(seq)$ removes the first item in the sequence and returns the resulting new sequence. Hence, to access the different channels of a DDC unit, the following definitions are found useful:

$$ctrl(ddcPort) = head(ddcPort) \tag{5.31}$$
$$dIn(ddcPort) = head(tail(ddcPort)) \tag{5.32}$$
$$dOut(ddcPort) = head(tail(tail(ddcPort))) \tag{5.33}$$

5.3.3 Synchronisation and Timing Control (STC)

This functional block is responsible for keeping track of time by keeping count of the total number of clock cycles elapsed. This count has a resolution which is specified by the *TimeSize* constant defined in Eqn. 5.7. Once this count reaches its limit, it resets to zero and starts counting up again.

Semantic analysis of the STC block revealed that none of its functions actually involve manipulating the data stream. Its presence in the data pipeline, as depicted in Figure 5.3 is redundant since it is merely forwarding data forward and backward. For this reason it was removed from the data pipeline as shown in Figure 5.4.

In addition to its ability to control the timing of I/O operations, it has the ability to register the time stamp of each I/O operation, which enables the ISA to control subsequent timing operations. Timestamps are registered in a separate subprocess called STC_{stamp}, which is internal to the STC process. The introduction of such an internal process was necessary due to limitations in the FDR compiler, which restricts the total number of states a single sequential process can have: if the time stamp of the previous operation was kept as an additional state variable to the STC process, in addition to the current time and the target time of the next operation, then the asymptotic *State-Space Complexity* of a single sequential process (STC_{check} in Eqn. 5.41) would be $O(2^{3\times \text{TS}})$. This would break the compilation process in FDR.

The STC block is defined as follows:

$$\mathrm{STC}_{isync} = \{\!|stcCtrlDdc.\,\mathrm{STMP}, tLocal, tock|\!\} \tag{5.34}$$

$$\mathrm{STC} = (\mathrm{STC}\prime(0) \underset{\mathrm{STC}_{isync}}{\|} \mathrm{STC}_{stamp}(0)) \setminus \{\!|tLocal|\!\} \tag{5.35}$$

$$\mathrm{STC}\prime(oTm) = \mathrm{STC}_{tock}(oTm) \tag{5.36}$$

$$\Box\ stcCtrl?\,\mathrm{TMDI}\,.tm \rightarrow \mathrm{STC}_{check}(oTm, tm, \mathrm{TMDI}) \tag{5.37}$$

$$\Box\ stcCtrl?\,\mathrm{TMDO}\,.tm \rightarrow \mathrm{STC}_{check}(oTm, tm, \mathrm{TMDO}) \tag{5.38}$$

$$\Box\ stcCtrlDdc?\,\mathrm{STMP} \rightarrow tLocal!oTm \rightarrow \mathrm{STC}_{tock}(oTm) \tag{5.39}$$

$$\mathrm{STC}_{tock}(oTm) = tock \rightarrow \mathrm{STC}\prime(inc(oTm, 2^{\mathrm{TS}})) \tag{5.40}$$

$$\mathrm{STC}_{check}(oTm, tm, io) = \tag{5.41}$$

$$\begin{cases} \mathrm{STC}_{command}(oTm, io) & \text{if tm=oTm} \\ tock \rightarrow \mathrm{STC}_{check}(inc(oTm, 2^{\mathrm{TS}}), tm, io) & \text{otherwise} \end{cases} \tag{5.42}$$

$$\mathrm{STC}_{command}(oTm, io) = \tag{5.43}$$

$$\begin{cases} ddcCtrlStc!\,\mathrm{RAWI} \rightarrow \mathrm{STC}_{finish}(oTm) & \text{if io=TMDI} \\ ddcCtrlStc!\,\mathrm{RAWO} \rightarrow \mathrm{STC}_{finish}(oTm) & \text{otherwise} \end{cases} \tag{5.44}$$

$$\mathrm{STC}_{finish}(oTm) = \mathrm{STC}_{tock}(oTm) \tag{5.45}$$

$$\Box\ stcCtrlDdc?\,\mathrm{STMP} \rightarrow tLocal!oTm \rightarrow \mathrm{STC}_{tock}(oTm) \tag{5.46}$$

$$\mathrm{STC}_{stamp}(sTime) = tock \rightarrow \mathrm{STC}_{stamp}(sTime) \tag{5.47}$$

$$\Box\ stcCtrlDdc?\,\mathrm{STMP} \rightarrow tLocal?new \rightarrow \mathrm{STC}_{stamp}(new) \tag{5.48}$$

$$\Box\ stcTime!sTime \rightarrow tock \rightarrow \mathrm{STC}_{stamp}(sTime) \tag{5.49}$$

Hence, the STC responds to the following commands.

- TMDI: timed input (or time dependent input) command instantiated by the ISA block. The STC block compares the required operation time with the current time at each cycle in Eqn. 5.41. Once the operation time is reached, it signals the DDC block with a RAWI command in Eqn. 5.43.
- TMDO: timed output (or time dependent output) command, which is similar to the TMDI except that once the specified time has been met a RAWO command is signalled to the DDC block.
- STMP: timestamping command, which is received from DDC. Its response is internal to the STC, which involves registering a copy of the current time counter in the STC_{stamp} process in Eqn. 5.47.

- Timestamp read command in which the ISA reads the last timestamp through the *stcTime* channel.

It is essential for the DDC block to request the timestamping at the exact clock cycle an I/O operation completes. It is also essential that ISA instructions happen in a timely manner for the whole system to meet protocol deadlines. Special attention was needed to avoid race conditions and time lags between different system components. The absence of such issues was established through system-level verification, as described in Chapter 7.

5.3.4 Serialisation and Buffering Control (SBC)

This section defines the SBC block. It supports bidirectional serialisation and de-serialisation of data words from/to the ISA process through the DDC process onto the environment. This is useful when the ISA pipeline is relatively slower than the physical interface. For this reason, the SBC unit is able to shift data in and out of the system at speeds as fast as 1 bit each clock cycle, while the ISA process might need many clock cycles per data item. This means a shifting and serialisation procedure implemented in the ISA process is likely to take many cycles per bit.

The SBC block takes commands from the ISA process on the *sbcCtrl* channel. Currently, it uses the global clock (or a divider of the global clock) to control the serialisation and de-serialisation procedure. Data shifting in both directions (to and from the physical interface) has been modelled.

The SBC process is defined as follows:

$$\mathrm{SBC}_{sync} = \{|sbcCtrl, tock|\} \tag{5.50}$$

$$\mathrm{SBC} = \mathrm{SBC}\prime(port0, sbcCtrl) \tag{5.51}$$

$$\mathrm{SBC}\prime(port, sbcCtrl) = tock \rightarrow \mathrm{SBC}\prime(port, sbcCtrl) \tag{5.52}$$

$$\Box\ sbcCtrl.\,\mathrm{BUFI}?delay \rightarrow \tag{5.53}$$

$$\mathrm{SBC}_{inStart}(port, sbcCtrl, delay, 0) \tag{5.54}$$

$$\Box\ sbcCtrl.\,\mathrm{BUFO}?delay \rightarrow \tag{5.55}$$

$$sbcCtrl.\,\mathrm{BUFD}?data \rightarrow \tag{5.56}$$

$$\mathrm{SBC}_{out}(port, sbcCtrl, delay, \mathrm{DS}, data) \tag{5.57}$$

$$\mathrm{SBC}_{inStart}(port, sbcCtrl, d, w) = \mathrm{TOCK}(d - IDelay); \tag{5.58}$$

$$ctrl(port)!\,\mathrm{RAWI} \rightarrow dIn(port)?bit \rightarrow \tag{5.59}$$

$$\mathrm{SBC}_{in}(port, sbcCtrl, d, \mathrm{DS}-1, sright(w, bit)) \tag{5.60}$$

IDelay is used to model the number of cycles each instruction takes to execute in the ISA pipeline. The use of *IDelay* will become more apparent in Section 5.3.5.

In $\mathrm{CSP_M}$, the semicolon is used to compose two processes in sequence: when the first process successfully terminates (which is denoted by executing the SKIP process), the second process starts execution.

$$\mathrm{SBC}_{in}(port, sbcCtrl, d, count, w) = \tag{5.61}$$

$$\begin{cases} \left.\begin{array}{l} sbcCtrl.\,\mathrm{BUFD}!w \rightarrow \\ \mathrm{SBC}\prime(port, sbcCtrl) \end{array}\right\} & \text{if count=0} \\ \left.\begin{array}{r} \mathrm{TOCK}(d); \\ ctrl(port)!\,\mathrm{RAWI} \rightarrow \\ dIn(port)?bit \rightarrow \\ \mathrm{SBC}_{in}(port, sncCtrl, d, \\ count - 1, sright(w, bit)) \end{array}\right\} & \text{otherwise} \end{cases} \tag{5.62}$$

$$\mathrm{SBC}_{out}(port, sbcCtrl, d, count, w) = \tag{5.63}$$

$$\begin{cases} \left.\begin{array}{r} \mathrm{SBC}_{outOne}(port, w \bmod 2); \\ \mathrm{TOCK}(d - IDelay); \\ sbcCtrl.\,\mathrm{BUFR} \rightarrow \\ \mathrm{SBC}\prime(port, sbcCtrl) \end{array}\right\} & \text{if count=1} \\ \left.\begin{array}{r} \mathrm{SBC}_{outOne}(port, w \bmod 2); \\ \mathrm{TOCK}(d); \\ \mathrm{SBC}_{out}(port, sbcCtrl, \\ d, count - 1, w/2) \end{array}\right\} & \text{otherwise} \end{cases} \tag{5.64}$$

$$\mathrm{SBC}_{outOne}(port, bit) = ctrl(port)!\,\mathrm{RAWO} \rightarrow \tag{5.65}$$

$$dOut(port)!bit \rightarrow \mathrm{SKIP} \tag{5.66}$$

According to the specifications above, the SBC unit responds to the following two commands only:

1. BUFO: buffered output or serialisation; the SBC receives the delay (d) between each data bit counted in clock cycles. It also receives the *data* to be serialised. After all the data bits have been output on the attached DDC unit, the SBC notifies the ISA unit by issuing the buffer ready event (BUFR).

2. BUFI: buffered input or de-serialisation; similar to the BUFO, except that the data is transferred in the opposite direction (from the DDC through the SBC to the ISA). The transfer completes when the SBC notifies the ISA with the availability of data, using the BUFD event accompanied with the data word.

From a timing perspective, the SBC output process mirrors the SBC input process with respect to the relation between the bit I/O operations and the inserted delays. The exact timing specifications were developed and verified for correctness through interoperability assertions which coupled unbuffered configurations with the buffered ones. More details about those assertions will be discussed in Chapter 7.

5.3.5 Instruction Set Architecture (ISA)

The ISA process is an abstraction of the instruction set of a simple controller. The modelled instructions are needed for communication system configuration, data I/O and a selected set of arithmetic operations. The ISA process synchronises on *tock* events in order to meet the deadlines of communication protocols and avoid any race conditions. Each instruction takes at least one clock cycle (one *tock* event). The instruction delay could be manually changed to explore various design considerations. It is globally defined as *IDelay*. A special No-Operation (NOP) instruction is provided for implementing time delays.

5.3.5.1 Register File

While modelling the ISA process it became clear that the atomicity of instructions is important for subsequent modelling stages. Intermediate ISA state variables, such as inputted data and timestamps were best maintained using separate processes similar to the STC_{stamp} subprocess used by the STC process. Those processes make the stored state variables available to subsequent instructions. They essentially represent registers in traditional Instruction Set Architectures. Three registers were modelled; a time register, a bit register and a word register. The were all interleaved in a top-level *register file* process called REG. Many approaches to modelling the register file were attempted. Those include:

- an arbitrary sequence of any number of reads and writes;
- a single write followed by a number of reads; and
- enforcing a read after each write.

The last option proved most optimal in terms of state-space size and hence it was initially used for one register before it was adopted as the standard mechanism for reading and writing ISA state variables. This added a slight modelling overhead when used in the modelling of the whole instruction set. The adopted definition of a single register can be seen in Eqn. 5.75. Each register was accessed using an identification sequence. The first element in the sequence is the write channel, the second element is the read channel, and the third and last element is the capacity (the number of distinct values it could hold). The definition of those identification sequences can be seen in Eqns. 5.67 to 5.69.

$$bReg0 = \langle b0w, b0r, 2 \rangle \tag{5.67}$$
$$wReg0 = \langle w0w, w0r, 2^{\mathrm{DS}} \rangle \tag{5.68}$$
$$tReg0 = \langle t0w, t0r, 2^{\mathrm{TS}} \rangle \tag{5.69}$$
$$regs = \{bReg0, wReg0, tReg0\} \tag{5.70}$$

The following definitions were found useful for accessing the different elements of a register identification sequence:

$$write(reg) = head(reg) \tag{5.71}$$
$$read(reg) = head(tail(reg)) \tag{5.72}$$
$$limit(reg) = head(tail(tail(reg))) \tag{5.73}$$

Finally, the registers were defined as follows:

$$\mathrm{RO}(reg) = write(reg)?new \rightarrow \tag{5.74}$$
$$read(reg)!new \rightarrow \mathrm{RO}(reg) \tag{5.75}$$
$$\mathrm{WR} = \mathrm{RO}(wReg0) \tag{5.76}$$
$$\mathrm{TR} = \mathrm{RO}(tReg0) \tag{5.77}$$
$$\mathrm{DR} = \mathrm{RO}(bReg0) \tag{5.78}$$
$$\mathrm{REG} = \mathrm{DR} \;|||\; \mathrm{TR} \;|||\; \mathrm{WR} \tag{5.79}$$

The definitions above represent asynchronous zero-time read/write registers and hence timing consistency is maintained by the ISA process that is synchronising with this register file.

Roscoe [Chapter 18, 38] discusses Shared-Variables Programs, where a number of threads have access to a register file. In addition, Roscoe uses more complex models for the individual registers. The presented techniques are thought to be unrealistic for modelling data-sensitive programs and systems due to the state-space explosion problem, but could be useful for modelling abstract data-types.

5.3.5.2 Instruction Set

Instructions take $n \times IDelay$ cycles to execute where $n \geq 1$. For simple I/O and arithmetic instructions, n could equal 1. However, and for some experiments, n was allowed to be 0 which give rise to zero-time instructions. Of course this is not practically possible and would violate many timing checks, but when those instructions are coupled with I/O instructions or other instructions where $n \geq 1$, timing consistency can be established. In final experiments, n is always ≥ 1 for all instructions.

The simple arithmetic instructions which were defined for variable delay are defined as follows:

$$\mathrm{TC} = \mathrm{TOCK}(c) \tag{5.80}$$

$$\mathrm{ADD}(reg, val, c) = read(reg)?d \rightarrow \tag{5.81}$$

$$write(reg)!((d + val) \bmod count(reg)) \rightarrow \mathrm{TC} \tag{5.82}$$

$$\mathrm{TEST}(reg, val, \mathrm{PT}, \mathrm{PF}, c) = read(reg)?data \rightarrow write(reg)!data \rightarrow \tag{5.83}$$

$$\begin{cases} \mathrm{TC}\,;\mathrm{PT} & \text{if val=data} \\ \mathrm{TC}\,;\mathrm{PF} & \text{otherwise} \end{cases} \tag{5.84}$$

$$\mathrm{SHIFT_L}(wReg, bReg, c) = read(wReg)?word \rightarrow \tag{5.85}$$

$$write(wReg)!(word/2) \rightarrow \tag{5.86}$$

$$read(bReg)?dump \rightarrow \tag{5.87}$$

$$write(bReg)!(word \bmod 2) \rightarrow \mathrm{TC} \tag{5.88}$$

$$\mathrm{SHIFT_R}(wReg, bReg, c) = read(bReg)?bit \rightarrow \tag{5.89}$$

$$write(bReg)!bit \rightarrow read(wReg)?w \rightarrow \tag{5.90}$$

$$\begin{cases} write(wReg)!(w/2 + 2^{\mathrm{DS}-1}) \rightarrow \mathrm{TC} & \text{if b=1} \\ write(wReg)!(w/2) \rightarrow \mathrm{TC} & \text{otherwise} \end{cases} \tag{5.91}$$

$$\mathrm{INIT}(reg, val, c) = read(reg)?dump \rightarrow \tag{5.92}$$

$$write(reg)!val \rightarrow \mathrm{TC} \tag{5.93}$$

$$\mathrm{CLEAR}(reg, c) = write(reg)!0 \rightarrow \mathrm{TC} \tag{5.94}$$

$$\Box\ read(reg)?dump \rightarrow write(reg)!0 \rightarrow \mathrm{TC} \tag{5.95}$$

In the CSP description above, most instructions simply read register values, modify them and write the value back to the relevant registers. No interaction with the configurable blocks is needed. Because all registers are modelled as sequences of exactly one read and one write, the clear instruction accommodates for two alternatives: clearing the registers at the start of an instruction sequence as in Eqn. 5.94 and for intermediate clearing of register in Eqn. 5.95.

The list of ports that are available to I/O instructions could be defined as follows:

$$port0 = \langle ddcCtrl0, isaIn0, isaOut0 \rangle \tag{5.96}$$

$$port1 = \langle ddcCtrl1, isaIn1, isaOut1 \rangle \tag{5.97}$$

$$port2 = \langle ddcCtrl2, isaIn2, isaOut2 \rangle \tag{5.98}$$

$$port3 = \langle ddcCtrl3, isaIn3, isaOut3 \rangle \tag{5.99}$$

$$ports = \{port0, port1, port2, port3\} \tag{5.100}$$

Instructions that take exactly *IDelay* clock cycles include simple I/O instructions. Those are defined as follows:

$$\mathrm{TD} = \mathrm{TOCK}(IDelay) \tag{5.101}$$

$$\mathrm{RAW_{OUTV}}(port, data) = ctrl(port)!\,\mathrm{RAWO} \rightarrow \tag{5.102}$$

$$dOut(port)!data \rightarrow \mathrm{TD} \tag{5.103}$$

$$\mathrm{RAW_{OUT}}(port, reg) = read(reg)?data \rightarrow write(reg)!data \rightarrow \tag{5.104}$$

$$ctrl(port)!\,\mathrm{RAWO} \rightarrow dOut(port)!data \rightarrow \mathrm{TD} \tag{5.105}$$

$$\mathrm{RAW}_{\mathrm{IN}}(port, reg) = ctrl(port)!\,\mathrm{RAWI} \rightarrow dIn(port)?data \rightarrow \tag{5.106}$$
$$read(reg)?dump \rightarrow write(reg)!data \rightarrow \mathrm{TD} \tag{5.107}$$

$$\mathrm{STMP}_{\mathrm{READ}}(dPort, reg) = dPort?stmpTime \rightarrow read(reg)?dump \rightarrow \tag{5.108}$$
$$write(reg)!stmpTime \rightarrow \mathrm{TD} \tag{5.109}$$

$$\mathrm{GET}(chan, reg) = chan?data \rightarrow read(reg)?dump \rightarrow \tag{5.110}$$
$$write(reg)!data \rightarrow \mathrm{TD} \tag{5.111}$$

$$\mathrm{PUT}(chan, reg) = read(reg)?data \rightarrow \tag{5.112}$$
$$write(reg)!data \rightarrow chan!data \rightarrow \mathrm{TD} \tag{5.113}$$

$$\mathrm{NOP} = \mathrm{TD} \tag{5.114}$$

The RAW instructions interact with a DDC unit, the $\mathrm{STMP}_{\mathrm{READ}}$ interact with an STC block, and the GET / PUT pair are abstract data input and output instructions used for functional verification.

More complex instructions can potentially take more than *IDelay* cycles. Examples of complex instructions are time or data dependent instructions. Those execute in two stages, an initial configuration stage and a wait stage. For the wait stage, the following micro-instructions are defined:

$$\mathrm{WAIT}_{\mathrm{DATA}}(chan, reg) = chan?data \rightarrow read(reg)?dump \rightarrow \tag{5.115}$$
$$write(reg)!data \rightarrow \mathrm{TD} \tag{5.116}$$
$$\Box\ \mathrm{TD}\,;\mathrm{WAIT}_{\mathrm{DATA}}(chan, reg) \tag{5.117}$$

$$\mathrm{WAIT}_{\mathrm{VALUE}}(chan, val) = chan.val \rightarrow \mathrm{TD} \tag{5.118}$$
$$\Box\ \mathrm{TD}\,;\mathrm{WAIT}_{\mathrm{VALUE}}(chan, val) \tag{5.119}$$

$$\mathrm{WAIT}(chan) = chan \rightarrow \mathrm{TD} \tag{5.120}$$
$$\Box\ \mathrm{TD}\,;\mathrm{WAIT}(chan) \tag{5.121}$$

The $\mathrm{WAIT}_{\mathrm{DATA}}$ process waits for any item to arrive on *chan* which is then written to the provided register (*reg*) and the process finishes successfully. $\mathrm{WAIT}_{\mathrm{VALUE}}$ waits for the specified value to arrive on the communication channel, but does not save that value to a register. $\mathrm{WAIT}_{\mathrm{VALUE}}$ could be used to implement both delayed input

and delayed output. Finally, WAIT only waits for a simple event to happen on the provided channel.

Then, using those WAIT processes, the more complex I/O instructions are defined as follows:

$$\begin{aligned}
\mathrm{COND}_{\mathrm{IN}}(port, cond) = {} & ctrl(port).\,\mathrm{COND}!cond \rightarrow && (5.122)\\
& \mathrm{WAIT}_{\mathrm{VALUE}}(dIn(port), cond) && (5.123)\\
\mathrm{TMD}_{\mathrm{IN}}(tReg, bReg) = {} & read(tReg)?rTime \rightarrow write(tReg)!rTime \rightarrow && (5.124)\\
& iTmd.\,\mathrm{TMDI}!rTime \rightarrow \mathrm{WAIT}_{\mathrm{DATA}}(isaIn0, bReg) && (5.125)\\
\mathrm{TMD}_{\mathrm{OUT}}(tReg, bReg) = {} & read(tReg)?rTime \rightarrow write(tReg)!rTime \rightarrow && (5.126)\\
& iTmd.\,\mathrm{TMDO}!rTime \rightarrow read(bReg)?data \rightarrow && (5.127)\\
& write(bReg)!data \rightarrow \mathrm{WAIT}_{\mathrm{VALUE}}(isaOut0, data) && (5.128)\\
\mathrm{BUF}_{\mathrm{IN}}(del, reg) = {} & iBuf.\,\mathrm{BUFI}!del \rightarrow \mathrm{WAIT}_{\mathrm{DATA}}(iBuf.\,\mathrm{BUFD}, reg) && (5.129)\\
\mathrm{BUF}_{\mathrm{OUT}}(delay, reg) = {} & iBuf.\,\mathrm{BUFO}!delay \rightarrow && (5.130)\\
& read(reg)?data \rightarrow write(reg)!data \rightarrow && (5.131)\\
& iBuf.\,\mathrm{BUFD}!data \rightarrow \mathrm{WAIT}(iBuf.\,\mathrm{BUFR}) && (5.132)
\end{aligned}$$

The TMD instructions communicate with the STC unit to achieve timed input and timed output. For demonstration purposes, only one STC unit has been instantiated which is attached to *port0*. The BUF instructions communicate with the SBC unit to model buffered input and output. These instructions take the number of cycles between individual buffered items (*delay*), which is equivalent to deriving a regular clock from the system clock. Also, only one buffering unit has been instantiated, which is also assumed to be attached to *port0* in the specifications above. Finally, the data dependent I/O instruction ($\mathrm{COND}_{\mathrm{IN}}$) communicates with a DDC unit and commits once the correct data value has been observed on the physical interface.

A top-level abstract description of an ISA process is seen as any possible sequence of the above instructions.

$$\begin{aligned}
\mathrm{BOOT}_{\mathrm{CODE}} = {} & \mathrm{CLEAR}(wReg0, IDelay); && (5.133)\\
& \mathrm{CLEAR}(tReg0, IDelay); && (5.134)\\
& \mathrm{CLEAR}(bReg0, IDelay) && (5.135)
\end{aligned}$$

$$\mathrm{ISA} = \mathrm{BOOT}_{\mathrm{CODE}} \,;\, \mathrm{ISA}_{\mathrm{L}} \tag{5.136}$$

The provided description of an ISA process which could perform any possible sequence of instructions is only useful in some contexts: for example, to verify the deadlock, livelock freedom of all possible programs and their timing consistency. Such checks are discussed in Chapter 7. However, as the instruction set and the associated data sets grow, such checks become unrealistic. In such cases, the data sets would have to be limited to selected interesting values or restrictions would have to be imposed on the allowable sequences of instructions. In Chapter 6, instruction sequences are used to identify the functional and performance specification of selected communication protocols, which are then used to perform functional, performance and conformance checks in Chapter 7.

$$\begin{aligned}
\mathrm{ISA}_{\mathrm{L}} = {} & (\sqcap \forall\, port \in ports \wedge c \in Data \Rightarrow \mathrm{COND}_{\mathrm{IN}}(port, c) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.137)\\
\sqcap {} & (\sqcap \forall\, port \in ports \wedge d \in Data \Rightarrow \mathrm{RAW}_{\mathrm{OUTV}}(port, d) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.138)\\
\sqcap {} & (\sqcap \forall\, port \in ports \Rightarrow \mathrm{RAW}_{\mathrm{IN}}(port, bReg0) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.139)\\
\sqcap {} & (\sqcap \forall\, port \in ports \Rightarrow \mathrm{RAW}_{\mathrm{OUT}}(port, bReg0) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.140)\\
\sqcap {} & (\sqcap \forall\, delay \in TInterval \Rightarrow \mathrm{BUF}_{\mathrm{IN}}(delay, wReg0) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.141)\\
\sqcap {} & (\sqcap \forall\, delay \in TInterval \Rightarrow \mathrm{BUF}_{\mathrm{OUT}}(delay, wReg0) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.142)\\
\sqcap {} & (\sqcap \forall\, delay \in TInterval \Rightarrow \mathrm{ADD}(tReg0, delay, IDelay) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.143)\\
\sqcap {} & (\sqcap \forall\, d \in Data \Rightarrow \mathrm{TEST}(bReg0, d, \mathrm{SKIP}, \mathrm{SKIP}, IDelay) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.144)\\
\sqcap {} & (\sqcap \forall\, d \in Data \Rightarrow \mathrm{INIT}(bReg0, d, IDelay) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.145)\\
\sqcap {} & (\sqcap \forall\, reg \in regs \Rightarrow \mathrm{CLEAR}(reg, IDelay) \,;\, \mathrm{ISA}_{\mathrm{L}}) && (5.146)\\
\sqcap {} & \mathrm{STMP}_{\mathrm{READ}}(iTime, tReg0) \,;\, \mathrm{ISA}_{\mathrm{L}} && (5.147)\\
\sqcap {} & \mathrm{TMD}_{\mathrm{IN}}(tReg0, bReg0) \,;\, \mathrm{ISA}_{\mathrm{L}} && (5.148)\\
\sqcap {} & \mathrm{TMD}_{\mathrm{OUT}}(tReg0, bReg0) \,;\, \mathrm{ISA}_{\mathrm{L}} && (5.149)\\
\sqcap {} & \mathrm{GET}(produce, wReg0) \,;\, \mathrm{ISA}_{\mathrm{L}} && (5.150)\\
\sqcap {} & \mathrm{PUT}(consume, wReg0) \,;\, \mathrm{ISA}_{\mathrm{L}} && (5.151)\\
\sqcap {} & \mathrm{SHIFT}_{\mathrm{R}}(wReg0, bReg0, IDelay) \,;\, \mathrm{ISA}_{\mathrm{L}} && (5.152)\\
\sqcap {} & \mathrm{SHIFT}_{\mathrm{L}}(wReg0, bReg0, IDelay) \,;\, \mathrm{ISA}_{\mathrm{L}} && (5.153)\\
\sqcap {} & \mathrm{NOP} \,;\, \mathrm{ISA}_{\mathrm{L}} && (5.154)
\end{aligned}$$

5.4 Design Exploration: Configurable System Construction

Once the configurable blocks have been modelled and verified individually, a system can be constructed by connecting the relevant control channels between the individual components. This is achieved through renaming and parallel composition in CSP. For example, a system with a single DDC unit, connected to an STC unit and an SBC unit could be constructed as follows:

$$aReg = \{|b0r, b0w, t0r, t0w, w0r, w0w|\} \tag{5.155}$$

$$aSbc = \{|sbcCtrl, tock|\} \tag{5.156}$$

$$aStc = \{|stcCtrl, stcTime, tock|\} \tag{5.157}$$

$$aDdc = \{|ddcCtrl0, isaIn0, isaOut0|\} \cup \{|stcCtrlDdc|\} \cup \{tock\} \tag{5.158}$$

$$dLI = ddcLeftIn \tag{5.159}$$

$$dLO = ddcLeftOut \tag{5.160}$$

$$dRI = ddcRightIn \tag{5.161}$$

$$dRO = ddcRightOut \tag{5.162}$$

$$\text{DDC0} = \tag{5.163}$$

$$\text{DDC} \, [\![{}^{dLI, dLO, dRI, dRO, ddcCtrl}/_{isaOut0, isaIn0, dRI0, dRO0, ddcCtrl0}]\!] \tag{5.164}$$

$$\text{SYSTEM}_{one\text{-}bit} = (((\text{ISA} \underset{aReg}{\parallel} \text{REG}) \underset{aSbc}{\parallel} \text{SBC}) \underset{aStc}{\parallel} \text{STC}) \underset{aDdc}{\parallel} \text{DDC0} \tag{5.165}$$

Because the composition only involves one process of each configurable block, the construction is relatively straightforward: connecting the DDC unit to the ISA through attaching the relevant data channels. A form of CSP renaming is used to accomplish this, as seen in Eqn. 5.164. Similarly, the SBC unit is attached to the ISA process, as well as to the DDC unit through functional style renaming (See Eqn. 5.51). A block diagram of $\text{SYSTEM}_{one\text{-}bit}$ is demonstrated in Figure 5.4.

It is evident from the CSP descriptions of the relevant units, as well as from Figure 5.4 that there are two control channels between the STC and the DDC unit. This is useful when the number of STC units constructed does not match the number of DDC units, in which situation it is possible to connect either the forward control channel (*ddcCtrlStc*) or the feedback control channel (*stcCtrlDdc*) as per the explored

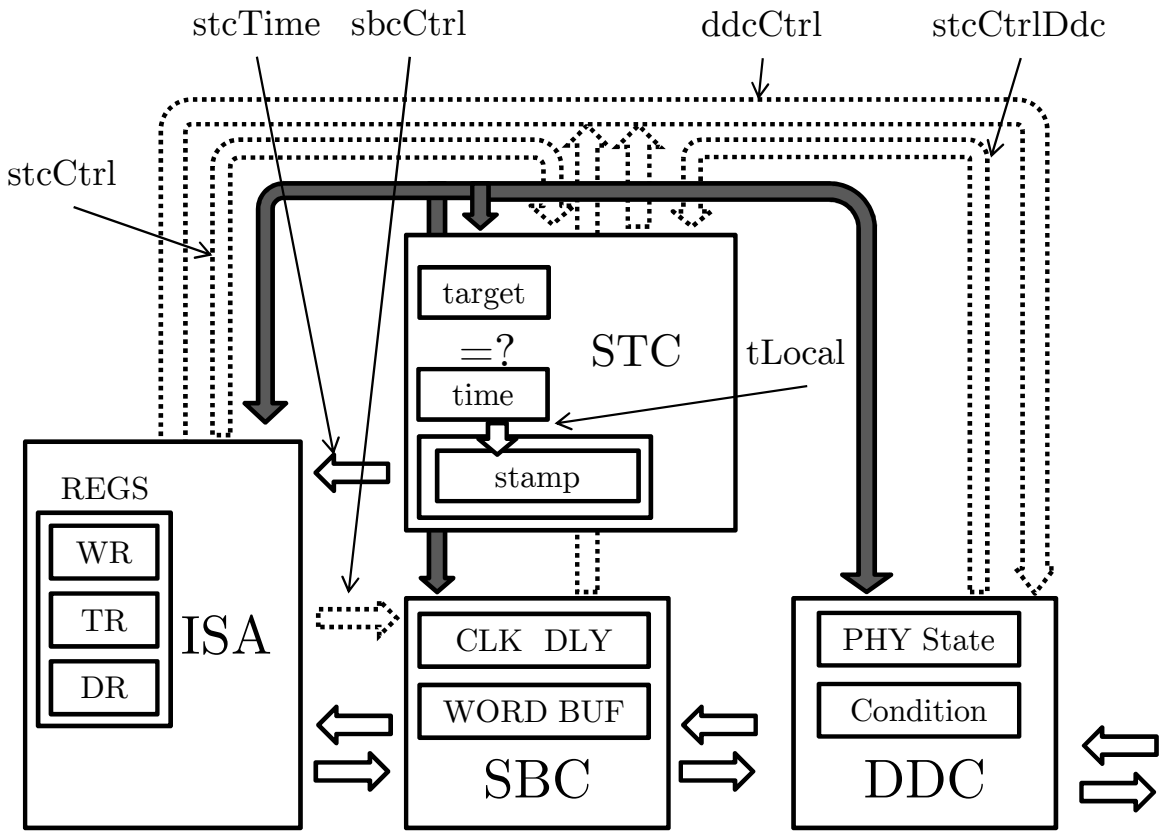


Figure 5.4 One Bit Concrete Communication System

design requirements. For example, a single STC could be made to issue synchronised timed commands to a number of DDC units, but only one of the DDC units is connected through the *stcCtrlDdc* channel to the STC unit and hence is able to respond with timestamping commands.

Many communication protocols would have wider interfaces with more than 1 bit, in which case DDC processes are composed in parallel to reflect the number of bits the interface has. For this reason, the DDC unit went through a rigorous optimisation process, in order for the interleaving of many units not to have a great impact on the complexity of the overall system.

This enabled the modelling and subsequent verification of protocols which use many independent one bit ports, such as the SPI protocol. This is demonstrated in Section 6.4.2.

As an example, a system involving 4 DDC units and 1 STC unit is constructed in Eqn. 5.179. A top-level diagram of this construction can be seen in Figure 5.5.

$$\text{DDC1} = \tag{5.166}$$

$$\text{DDC} \llbracket dLI, dLO, dRI, dRO, ddcCtrl / isaOut1, isaIn1, dRI1, dRO1, ddcCtrl1 \rrbracket \tag{5.167}$$

$$\text{DDC2} = \tag{5.168}$$

$$\text{DDC} \llbracket dLI, dLO, dRI, dRO, ddcCtrl / isaOut2, isaIn2, dRI2, dRO2, ddcCtrl2 \rrbracket \tag{5.169}$$

$$\text{DDC3} = \tag{5.170}$$

$$\text{DDC} \llbracket dLI, dLO, dRI, dRO, ddcCtrl / isaOut3, isaIn3, dRI3, dRO3, ddcCtrl3 \rrbracket \tag{5.171}$$

$$aDdc4Bit = \{\!| ddcCtrl0, isaIn0, isaOut0 |\!\} \tag{5.172}$$

$$\cup \{\!| ddcCtrl1, isaIn1, isaOut1 |\!\} \tag{5.173}$$

$$\cup \{\!| ddcCtrl2, isaIn2, isaOut2 |\!\} \tag{5.174}$$

$$\cup \{\!| ddcCtrl3, isaIn3, isaOut3 |\!\} \tag{5.175}$$

$$\cup \{\!| stcCtrlDdc |\!\} \tag{5.176}$$

$$\cup \{tock\} \tag{5.177}$$

$$\text{DDC}_{\textit{4-bit}} = ((\text{DDC0} \underset{\{tock\}}{\parallel} \text{DDC1}) \underset{\{tock\}}{\parallel} \text{DDC2}) \underset{\{tock\}}{\parallel} \text{DDC3} \tag{5.178}$$

$$\text{SYSTEM}_{\textit{four-bit}} = (((\text{ISA} \underset{aReg}{\parallel} \text{REG}) \underset{aSbc}{\parallel} \text{SBC}) \underset{aStc}{\parallel} \text{STC}) \underset{aDdc}{\parallel} \text{DDC}_{\textit{4-bit}} \tag{5.179}$$

5.5 Results and Complexity Analysis

Contrary to Section 3.4, where a universal data-type was used, the concrete system used multiple data-types, many of which are of fixed sizes, such as the *Boolean* data-type. Two types, however, still had variable sizes: the *DataWord* type, which is configurable using DS and the *TimeWord* type, which is configurable using TS. In essence, *TimeWord* is a new type, which is used for capturing time, while the *DataWord* type discussed in Section 3.4 has been used only for holding data words to be communicated by the modelled protocol.

Table 5.1 shows how the size of each system component grows as the value of DS changes. The results were gathered while fixing TS to be equal 2. Table 5.2, on the other hand, shows the results as TS changes, while DS equals 2.

The results were obtained using the optimised version of FDR 2.91 Academic Release running on a Linux kernel version 2.6.32. In addition to the optimisations described in Section 4.5, an efficient statistics extraction mechanism was integrated

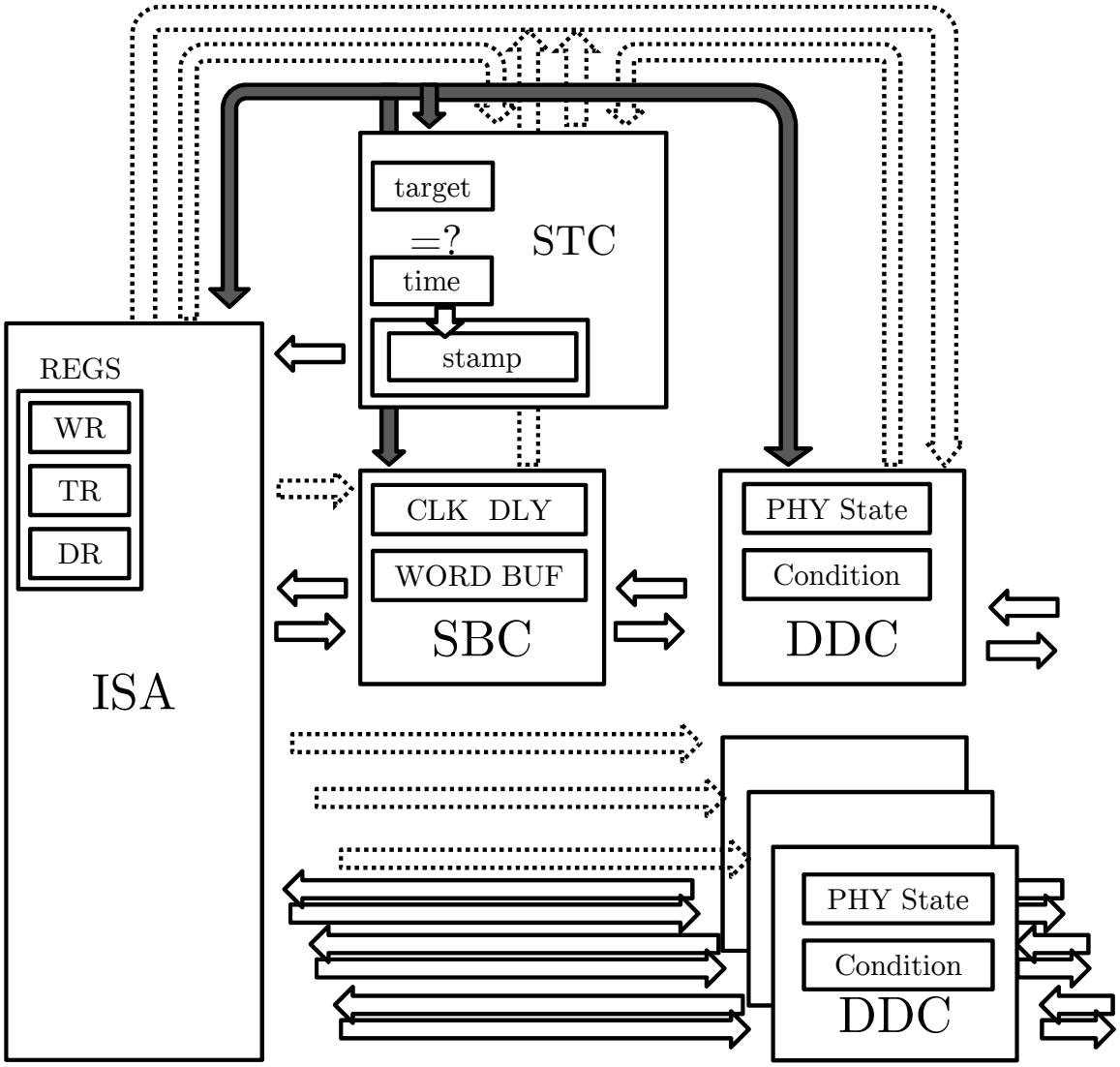


Figure 5.5 Four Bit Concrete Communication System

into FDR. The computer used had 8 Cores, each of which is a 3.16 GHz Intel Xeon processor. All processors shared 40 GB of available main memory. Under this environment, extracting the metrics in Table 5.1 took 48 minutes and 34 seconds, where all metrics were extracted for DS = $\{1..8\}$. Extracting the metrics in Table 5.2 took 55 minutes and 15 seconds. Again, all metrics were extracted for TS = $\{1..8\}$ except for the top-level SYSTEM process, which was only extracted for TS = $\{1..4\}$. In an attempt to extract the metrics for the SYSTEM process for TS = 5, FDR consumed all available 40 GB of main memory, before it was gracefully terminated by the kernel.

By using the same technique described in Section 4.6.3, the results in Tables 5.1 and 5.2 were analysed.

Table 5.1 Metric Results by Varying $DataSize$ while $TimeSize = 2$

DataSize (DS)			1	2	3	4	5	6	7	8	SS
REG	Events	Measured	18	22	30	46	78	142	270	526	
		Predicted	18	22	30	46	78	142	270	526	
		$Error^2$	0	0	0	0	0	0	0	0	0
	States	Measured	45	75	135	255	495	975	1935	3855	
		Predicted	45	75	135	255	495	975	1935	3855	
		$Error^2$	0	0	0	0	0	0	0	0	0
	Trans.	Measured	192	340	636	1228	2412	4780	9516	18988	
		Predicted	192	340	636	1228	2412	4780	9516	18988	
		$Error^2$	0	0	0	0	0	0	0	0	0
ISA	Events	Measured	72	82	102	142	222	382	702	1342	
		Predicted	72	82	102	142	222	382	702	1342	
		$Error^2$	0	0	0	0	0	0	0	0	0
	States	Measured	155	171	199	255	367	591	1039	1935	
		Predicted	157	171	199	255	367	591	1039	1935	
		$Error^2$	4	0	0	0	0	0	0	0	4
	Trans.	Measured	255	301	409	721	1729	5281	18529	69601	
		Predicted	258	300	408	720	1728	5280	18528	69600	
		$Error^2$	9	1	1	1	1	1	1	1	16

Table 5.1 Continued ↓

DataSize (DS)			1	2	3	4	5	6	7	8	SS
SBC	Events	Measured	20	22	26	34	50	82	146	274	
		Predicted	20	22	26	34	50	82	146	274	
		$Error^2$	0	0	0	0	0	0	0	0	0
	States	Measured	32	107	263	575	1199	2447	4943	9935	
		Predicted	78	156	312	624	1248	2496	4992	9984	
		$Error^2$	2116	2401	2401	2401	2401	2401	2401	2401	18923
	Trans.	Measured	45	133	315	679	1407	2863	5775	11599	
		Predicted	90	180	360	720	1440	2880	5760	11520	
		$Error^2$	2025	2209	2025	1681	1089	289	225	6241	15784
STC	Events	Measured	21	21	21	21	21	21	21	21	
		Predicted	21	21	21	21	21	21	21	21	
		$Error^2$	0	0	0	0	0	0	0	0	0
	States	Measured	332	332	332	332	332	332	332	332	
		Predicted	332	332	332	332	332	332	332	332	
		$Error^2$	0	0	0	0	0	0	0	0	0
	Trans.	Measured	880	880	880	880	880	880	880	880	
		Predicted	880	880	880	880	880	880	880	880	
		$Error^2$	0	0	0	0	0	0	0	0	0

Table 5.1 Continued ↓

DataSize (DS)			1	2	3	4	5	6	7	8	SS
DDC	Events	Measured	20	20	20	20	20	20	20	20	
		Predicted	20	20	20	20	20	20	20	20	
		Error^2	0	0	0	0	0	0	0	0	0
	States	Measured	20	20	20	20	20	20	20	20	
		Predicted	20	20	20	20	20	20	20	20	
		Error^2	0	0	0	0	0	0	0	0	0
	Trans.	Measured	62	62	62	62	62	62	62	62	
		Predicted	62	62	62	62	62	62	62	62	
		Error^2	0	0	0	0	0	0	0	0	0
SYSTEM	Events	Measured	79	89	109	149	229	389	709	1349	
		Predicted	79	89	109	149	229	389	709	1349	
		Error^2	0	0	0	0	0	0	0	0	0
	States	Measured	144221	336541	776477	1927709	5276701	16082973	53973021	1.9E+08	
		Predicted	175018	371220	827176	1993296	5342368	16107840	53908096	1.9E+08	
		Error^2	9.5E+08	1.2E+09	2.6E+09	4.3E+09	4.3E+09	6.2E+08	4.2E+09	1.2E+09	1.9E+10
	Trans.	Measured	313911	714039	1615415	3918903	10467383	31207479	1.0E+08	3.7E+08	
		Predicted	364838	769140	1696136	4023696	10573088	31248960	1.0E+08	3.7E+08	
		Error^2	2.6E+09	3.0E+09	6.5E+09	1.1E+10	1.1E+10	1.7E+09	1.1E+10	3.5E+08	4.7E+10

Table 5.1 Metric Results by Varying $DataSize$ while $TimeSize = 2$

Table 5.2 Metric Results by Varying $TimeSize$ while $DataSize = 2$

TimeSize (TS)			1	2	3	4	5	6	7	8	SS
REG	Events	Measured	18	22	30	46	78	142	270	526	
		Predicted	18	22	30	46	78	142	270	526	
		Error^2	0	0	0	0	0	0	0	0	0
	States	Measured	45	75	135	255	495	975	1935	3855	
		Predicted	45	75	135	255	495	975	1935	3855	
		Error^2	0	0	0	0	0	0	0	0	0
	Trans.	Measured	192	340	636	1228	2412	4780	9516	18988	
		Predicted	192	340	636	1228	2412	4780	9516	18988	
		Error^2	0	0	0	0	0	0	0	0	0
ISA	Events	Measured	68	82	110	166	278	502	950	1846	
		Predicted	68	82	110	166	278	502	950	1846	
		Error^2	0	0	0	0	0	0	0	0	0
	States	Measured	147	171	219	315	507	891	1659	3195	
		Predicted	147	171	219	315	507	891	1659	3195	
		Error^2	0	0	0	0	0	0	0	0	0
	Trans.	Measured	241	301	469	997	2821	9541	35269	135877	
		Predicted	241	301	469	997	2821	9541	35269	135877	
		Error^2	0	0	0	0	0	0	0	0	0

Table 5.2 Continued ↓

TimeSize (TS)			1	2	3	4	5	6	7	8	SS
SBC	Events	Measured	18	22	30	46	78	142	270	526	
		Predicted	18	22	30	46	78	142	270	526	
		Error^2	0	0	0	0	0	0	0	0	0
	States	Measured	52	107	253	689	2137	7337	26953	103049	
		Predicted	52	107	253	689	2137	7337	26953	103049	
		Error^2	0	0	0	0	0	0	0	0	0
	Trans.	Measured	66	133	303	787	2331	7723	27723	104587	
		Predicted	66	133	303	787	2331	7723	27723	104587	
		Error^2	0	0	0	0	0	0	0	0	0
STC	Events	Measured	15	21	33	57	105	201	393	777	
		Predicted	15	21	33	57	105	201	393	777	
		Error^2	0	0	0	0	0	0	0	0	0
	States	Measured	54	332	2328	17456	135264	1065152	8454528	67371776	
		Predicted	54	332	2328	17456	135264	1065152	8454528	67371776	
		Error^2	0	0	0	0	0	0	0	0	0
	Trans.	Measured	144	880	6048	44608	342144	2679040	21201408	168690688	
		Predicted	144	880	6048	44608	342144	2679040	21201408	168690688	
		Error^2	0	0	0	0	0	0	0	0	0

Table 5.2 Continued ↓

TimeSize (TS)			1	2	3	4	5	6	7	8	SS
DDC	Events	Measured	20	20	20	20	20	20	20	20	
		Predicted	20	20	20	20	20	20	20	20	
		$Error^2$	0	0	0	0	0	0	0	0	0
	States	Measured	20	20	20	20	20	20	20	20	
		Predicted	20	20	20	20	20	20	20	20	
		$Error^2$	0	0	0	0	0	0	0	0	0
	Trans.	Measured	62	62	62	62	62	62	62	62	
		Predicted	62	62	62	62	62	62	62	62	
		$Error^2$	0	0	0	0	0	0	0	0	0
SYSTEM	Events	Measured	75	89	117	173					
		Predicted	75	89	117	173					
		$Error^2$	0	0	0	0					0
	States	Measured	35005	336541	3929629	58019869					
		Predicted	34945	336644	3929579	58019876					
		$Error^2$	3600	10650	2540	52					16842
	Trans.	Measured	73655	714039	8330807	121957431					
		Predicted	73113	714708	8330673	121957438					
		$Error^2$	293764	447829	17956	46					759595

Table 5.2 Metric Results by Varying $TimeSize$ while $DataSize = 2$

Table 5.3 shows all the multiplicand values for all the processes with respect to the results presented in Table 5.1. Table 5.4, on the other hand, shows the complexity results for Table 5.2.

Multiplicand		1	2^{DS}	$2^{2\times DS}$
REG	Events	14	2	
	States	15	15	
	Transitions	44	74	
ISA	Events	62	5	
	States	143	7	
	Transitions	224	15	1
SBC	Events	18	1	
	States	0	39	
	Transitions	0	45	
STC	Events	21		
	States	332		
	Transitions	880		
DDC	Events	20		
	States	20		
	Transitions	62		
SYSTEM	Events	69	5	
	States	0	82213	2648
	Transitions	0	172553	4933

Table 5.3 Factor Results by Varying $DataSize$ while $TimeSize = 2$

The complexity analysis was performed using Excel 2007, running on Windows 7 virtual machine (VMware Player version 5.0.2). The computer that was used had an 1.60 GHz Intel Core *i7* processor and 8 GB of available main memory. Using this configuration, producing all the results in Table 5.3 took 27 seconds, an average of 1.5 seconds per complexity equation. Similarly, solving all the results in Table 5.4 took 30 seconds, an average of 1.7 seconds per equation.

The algorithm succeeded in solving most of the equations and the final value of the sum of errors squares (SS) was zero. However, in some cases, this was not correct. Tables 5.1 and 5.2 show the final SS results after the algorithm had been applied.

For clarity, the *SComplex* and *CComplex* equations summarised in Tables 5.3 and 5.4 have been rewritten in equation form. The final *State-Space Complexity* and

Multiplicand		1	2^{TS}	$2^{2\times TS}$	$2^{3\times TS}$	$2^{4\times TS}$	$2^{5\times TS}$
REG	Events	14	2				
	States	15	15				
	Transitions	44	74				
ISA	Events	54	7				
	States	123	12				
	Transitions	197	18	2			
SBC	Events	14	2				
	States	9	18.5	1.5			
	Transitions	11	24.5	1.5			
STC	Events	9	3				
	States	0	3	4	4		
	Transitions	0	4	14	10		
DDC	Events	20					
	States	20					
	Transitions	62					
SYSTEM	Events	61	7				
	States	1073	8.8	0.2	3356.7	403.5	17
	Transitions	420.2	0	425	6960.9	889.8	33.4

Table 5.4 Factor Results by Varying $TimeSize$ while $DataSize = 2$

Communication-Space Complexity formulae with respect to the size of the *DataWord* data-type (DS) can be seen in Eqns. 5.180 to 5.191.

Eqns. 5.186 to 5.189 show the independence of both the DDC and STC units from DS, which conforms to good design and verification practices and justifies the effort dedicated to lifting the STC unit out of the data path.

$$\mathrm{REG}_{SComplex}(\mathrm{DS}) = 15 + 15 \times 2^{\mathrm{DS}} \tag{5.180}$$

$$\mathrm{REG}_{CComplex}(\mathrm{DS}) = 44 + 74 \times 2^{\mathrm{DS}} \tag{5.181}$$

$$\mathrm{ISA}_{SComplex}(\mathrm{DS}) = 143 + 7 \times 2^{\mathrm{DS}} \tag{5.182}$$

$$\mathrm{ISA}_{CComplex}(\mathrm{DS}) = 225 + 15 \times 2^{\mathrm{DS}} + 1 \times 2^{2\times \mathrm{DS}} \tag{5.183}$$

$$\mathrm{SBC}_{SComplex}(\mathrm{DS}) = 39 \times 2^{\mathrm{DS}} \tag{5.184}$$

$$\mathrm{SBC}_{CComplex}(\mathrm{DS}) = 45 \times 2^{\mathrm{DS}} \tag{5.185}$$

$$\mathrm{STC}_{SComplex}(\mathrm{DS}) = 332 \tag{5.186}$$

$$\mathrm{STC}_{CComplex}(\mathrm{DS}) = 880 \tag{5.187}$$

$$\mathrm{DDC}_{SComplex}(\mathrm{DS}) = 20 \tag{5.188}$$

$$\mathrm{DDC}_{CComplex}(\mathrm{DS}) = 62 \tag{5.189}$$

As for the SBC unit, it was not possible to get satisfactory solutions to the complexity formulae that closely match the metrics extracted from the model-checker. The results demonstrated in Eqns. 5.184 and 5.185 show the best possible solution using the constraints and generic complexity formulae discussed in Section 4.6. They are associated with relatively large errors (SS values) in Table 5.1.

By looking closely at the use of the DS configuration parameter in the SBC unit, it is evident that not only was it used to configure the size of the *DataWord* data-type, but also as a loop invariant for the buffering and serialisation mechanism. Eqns. 5.57 and 5.60 demonstrate this use, which is not thought to be of exponential nature. This is also evident in the complexity formulae of the top-level SYSTEM process (Eqns. 5.190 and 5.191). The associated SS values for the *SComplex* and *CComplex* equations are $1.9\,\mathrm{E}{+10}$ and $4.7\,\mathrm{E}{+10}$, respectively. See Table 5.1 for more details. For this reason, it is expected that the generic formula in Eqn. 4.8 would need to be altered to take into account non-exponential components.

$$\mathrm{SYSTEM}_{SComplex}(\mathrm{DS}) = 82147 \times 2^{\mathrm{DS}} + 2648 \times 2^{2\times\mathrm{DS}} \tag{5.190}$$

$$\mathrm{SYSTEM}_{CComplex}(\mathrm{DS}) = 172553 \times 2^{\mathrm{DS}} + 4933 \times 2^{2\times\mathrm{DS}} \tag{5.191}$$

With respect to the size of *TimeWord* data-type (TS), the resulted formulae for all processes can be seen in Eqns. 5.192 to 5.205.

$$\mathrm{REG}_{SComplex}(\mathrm{TS}) = 15 + 15 \times 2^{\mathrm{TS}} \tag{5.192}$$

$$\mathrm{REG}_{CComplex}(\mathrm{TS}) = 44 + 74 \times 2^{\mathrm{TS}} \tag{5.193}$$

$$\mathrm{ISA}_{SComplex}(\mathrm{TS}) = 123 + 12 \times 2^{\mathrm{TS}} \tag{5.194}$$

$$\mathrm{ISA}_{CComplex}(\mathrm{TS}) = 197 + 18 \times 2^{\mathrm{TS}} + 2 \times 2^{2\times\mathrm{TS}} \tag{5.195}$$

$$\mathrm{SBC}_{SComplex}(\mathrm{TS}) = 9 + 18.5 \times 2^{\mathrm{TS}} + 1.5 \times 2^{2\times\mathrm{TS}} \tag{5.196}$$

$$\mathrm{SBC}_{CComplex}(\mathrm{TS}) = 11 + 24.5 \times 2^{\mathrm{TS}} + 1.5 \times 2^{2\times\mathrm{TS}} \tag{5.197}$$

$$\mathrm{STC}_{SComplex}(\mathrm{TS}) = 3 \times 2^{\mathrm{TS}} + 4 \times 2^{2\times\mathrm{TS}} + 4 \times 2^{3\times\mathrm{TS}} \tag{5.198}$$

$$\mathrm{STC}_{CComplex}(\mathrm{TS}) = 4 \times 2^{\mathrm{TS}} + 14 \times 2^{2\times\mathrm{TS}} + 10 \times 2^{3\times\mathrm{TS}} \tag{5.199}$$

$$\mathrm{DDC}_{SComplex}(\mathrm{TS}) = 20 \tag{5.200}$$

$$\mathrm{DDC}_{CComplex}(\mathrm{TS}) = 62 \tag{5.201}$$

Eqns. 5.192 to 5.199 show the large dependency of all associated units on the *TimeWord* data-type. This is predictable for the STC, ISA and REG units: by definition the STC manages the timing and synchronisation aspects of the system, the ISA configures and manages that unit, while the REG is needed for temporarily holding timestamps during configuration.

As for the SBC unit, however, the *TimeWord* data-type is used for managing the buffering functions in a timely manner. By doing so, the SBC unit is decoupled from the STC unit. Another approach would be to let the STC unit manage the timing aspects of the SBC unit. This would introduce a design and modelling complication, but is expected to reduce the complexity of the SBC unit with respect to TS. It also means that both the SBC and the STC units would be involved in a single I/O operation.

Finally, the *SComplex* and *CComplex* equations of the top-level SYSTEM process with respect to TS are demonstrated in Eqns. 5.202 and 5.204, respectively.

$$\begin{aligned}
\mathrm{SYSTEM}_{SComplex}(\mathrm{TS}) &= 1073 + 8.8 \times 2^{\mathrm{TS}} + 0.2 \times 2^{2\times\mathrm{TS}} + && (5.202)\\
&\quad 3356.7 \times 2^{3\times\mathrm{TS}} + 403.5 \times 2^{4\times\mathrm{TS}} + 17 \times 2^{5\times\mathrm{TS}} && (5.203)\\
\mathrm{SYSTEM}_{CComplex}(\mathrm{TS}) &= 420.2 + 425 \times 2^{2\times\mathrm{TS}} + 6960.9 \times 2^{3\times\mathrm{TS}} + && (5.204)\\
&\quad 889.8 \times 2^{4\times\mathrm{TS}} + 33.4 \times 2^{5\times\mathrm{TS}} && (5.205)
\end{aligned}$$

5.5.1 Asymptotic Evaluation

Asymptotic components of the complexity formulae in Eqns. 5.180 to 5.191 are presented in Table 5.5.

Process	REG	ISA	SBC	STC	DDC	SYSTEM
$SComplex(\mathrm{DS}) \in$	$O(2^{\mathrm{DS}})$	$O(2^{\mathrm{DS}})$	$O(2^{\mathrm{DS}})$	$O(1)$	$O(1)$	$O(2^{2\times\mathrm{DS}})$
$CComplex(\mathrm{DS}) \in$	$O(2^{\mathrm{DS}})$	$O(2^{2\times\mathrm{DS}})$	$O(2^{\mathrm{DS}})$	$O(1)$	$O(1)$	$O(2^{2\times\mathrm{DS}})$

Table 5.5 Asymptotic Complexity of Processes by Varying *DataSize*

Similarly, asymptotic components of the formulae in Eqns. 5.192 to 5.205 are presented in Table 5.6.

Table 5.5 shows the low dependence of the SYSTEM process on DS, which was of $O(2^{2\times\mathrm{DS}})$, as opposed to the high dependence on the TS, which was of $O(2^{5\times\mathrm{TS}})$. This demonstrates the high quality of the associated CSP description and its low dependence on DS.

Process	REG	ISA	SBC	STC	DDC	SYSTEM
$SComplex(\mathrm{TS}) \in$	$O(2^{\mathrm{TS}})$	$O(2^{\mathrm{TS}})$	$O(2^{2\times\mathrm{TS}})$	$O(2^{3\times\mathrm{TS}})$	$O(1)$	$O(2^{5\times\mathrm{TS}})$
$CComplex(\mathrm{TS}) \in$	$O(2^{\mathrm{TS}})$	$O(2^{2\times\mathrm{TS}})$	$O(2^{2\times\mathrm{TS}})$	$O(2^{3\times\mathrm{TS}})$	$O(1)$	$O(2^{5\times\mathrm{TS}})$

Table 5.6 Asymptotic Complexity of Processes by Varying $TimeSize$

It is clear from Table 5.6 that the complexity of the DDC unit is orthogonal to TS, which proves the fact that the DDC unit is not functionally dependent on any timing aspects. With respect to the complexity of the STC unit in comparison to the results presented earlier in Table 4.10, it has been noticed that the complexity increased by a factor of $O(2^{\mathrm{TS}})$. This is due to the introduction of a timestamp function, which required the top-level STC process to keep an extra state variable of type *TimeWord*, in addition to the target time of the operation and the current time. The timestamping mechanism is seen as an extra architectural constraint imposed by the design decision-making process.

Finally, the dependence of many configurable units on TS results in the high dependence of the top-level SYSTEM construction on TS (with an asymptotic complexity of $O(2^{5\times\mathrm{TS}})$). Though the full complexity semantics of the CSP parallelism and synchronisation operators are not clear at this stage, one could deduce from the SYSTEM process construction in Eqn. 5.165 and from the synchronisation alphabets that the *TimeWord* data-type has been used as a state variable in 5 different interleaved states at the same time. For example: REG read *or* write, SBC output delay *or* input delay, old STC timestamp, STC current time counter, and STC future output *or* input time.

5.6 Summary

Through the consideration of various design and verification approaches from the top-level configurable communication hierarchy to block level and instruction level architecture, the formal system presented in this chapter offers a solid design exploration platform for the design and verification of communication systems and protocols.

A formal CSP description was provided, along with a brief description of the design iterations involved such as complexity analysis, functional units optimisation and top-level construction and synchronisation considerations. The final models described in Section 5.3 showcase a selection of configurable units, configurations, as well as a minimum arithmetic instruction set. These models demonstrate the building blocks of the general purpose modelling system for communication protocols proposed in

this thesis. Its use for formal modelling and verification of communication protocols will be discussed in Chapters 6 and 7.

5.7 Future Work

It is of great importance to address the complexity and orthogonality issues arising from the high dependency of many configurable units and the top-level system construction on the *TimeWord* data-type. Once this is achieved, it would become more practical to extend the system with more functional units and additional configurations for the existing units.

A useful improvement to the complexity analysis performed in this chapter would be to explore a change in the format of the complexity formulae to take into account non-exponential aspects. Questions to be answered: would the non-exponential components have additive or multiplicative implications on the exponential formulae? Also, since these components would increase the complexity and the number of variables of the formulae, how would the GRG algorithm cope with such increase in complexity?

All of the analysis methods in Section 5.5 used one DDC unit. It would be interesting to perform complexity analysis and optimisation as the number of DDC units increases.

The models discussed in this chapter could become an integral part of a formal design exploration tool, which could be used in the design of both I/O *Instruction Set Architecture*s and communication protocols. This tool could define an abstract protocol specification language at a level higher or equivalent to ISA, which could be compiled into an ISA specification. The ISA specification can then be automatically translated into CSP. This would be facilitated by the availability of a set of CSP_M libraries, reusable components and extendible interfaces similar to that which has been presented in this chapter. All this would facilitate and speed up the exploration and formal verification of larger systems.

Finally, it would also be interesting to explore the design of higher level abstractions, such as a multithreaded *Instruction Set Architecture*. Analysis of the possible performance implications on the associated communication interfaces would be interesting.

CHAPTER 6

ISA-Oriented and Abstract Specification of Communication Protocols

6.1 Motivation and Chapter Structure

Chapter 5 presented the formal specification of the configurable system along with its instruction set. In this chapter, the use of such a configurable system for the modelling of communication protocols is demonstrated. First, Section 6.2 discusses previous attempts on modelling and model-checking individual communication protocols and how those attempts fell short from providing a general model-checking framework for communication protocols. Then Section 6.3 presents some preliminary definitions that are used in the specification process.

Section 6.4 presents a CSP modelling and verification approach called: ISA-*Oriented Specification*. This approach employs the instruction set modelled in Chapter 5 in order to specify the collective functional and performance properties of the selected communication protocols. This is achieved in two steps:

1. Static construction of the configurable blocks into the desired communication system.

2. Dynamic configuration of the communication blocks through the use of ISA-Oriented Specifications.

An additional proof of the validity of the ISA-*Oriented Specification* models is needed. Independent and simplistic abstract models of the selected protocols are presented in Section 6.5. Those specifications are independent of all the technical details

of the configurable system and the associated ISA-*Oriented Specification* approach. The use of those abstract models for proving the conformance of the ISA-*Oriented Specification* will become clear in Chapter 7.

Figure 6.1 highlights the abstract modelling of the communication protocols and its position in the design and verification of the configurable communication system.

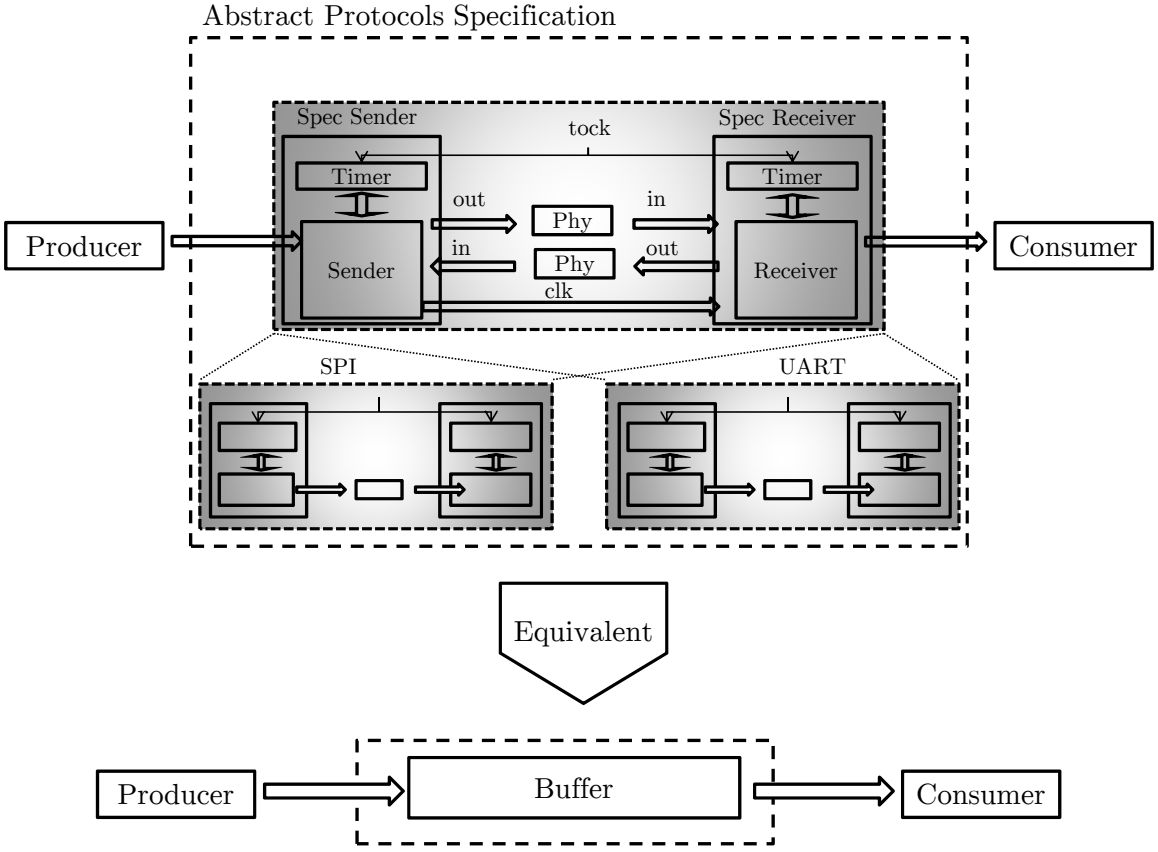


Figure 6.1 Outline of the Abstract Protocol Specification Methodology

Each protocol can be specified at various abstraction levels where functional and real-time details are added incrementally. A specification that captures both the timing and functional aspects was selected. Error checking and fault tolerance specifications were not taken into account when verifying for conformance. More details on the verification process are discussed in Chapter 7.

Two protocols (Uart and Spi) are modelled in Sections 6.4 and 6.5. Whether the specification is abstract or ISA-based, a protocol specification is split into a transmitting process and a receiving process. Once the two are connected using the appropriate interfaces, they form an abstract simplex communication channel. In the Spi protocol, each device can send and receive at the same time giving rise to the possibility of a full-duplex channel. For this reason, an Spi master is modelled only

as a transmitter and the slave as a receiver. This will allow for the same verification techniques to be applied for all the modelled protocols at all abstraction levels.

Finally, the summary and future work are discussed in Sections 6.6 and 6.7, respectively.

6.2 Background

Previous attempts at formal modelling and model-checking of specific communication protocol and hardware constructs are abundant in literature. Though this thesis is mostly dedicated to the inception of a general formal modelling and model-checking framework, a great insight has been gained from analysing the approaches used for the verification of individual protocols. A discussion of the most relevant work is introduced in this section.

Paliwoda and Sanders [72] used CSP in the modelling of the Sliding Window Protocol (SWP). The verification is performed by showing the equivalence of the detailed protocol specification to a higher-level functional specification. It is a good demonstration of the strengths of CSP for specifying a communication protocol through many levels of abstractions. This is because the SWP specification was structured in terms of the simpler ABP protocol, which in turn was expressed in terms of the Stop-and-Wait Protocol (SAWP). The work presented does not show the use of automated model-checking tools for verification purposes. It is clear that as the system grows and more implementation details are added, the process becomes tedious and hence automation becomes necessary.

May et al. [73] have used *Occam* in establishing the correctness of the microcode produced for parts of the *T800* transputer. The approach involved the derivation of an implementation-level microcode program from a high-level specification represented in *Occam* [74]. Those two abstractions are shown to be equivalent through the automated use of a sequence of refinements which apply some semantics preserving transformations of *Occam*.

In a related development Barrett [75] has used CSP in the verification of the *T9000* transputer. In particular, the paper discusses the verification of the Virtual Channel Processor (VCP) and the associated Transputer Link Protocol. Though the work presented was mostly experimental, it was a good practical application of CSP in the verification of hardware designs and communication protocols. It also highlights the lack of industrial-strength tools and a gap between the formal modelling and verification tools and the HDL implementation and simulation tools. However,

since Barrett [75], various attempts at bridging the gap between formal verification formalisms and HDLs have emerged. See [76–78] for more details.

The case studies highlighted so far address specific technologies and protocols and, in some cases, they fine-tune those technologies for the verification of a specific protocol or algorithm. They fall short of presenting a transferable framework for the modelling and verification of communication protocols in general, which is the aim of this thesis. The formal modelling of communication protocols is demonstrated in the remainder of this chapter and the verification aspects are addressed in Chapter 7.

6.3 Preliminaries

In Section 5.3.1 the preliminaries for the whole modelling framework were presented. In this section, the preliminaries defined are only relevant to the modelling of the communication protocols in question. These will be shared amongst the ISA-*Oriented Specification* in Section 6.4 and the *Abstract Specification* in Section 6.5.

$$\textsc{Uart}_{BitRate} = uBR = 2.4 \times 10^6 \text{ bits per second} \tag{6.1}$$

$$startBit = 0 \tag{6.2}$$

$$stopBit = 1 \tag{6.3}$$

$$\textsc{Spi}_{BitRate} = sBR = 4.8 \times 10^6 \text{ bits per second} \tag{6.4}$$

Eqns. 6.1 and 6.4 specify the bit-rate for the Uart and Spi protocols, respectively. Those will be used to calculate the timing information required for managing the STC and SBC units in the ISA-*Oriented Specification*. They will also be used to generate similar but independent timing information for the *abstract* specifications.

6.4 ISA-Oriented Specifications

Schneider [Chapter 5 and 7, 39] first describes *Property-Oriented Specification* which specifies the allowable and prohibited behaviours a process can make on the form of sets of allowable traces in the trace model. These sets of traces are appended with refusal sets in the failure model, which are sets of events the process should not be allowed to refuse.

Specifying a process behaviour on the form of sets of traces and refusals is then extended to incorporate all allowable traces and refusals on the form of a CSP process description. A process description corresponds to a set of traces that a process can

exhibit. This rather philosophical extension of the traces and failures model to include specification processes, as opposed to specification traces is called *Process-Oriented Specification*. These specification processes are then used in the verification of the system using the appropriate refinement model.

However, the system interface described in Section 5.3.5 is specified in terms of an instruction set. Each instruction is a standalone CSP process. A sequence of instructions is also a CSP process. If a specification process is limited to any sequence of this special kind of processes (the instruction set), that specification shall be called an ISA-*Oriented Specification*.

An ISA-*Oriented Specification* of a communication protocol is then seen as a well-defined set of instruction sequences, a sequence of instructions for each subset of protocol specifications. For example, a protocol specification could be split into a transmitter and a receiver. Each could be modelled as a sequence of instructions. The set of the two sequences could then be called: ISA-*Oriented Specification*s of that protocol. This section presents the ISA-*Oriented Specification*s of selected communication protocols. These specifications serve a number of purposes, listed below.

- Demonstration of a higher-level specification abstraction for CSP, which helps bridge the gap between mathematical abstractions such as *Property* and *Process-Oriented Specification* and hardware modelling and implementation abstractions.

- Functional verification of the configurable communication system presented in Chapter 5.

- Conformance verification of the configurable specification methodology proposed in this thesis when verified against the standalone abstract specifications presented in Section 6.5. This conformance verification is demonstrated in Chapter 7.

It must be noted, however, that the ISA-*Oriented Specification* alone is not sufficient to specify a configurable communication interface. In addition to being used to specify arithmetic and state variable manipulation operations (i.e. register file operations), the ISA is also being used to specify configuration and communication specification with the relevant configurable units, which in turn govern the overall interactions of the communication system with the environment. For this reason, an ISA specification is always associated with a system construction specification which

specifies the required configurable units and their interfaces. This association will become clear in the following sections. See Eqns. 6.20 and 6.35 for example.

Lowe [79] discussed an interesting extension to the CSP_M model-checking tool set. Lowe developed a verification framework and an associated tool called "Casper: A Compiler for the Analysis of Security Protocols". *Casper* provided a similar protocol verification framework to the one discussed in this thesis, which was only relevant to the analysis of security protocols. The ISA-*Oriented Specification* methodology demonstrated in this section could serve as a proof of concept for a general specification and verification tool that is capable of verifying the provided ISA specifications of a communication protocol using an underlying CSP engine.

In an ISA-*Oriented Specification* environment for modelling real-time communication protocols, real-time assumptions are necessary. Some assumptions are protocol-specific and will be specified in the relevant sections. Other assumptions could be made about the overall system. For example; the system clock frequency (*isaSysClk*) which specifies the number of *tock* events per second. This was arbitrarily defined as follows:

$$isaSysClk = 19.2 \times 10^6 Hz \tag{6.5}$$

The ISA-*Oriented Specifications* of selected communication protocols follows.

6.4.1 Universal Asynchronous Receiver/Transmitter

The bit-rate (uBR) was defined in Section 6.3. Using uBR, the number of *tock* events per bit ($\text{UART}_{ClkPerBit}$) could be defined as follows:

$$\text{UART}_{ClkPerBit} = uCPB = isaSysClk/uBR \tag{6.6}$$

First, using only an STC unit to manage the timing specifications and a DDC unit for managing the physical interface, the following *unbuffered* ISA *Specifications* of UART could be defined.

6.4.1.1 Unbuffered Receiver

At the start of any ISA-*Oriented Specification*, the BOOT_CODE is used to initialise system state variables and registers. Then the instruction sequence $\text{UART}_{in\text{-}start}$ specifies an infinite receiving process. The process is provided with an I/O port sequence,

which represents an interface to the associated DDC unit. The start-bit is identified by a conditional input of the value zero. Then the timestamp of the start-bit is obtained from the STC unit. It must be noted that the STC unit is by default attached to *port0*. Then the next sampling time is calculated by adding 1.5 bit-time to the time of the start-bit. This is equivalent to sampling the first data bit at the halfway point. Additional timing specifications, such as set-up time and hold time could be added if necessary. Finally, the Bit_{in} instruction sequence is used to initiate the input of DS data bits, which is by default stored in the word register (*wReg0*).

$$\textsc{Uart} = \text{BOOT}_{\text{CODE}}\,; \tag{6.7}$$

$$\textsc{Uart}_{\textit{in-start}}(port0) \tag{6.8}$$

$$\textsc{Uart}_{\textit{in-start}}(rxPort) = \tag{6.9}$$

$$\text{COND}_{\text{IN}}(rxPort, startBit); \tag{6.10}$$

$$\text{STMP}_{\text{READ}}(stcTime, tReg0); \tag{6.11}$$

$$\text{ADD}(tReg0, (uCPB + uCPB/2) \bmod 2^{\text{TS}}, IDelay); \tag{6.12}$$

$$Bit_{in}(\text{DS}) \tag{6.13}$$

$$Bit_{in}(bitCount) = \tag{6.14}$$

$$\left\{\begin{array}{rl} \textsc{Uart}_{\textit{in-stop}} & \text{if bitCount=0} \\ \left.\begin{array}{r} \text{TMD}_{\text{IN}}(tReg0, bReg0); \\ \text{SHIFT}_{\text{R}}(wReg0, bReg0, IDelay); \\ \text{ADD}(tReg0, uCPB \bmod 2^{\text{TS}}, IDelay); \\ Bit_{in}(bitCount - 1) \end{array}\right\} & \text{otherwise} \end{array}\right. \tag{6.15}$$

The Bit_{in} is specified in a hybrid ISA and *Process-Oriented Specification* merely due to the lack of an additional register to hold the data count value. This is done so as to reduce the complexity of the ISA instruction set and the associated register file. Hence the data bit counter is held as a process variable for the Bit_{in} process. While the bit-count is not zero, a timed input is performed the time of which is specified in the time register (*tReg0*). The timed input instruction stores the input data value in the data register (*bReg0*) which is then shifted into the word register (*wReg0*) using the SHIFT_{R} instruction. Finally, a single bit-time (*uCPB*) is added to the time register and the Bit_{in} process is called recursively decreasing the bit-count in each

iteration. When the bit-count reaches zero, the Bit_{in} passes control to the $\textsc{Uart}_{in\text{-}stop}$ process.

$$\textsc{Uart}_{in\text{-}stop} = \mathrm{TMD}_{\mathrm{IN}}(tReg0, bReg0); \tag{6.16}$$

$$\mathrm{TEST}(bReg0, stopBit, \textsc{Uart}_{in\text{-}finish}, \mathrm{STOP}, IDelay) \tag{6.17}$$

$$\textsc{Uart}_{in\text{-}finish} = \mathrm{PUT}(consume, wReg0); \tag{6.18}$$

$$\textsc{Uart}_{in\text{-}start} \tag{6.19}$$

The $\textsc{Uart}_{in\text{-}stop}$ performs the last timed input operation representing the stop-bit which must have the value 1 if the transmitting process was correct. This is ensured using the TEST instruction, which *deadlocks* using the STOP process if the stop-bit is not observed. Otherwise, $\textsc{Uart}_{in\text{-}finish}$ sends the input data which is now stored in *wReg0* to the consume channel, using the PUT instruction and operation continues with the $\textsc{Uart}_{in\text{-}start}$ process.

$$\textsc{Uart}_{\text{ISA-RX}} = ((\textsc{Uart} \underset{aReg}{\parallel} \mathrm{REG}) \underset{aStc}{\parallel} \mathrm{STC}\,0) \underset{aDdc}{\parallel} \mathrm{DDC0} \tag{6.20}$$

A system construction of an ISA-*Oriented Specification* for a Uart receiver is demonstrated in Eqn. 6.20, which synchronises the Uart process with a register file (REG), an STC unit and a DDC unit.

6.4.1.2 Unbuffered Transmitter

$$\textsc{Uart}_{\mathrm{OUT}} = \mathrm{BOOT}_{\mathrm{CODE}}\,; \tag{6.21}$$

$$\textsc{Uart}_{out\text{-}start}(port0) \tag{6.22}$$

$$\textsc{Uart}_{out\text{-}start}(txPort) = \tag{6.23}$$

$$\mathrm{GET}(produce, wReg0); \tag{6.24}$$

$$\mathrm{RAW}_{\mathrm{OUTV}}(txPort, startBit); \tag{6.25}$$

$$\mathrm{STMP}_{\mathrm{READ}}(stcTime, tReg0); \tag{6.26}$$

$$\mathrm{ADD}(tReg0, uCPB \bmod 2^{\mathrm{TS}}, IDelay); \tag{6.27}$$

$$Bit_{out}(\mathrm{DS}) \tag{6.28}$$

The receiver side of a Uart interface starts similarly with the $\mathrm{BOOT}_{\mathrm{CODE}}$, then the $\textsc{Uart}_{out\text{-}start}$ receives the data word to be transmitted from the *produce* channel

(GET), outputs the start-bit ($\mathrm{RAW}_{\mathrm{OUTV}}$) and reads the timestamp of the start-bit ($\mathrm{STMP}_{\mathrm{READ}}$). A single bit-time is added to the time register, followed by the Bit_{out} process:

$$Bit_{out}(bitCount) = \tag{6.29}$$

$$\begin{cases} \textsc{Uart}_{out\text{-}stop} & \text{if bitCount=0} \\ \left.\begin{array}{r} \mathrm{SHIFT_L}(wReg0, bReg0, IDelay); \\ \mathrm{TMD_{OUT}}(tReg0, bReg0); \\ \mathrm{ADD}(tReg0, uCPB \bmod 2^{\mathrm{TS}}, IDelay); \\ Bit_{out}(bitCount-1) \end{array}\right\} & \text{otherwise} \end{cases} \tag{6.30}$$

Bit_{out} outputs *bitCount* data bits from the data word register (*wReg0*). This is achieved through shifting one bit at a time into the data register (*bReg0*), followed by a timed output operation. Finally, one bit-time is added to the time register and the process is repeated. Once all data bits have been output, the process passes control to the $\textsc{Uart}_{out\text{-}stop}$ process.

$$\textsc{Uart}_{out\text{-}stop} = \mathrm{INIT}(bReg0, stopBit, IDelay); \tag{6.31}$$

$$\mathrm{TMD_{OUT}}(tReg0, bReg0); \tag{6.32}$$

$$\mathrm{NOPS}(uCPB/IDelay); \tag{6.33}$$

$$\textsc{Uart}_{out\text{-}start} \tag{6.34}$$

The $\textsc{Uart}_{out\text{-}stop}$ initialises the data register with the value of the stop-bit, then performs a timed output of that value. A mechanism to ensure that the duration of the stop-bit has elapsed before the transmission of the next data item is implemented using the NOP instruction. The process then proceeds to the next data item for transmission.

$$\textsc{Uart}_{\text{ISA-TX}} = ((\textsc{Uart}_{\mathrm{OUT}} \underset{aReg}{\parallel} \mathrm{REG}) \underset{aStc}{\parallel} \mathrm{STC}\,0) \underset{aDdc}{\parallel} \mathrm{DDC0} \tag{6.35}$$

A system construction of the Uart transmitter is demonstrated in Eqn. 6.35.

6.4.1.3 Buffered Receiver

A buffering unit is used to serialise all data bits in a buffered ISA-*Oriented Specification* of the UART protocol. Such a buffered specification is considerably simpler than the unbuffered specification, because all data I/O is managed by the SBC unit itself and the inner recursive processes of Eqns. 6.15 and 6.30 of the unbuffered specifications are eliminated.

This section demonstrates a buffered UART receiver that uses an SBC unit.

$$\text{UART}_{\textit{in-buffered}} = \text{BOOT}_{\text{CODE}}\,; \tag{6.36}$$

$$\text{UART}_{\textit{in-buf-start}}(\textit{port}0) \tag{6.37}$$

$$\text{UART}_{\textit{in-buf-start}}(rxPort) = \text{COND}_{\text{IN}}(rxPort, startBit); \tag{6.38}$$

$$\text{NOPS}(uCPB/IDelay - IDelay); \tag{6.39}$$

$$\text{BUF}_{\text{IN}}(uCPB, wReg0); \tag{6.40}$$

$$\text{PUT}(consume, wReg0); \tag{6.41}$$

$$\text{RAW}_{\text{IN}}(rxPort, bReg0); \tag{6.42}$$

$$\text{TEST}(bReg0, stopBit, \text{UART}_{\textit{in-buf-start}}, \text{STOP}, IDelay) \tag{6.43}$$

In Eqn. 6.38, similar to the unbuffered specification, the data input process is initiated by receiving the start-bit using the COND_{IN} instruction. Then using the NOP instruction, the buffering operation is delayed for the duration of the start-bit. Then the BUF_{IN} instruction is used to configure the buffering unit for an input of a data word using an inter-bit delay of *uCPB*. Finally, the input data is forwarded to the *consume* channel and the stop-bit is observed as before.

$$\text{UART}_{\text{ISA-BUF-RX}} = ((\text{UART}_{\textit{in-buffered}} \underset{aReg}{\|} \text{REG}) \underset{aSbc}{\|} \text{SBC}\,0) \underset{aDdc}{\|} \text{DDC}0 \tag{6.44}$$

A system construction of a buffered UART receiver is demonstrated in Eqn. 6.44.

6.4.1.4 Buffered Transmitter

$$\text{UART}_{\textit{out-buffered}} = \text{BOOT}_{\text{CODE}}\,; \tag{6.45}$$

$$\text{UART}_{\textit{out-buf-start}}(\textit{port}0) \tag{6.46}$$

$$\textsc{Uart}_{\textit{out-buf-start}}(txPort) = \tag{6.47}$$

$$\text{GET}(produce, wReg0); \tag{6.48}$$

$$\text{RAW}_{\text{OUTV}}(txPort, startBit); \tag{6.49}$$

$$\text{NOPS}(uCPB/IDelay - IDelay); \tag{6.50}$$

$$\text{BUF}_{\text{OUT}}(uCPB, wReg0); \tag{6.51}$$

$$\text{RAW}_{\text{OUTV}}(txPort, stopBit); \tag{6.52}$$

$$\text{NOPS}(uCPB/IDelay - IDelay); \tag{6.53}$$

$$\textsc{Uart}_{\textit{out-buf-start}} \tag{6.54}$$

The transmitter side of a buffered ISA-*Oriented Specification* of the Uart protocol is demonstrated in Eqn. 6.45. Once the start-bit is transmitted in a similar manner to the unbuffered specification, the BUF_{OUT} instruction is used to configure the SBC unit for buffered output. BUF_{OUT} instruction does not finish until all data bits have been serialised on the associated DDC unit. Then the stop-bit is transmitted using the RAW_{OUTV} instruction.

$$\textsc{Uart}_{\text{ISA-BUF-TX}} = ((\textsc{Uart}_{\textit{out-buffered}} \underset{aReg}{\parallel} \text{REG}) \underset{aSbc}{\parallel} \text{SBC}\,0) \underset{aDdc}{\parallel} \text{DDC0} \tag{6.55}$$

The system construction of a buffered Uart transmitter then synchronises the $\textsc{Uart}_{\textit{out-buffered}}$ process with a register file, an SBC unit and a DDC unit. This is demonstrated in Eqn. 6.55.

It is plausible that two independent ISA-*Oriented Specifications* are equivalent if both the functional and the performance specifications are equivalent. For example, under certain performance conditions, the buffered ISA-*Oriented Specifications* of a Uart interface could be equivalent to the unbuffered specification. This possible equivalence will be explored in Section 7.9.

6.4.2 Serial Peripheral Interface Bus

The Spi protocol requires additional DDC units. Specifications of the chip select signal are omitted for simplicity. Therefore, only three ports and their associated DDC units are used: system clock (*clk*), Master Output Slave Input (*mosi*) and Master Input Slave Output (*miso*). In theory, both Master and Slave devices are capable of transmitting and receiving data at the same time enabling a full-duplex

communication channel. However, in the specification that follows, only a simplex channel of information is modelled with the *miso* signal not being fully modelled. This should be sufficient to establish the functional and performance correctness of a simplex channel, which is the main focus of Chapter 7.

Prior to defining the ISA-*Oriented Specification* of the SPI protocol, the following definitions are made: $\text{DDC}_{\textit{3-bit}}$ combines all the DDC units required by the SPI protocol into a single 3 bit wide DDC unit with interfaces *port0*, *port1*, and *port2*. $\text{SPI}_{ClkPerBit}$ (also defined as $sCPB$) defines the number of *tock* cycles per SPI data bit. Finally, *aDdc3Bit* defines the synchronisation interface for the $\text{DDC}_{\textit{3-bit}}$ process.

$$\text{SPI}_{ClkPerBit} = sCPB = isaSysClk/sBR \tag{6.56}$$
$$aDdc3Bit = \{\!|ddcCtrl0, isaIn0, isaOut0|\!\} \tag{6.57}$$
$$\cup \{\!|ddcCtrl1, isaIn1, isaOut1|\!\} \tag{6.58}$$
$$\cup \{\!|ddcCtrl2, isaIn2, isaOut2|\!\} \tag{6.59}$$
$$\cup \{\!|stcCtrlDdc|\!\} \tag{6.60}$$
$$\cup \{tock\} \tag{6.61}$$
$$\text{DDC}_{\textit{3-bit}} = (\text{DDC0} \underset{\{tock\}}{\parallel} \text{DDC1}) \underset{\{tock\}}{\parallel} \text{DDC2} \tag{6.62}$$

The ISA-*Oriented Specification*s of an SPI master and an SPI slave are discussed in Sections 6.4.2.1 and 6.4.2.2, respectively.

6.4.2.1 Master

$$\text{SPI}_{master} = \text{BOOT}_{\text{CODE}}\,; \tag{6.63}$$
$$\text{SPI}_{master\text{-}start}(port0, port1, port2) \tag{6.64}$$

$$\text{SPI}_{master\text{-}start}(clk, miso, mosi) = \tag{6.65}$$
$$\text{GET}(produce, wReg0); \tag{6.66}$$
$$\text{RAW}_{\text{OUTV}}(clk, 0); \tag{6.67}$$
$$\text{STMP}_{\text{READ}}(iTime, tReg0); \tag{6.68}$$
$$\text{SPI}_{master\text{-}bit}(miso, mosi); \tag{6.69}$$
$$\text{SPI}_{master\text{-}word}(clk, miso, mosi, \text{DS}-1) \tag{6.70}$$

$$\textsc{Spi}_{\textit{master-bit}}(miso, mosi) = \text{SHIFT}_{\text{L}}(wReg0, bReg0, IDelay); \tag{6.71}$$
$$\text{RAW}_{\text{OUT}}(mosi, bReg0); \tag{6.72}$$
$$\text{ADD}(tReg0, sCPB/2 \bmod 2^{\text{TS}}, IDelay); \tag{6.73}$$
$$\textsc{Spi}_{\textit{clk-high}}\,; \tag{6.74}$$
$$\text{RAW}_{\text{IN}}(miso, bReg0); \tag{6.75}$$
$$\text{ADD}(tReg0, sCPB/2 \bmod 2^{\text{TS}}, IDelay) \tag{6.76}$$

$$\textsc{Spi}_{\textit{clk-high}} = \text{INIT}(bReg0, 1, IDelay); \tag{6.77}$$
$$\text{TMD}_{\text{OUT}}(tReg0, bReg0) \tag{6.78}$$
$$\textsc{Spi}_{\textit{clk-low}} = \text{INIT}(bReg0, 0, IDelay); \tag{6.79}$$
$$\text{TMD}_{\text{OUT}}(tReg0, bReg0) \tag{6.80}$$

A master transmission cycle starts with driving the clock low in Eqn. 6.67. Then the timestamp of the negative edge of the clock is read. Then the first data bit to be transmitted is shifted into the data register, which is then output on the *mosi* port in Eqn. 6.72. Then at the halfway point of the first bit, the clock is driven high in Eqn. 6.74. Finally, the data bit received on the *miso* port is read in Eqn. 6.75.

The first cycle of a word transmission differs from subsequent cycles in the fact that the start of the clock cycle is not bound by any timing conditions, hence the use of the RAW_{OUTV} for initiating that cycle. Subsequent cycles start by a TMD_{OUT} instead, as can be seen in Eqn. 6.81.

$$\textsc{Spi}_{\textit{master-word}}(clk, miso, mosi, count) = \tag{6.81}$$
$$\begin{cases} \textsc{Spi}_{\textit{clk-low}}\,; \textsc{Spi}_{\textit{master-bit}}(miso, mosi); \\ \textsc{Spi}_{\textit{master-word}}(clk, miso, mosi, count - 1) \end{cases} \quad \text{if count} \neq 0$$
$$\textsc{Spi}_{\textit{master-cycle}}\,; \textsc{Spi}_{\textit{master-start}}(clk, miso, mosi) \quad \text{otherwise} \tag{6.82}$$

$$\textsc{Spi}_{\textit{master-cycle}} = \textsc{Spi}_{\textit{clk-low}}\,; \tag{6.83}$$
$$\text{ADD}(tReg0, sCBP/2 \bmod 2^{\text{TS}}, IDelay); \tag{6.84}$$
$$\textsc{Spi}_{\textit{clk-high}}\,; \tag{6.85}$$
$$\text{ADD}(tReg0, sCBP/2 \bmod 2^{\text{TS}}, IDelay); \tag{6.86}$$
$$\textsc{Spi}_{\textit{clk-high}} \tag{6.87}$$

Finally, once the data counter (*count*) reaches zero, then the specification allows for an empty bus cycle before the transmission of the next data word can begin.

$$\textsc{Spi}_{\text{ISA-MASTER}} = ((\textsc{Spi}_{master} \underset{aReg}{\parallel} \text{REG}) \underset{aStc}{\parallel} \text{STC}\,0) \underset{aDdc}{\parallel} \text{DDC}_{3\text{-}bit} \tag{6.88}$$

A system construction of an Spi master device is defined in Eqn. 6.88.

6.4.2.2 Slave

$$\textsc{Spi}_{slave} = \text{BOOT}_{\text{CODE}}\,; \tag{6.89}$$

$$\textsc{Spi}_{slave\text{-}word}(port0, port1, port2, \text{DS}) \tag{6.90}$$

An ISA-*Oriented Specification* of a slave is much simpler than the master, because the slave is not responsible for driving the clock signal.

$$\textsc{Spi}_{slave\text{-}word}(clk, miso, mosi, count) = \tag{6.91}$$

$$\begin{cases} \left.\begin{array}{l} \text{COND}_{\text{IN}}(clk, 0); \\ \text{RAW}_{\text{OUTV}}(miso, 0); \\ \text{COND}_{\text{IN}}(clk, 1); \\ \text{RAW}_{\text{IN}}(mosi, bReg0); \\ \text{SHIFT}_{\text{R}}(wReg0, bReg0, IDelay); \\ \textsc{Spi}_{slave\text{-}word}(clk, miso, mosi, count - 1) \end{array}\right\} & \text{if count} \neq 0 \\ \left.\begin{array}{l} \text{PUT}(consume, wReg0); \\ \text{COND}_{\text{IN}}(clk, 0); \\ \text{COND}_{\text{IN}}(clk, 1); \\ \textsc{Spi}_{slave\text{-}word}(clk, miso, mosi, \text{DS}) \end{array}\right\} & \text{otherwise} \end{cases} \tag{6.92}$$

A data cycle starts by observing the negative edge of the clock. Subsequently, the data is typically output on the *miso* port in a full-duplex specification. For simplicity the slave transmits zero in the specifications above. Then, once the positive edge of the clock is observed, the *mosi* port is read and the received data bit is shifted into the word register. Once a full data word has been received, the word is forwarded along the *consume* channel using the PUT instruction. Finally, an empty bus cycle is observed and the process proceeds to the reception of the next data word.

$$\text{SPI}_{\text{ISA-SLAVE}} = ((\text{SPI}_{slave} \underset{aReg}{\parallel} \text{REG}) \underset{aStc}{\parallel} \text{STC}\,0) \underset{aDdc}{\parallel} \text{DDC}_{3\text{-}bit} \tag{6.93}$$

A system construction of an SPI slave device is defined in Eqn. 6.93.

6.4.3 Results

The ISA-*Oriented Specification* technique reduces the complexity of the analysed configurable system considerably. The number of all possible sequences of configurations is reduced to those of essential interest. These are the configurations corresponding to the selected functional and performance requirements of the protocols in question. Table 6.1 summarises the metrics for all ISA *Specifications* presented in this chapter.

DataSize		1	2	3	4	5
$\text{UART}_{\text{ISA-RX}}$	States	5122	12034	28226	65346	149058
	Trans.	8658	20370	47890	111122	253970
$\text{UART}_{\text{ISA-TX}}$	States	541	1231	2611	5371	10891
	Trans.	638	1442	3050	6266	12698
$\text{UART}_{\text{ISA-BUF-RX}}$	States	3114	7946	22794	73226	257034
	Trans.	5130	13082	37626	121274	426810
$\text{UART}_{\text{ISA-BUF-TX}}$	States	673	1723	4335	10967	28583
	Trans.	770	2030	5302	14118	39366
$\text{SPI}_{\text{ISA-MASTER}}$	States	1914	9914	6058	47450	49914
	Trans.	3950	20430	12522	97758	102950
$\text{SPI}_{\text{ISA-SLAVE}}$	States	9762	34306	99074	259970	644482
	Trans.	26106	90618	260602	682490	1690106

Table 6.1 Metrics of ISA-Oriented Specifications by Varying *DataSize*

Table 6.1 shows how the models grow as the size of the *DataWord* set (DS) grows while fixing the size of the *TimeWord* set to an arbitrary value (TS = 3). The table shows the metrics of the fully constructed systems in Eqns.6.20, 6.35, 6.44, 6.55, 6.88 and 6.93 and not only the ISA-*Oriented Specification*s.

If the metrics of the UART receiver ($\text{UART}_{\text{ISA-RX}}$) in Table 6.1 are compared to the unconfigured one bit system ($\text{SYSTEM}_{one\text{-}bit}$) in Table 5.1, it is evident that the complexity metrics are reduced by a factor of 400. Another interesting conclusion from Table 6.1 is the marginal increase in complexity that the use of the SBC unit has. It increased the metrics of the $\text{UART}_{\text{ISA-BUF-TX}}$ compared to the metrics of $\text{UART}_{\text{ISA-TX}}$

slightly. The increase in not linear. For example, at DS = 1, the buffered metrics were 1.2 times the unbuffered ones. When DS = 5, the factor grows to 3.1.

In any case, because the complexity of the configured system is significantly reduced compared to the unconfigured system, a complete complexity analysis similar to Section 5.5 was not necessary.

6.5 Abstract Specifications

Abstract protocol specifications are required mainly for conformance verification: proof that the ISA-*Oriented Specification*s described is Section 6.4 conform to a specification which is independent from the technical details of the configurable system. Such independent specifications must not rely on any configurable units to establish the specifications of the communication protocol.

To simplify the specification process, a simple timer and a simple physical interface are first defined. Then the abstract specifications of both the Uart and Spi protocols are provided.

The shared preliminaries which were presented in Section 5.3 shall be made available to this specification process. In addition, the preliminaries provided in Section 6.3 shall also be available here.

One additional preliminary definition that is only specific to abstract protocol specifications is sTS: the time unit size in bits. The value of sTS depends on the target protocols and the duration of the longest transaction. For the verification of the protocols presented in this chapter using the bit-rates defined earlier, $sTS = 3$ was sufficient.

6.5.1 Abstract Timer

Tock-CSP is deployed in a simplistic manner in order to provide timing information at the protocol specification level: only one process is responsible for executing *tock* events (called *SpecTimer*). All other processes in the abstract specifications synchronise with the *SpecTimer* process on other events such as *reset*, *set* and *trigger* to establish timing information, but not on *tock*. This results in an implicit synchronisation on *tock*. Consequently, the required timing information for other specification processes is implicitly provided.

The processes and data-types used in the definition of the *SpecTimer* are:

- *sTime*, or specification time, which is a data-type used to specify events communicated by the *SpecTimer* process;
- *sTData*, or specification timer data set, which is the set of values that can be used to represent time; and
- *SpecTimer*: which is the top-level timer process.

Below is the CSP description of the *SpecTimer*, its subprocesses and related data-types:

$$sTData = \{0 \,..\, (2^{sTS} - 1)\} \tag{6.94}$$

$$datatype\, sTime = trigger \mid set.sTData \mid reset \tag{6.95}$$

$$SpecTimer(timer) = STimer(0, timer) \tag{6.96}$$

$$STimer(t,\, timer) = tock \rightarrow STimer(inc(t, 2^{sTS}),\, timer) \tag{6.97}$$

$$\Box\, timer?reset \rightarrow STimer(0,\, timer) \tag{6.98}$$

$$\Box\, timer.set?check \rightarrow TCheck(t,\, check,\, timer) \tag{6.99}$$

$$TCheck(t,\, c,\, timer) = \tag{6.100}$$

$$\begin{cases} timer!trigger \rightarrow STimer(t, timer) & \text{if } t = c \\ tock \rightarrow TCheck(inc(t, 2^{sTS}),\, c,\, timer) & \text{otherwise} \end{cases} \tag{6.101}$$

At system level, any component or process which uses the *SpecTimer* must synchronise with other components over *tock*, because they do not share the same *SpecTimer* process. This is due to the fact that there is not a global notion of time in the current CSP models. By synchronising all system components using *tock* (or other events that are directly generated from *tock* and available to all components), a global notion of time is established. This results in the following conundrum: the UART protocol discussed in this chapter is an asynchronous protocol, and in theory there is no need for exchanging timing information between the transmitter and the receiver, both only have to agree on the bit-rate. However, when modelling this aspect using tock-CSP and because there is no global notion of time, each process or subsystem that has real-time aspects cannot be modelled in a completely asynchronous manner from other system components. They all have to synchronise on *tock* events, whether explicitly using the actual *tock* event in the synchronisation alphabet, or implicitly synchronising on another event which is relative to *tock* such as the *set* and *trigger* events above.

Future implementations of tock-CSP verification models might have a global notion of time, in which case *tock* events become global to all processes. In that case explicit synchronisation on timing events becomes unnecessary.

6.5.2 Universal Asynchronous Receiver/Transmitter

Different abstraction levels were investigated where timing and functional details were added incrementally. The abstraction demonstrated in this section has been selected in such a way that all timing information is present.

The number of *tock* events per bit ($uCPB$) defined in Section 6.4.1 is used in this section to trigger events in the *SpecTimer* process. This is due to the conundrum described earlier, where all system components, whether specified using the ISA-*Oriented Specification* or abstractly as in this section must synchronise on *tock* events.

6.5.2.1 Receiver

The abstract specification of a UART receiver (USR) is modelled as a sequence of events on a CSP channel called UART_{in}, which represents the physical interface connection. The receive process is then defined as an infinite input of data words.

$$channel \quad rxTimer : sTime \tag{6.102}$$

$$channel \quad \text{UART}_{in} : Data \tag{6.103}$$

$$\text{USR}(timer) = \text{UART}_{in}?startBit \rightarrow timer!reset \rightarrow \tag{6.104}$$

$$USTi(\text{DS}, 0, ((uCPB \times 3)/2 \bmod 2^{sTS}, timer) \tag{6.105}$$

$$USRi(c, w, t, tr) = \tag{6.106}$$

$$\begin{cases} \left.\begin{array}{l} tr.set!t \rightarrow consume!w \rightarrow tr?trigger \rightarrow \\ \text{UART}_{in}?stopBit \rightarrow \text{USR}(tr) \end{array}\right\} & \text{if } c = 0 \\ \left.\begin{array}{l} tr.set!t \rightarrow tr?trigger \rightarrow \text{UART}_{in}?b \rightarrow \\ USRi(c-1, sright(b, w), (t + uCPB) \bmod 2^{sTS}, tr) \end{array}\right\} & \text{otherwise} \end{cases} \tag{6.107}$$

The receiver waits for the start-bit to appear ($\text{UART}_{in}?low$), at which point it sets the *SpecTimer* with duration $1.5 \times uCPB$. This is done in order to skip the start-bit and sample the first data bit at the midway point. Once the first data bit is input, the receiver sets the timer to trigger after $uCPB$ *tock*s and inputs the next data bit. The process is repeated until the number of bits input is equal to DS. Finally, the received data word is forwarded to the *consume* channel before the stop-bit is observed. It

is essential that forwarding the data word happens before the stop-bit is observed to enforce this forwarding action to happen in a specific time-frame. Otherwise, the specification would allow for an arbitrary number of clock cycles to elapse before the forwarding happens, in which case the receiver would miss any subsequent data words sent by the transmitter.

$$\textsc{Uart}_{\text{SPEC-RX}} = \tag{6.108}$$

$$(\text{USR}(rxTimer) \underset{\{|rxTimer|\}}{\parallel} SpecTimer(rxTimer)) \setminus \{|rxTimer|\} \tag{6.109}$$

Eqn. 6.108 shows the construction of the top-level Uart receiver ($\textsc{Uart}_{\text{SPEC-RX}}$) by synchronising the USR process with a *SpecTimer* process over the timer communication channel ($rxTimer$).

6.5.2.2 Transmitter

Similarly, the transmitter (UST) is modelled as a sequence of events on a CSP channel called $\textsc{Uart}_{out}$, which represents the connection to the physical interface. The transmitter process is then defined as an infinite transmission of data words.

$$channel \quad txTimer : sTime \tag{6.110}$$

$$channel \quad \textsc{Uart}_{out} : Data \tag{6.111}$$

$$uCPB = \textsc{Uart}_{ClkPerBit} \tag{6.112}$$

$$\text{UST}(timer) = produce?word \rightarrow \textsc{Uart}_{out}!startBit \rightarrow timer!reset \rightarrow \tag{6.113}$$

$$USTout(\text{DS}, word, uCPB, timer) \tag{6.114}$$

$$USTout(c, w, t, tr) = \tag{6.115}$$

$$\begin{cases} \left.\begin{array}{l} tr.set!t \rightarrow tr?trigger \rightarrow \textsc{Uart}_{out}!stopBit \rightarrow \\ tr.set!((t + uCPB + 1) \bmod 2^{sTS}) \,;\, \text{UST}(tr) \end{array}\right\} & \text{if } c = 0 \\ \left.\begin{array}{l} tr.set!t \rightarrow tr?trigger \rightarrow \textsc{Uart}_{out}!(w \bmod 2) \rightarrow \\ USTout(c-1, w/2, (t + uCPB) \bmod 2^{sTS}, tr) \end{array}\right\} & \text{otherwise} \end{cases} \tag{6.116}$$

The transmission of a data word in Eqn. 6.113 starts with the start-bit, followed by a timer *reset* command ($timer!reset$). Using the *SpecTimer* events *set* and *trigger*, all data bits are output serially ($\textsc{Uart}_{out}!(w \bmod 2)$), which are separated by a number of *tock* events equivalent to *uCPB*. Finally, the transmission ends in Eqn. 6.116 with the stop-bit ($\textsc{Uart}_{out}!high$).

$$\textsc{Uart}_{\text{SPEC-TX}} = \tag{6.117}$$

$$(\text{UST}(txTimer) \underset{\{|txTimer|\}}{\parallel} SpecTimer(txTimer)) \setminus \{|txTimer|\} \tag{6.118}$$

Eqn. 6.117 shows the construction of the top-level Uart transmitter by synchronising the UST process with a *SpecTimer* process over the timer communication channel ($txTimer$).

Figure 6.2 shows the waveform of the $\textsc{Uart}_{\text{SPEC-RX}}$ process in Eqn. 6.108. Similarly, Figure 6.3 shows the waveform of the $\textsc{Uart}_{\text{SPEC-TX}}$ process in Eqn. 6.117. The waveforms show the difference in timing between the Transmitter and Receiver since the receiver samples at half the bit-time, while the transmitter triggers the data at the start of the bit-time. The waveforms also show that the data values being transmitted by $\textsc{Uart}_{\text{SPEC-TX}}$ on the *produce* channel are indeed being received by $\textsc{Uart}_{\text{SPEC-RX}}$ on the *consume* channel. Finally, for demonstration purposes, the first data item to be transmitted was selected to be *0b01*. This means the *DataWord* transmission starts by the start-bit of the Uart protocol (0), followed by the first bit of the *DataWord* (1), second bit (0), and finally, the stop-bit of the protocol (1).

The waveforms were automatically generated using the *Tock-CSP Waveform Generator* developed as part of this framework and discussed briefly in Section 7.8.1.1. More details are available in Appendix B.

The traces used by the *Tock-CSP Waveform Generator* were obtained by adjusting the *Buffer′* process in Assertion 7.86 to STOP after buffering 3 data items. This caused the assertion to fail and a trace to be generated for demonstration purposes. The traces were generated using the following parameters: $uCPB = 4$, $sTS = 4$ and DS = 2.

6.5.3 Serial Peripheral Interface Bus

The Spi is a synchronous protocol in which a clock signal is exchanged. Abstractly modelling Spi fits well into the simplistic tock-CSP modelling approach involving the *SpecTimer* discussed in Section 6.3. The exchanged clock signal between the master device and slave devices is generated using *tock* events, hence there is no need for explicit synchronisation between different devices on *tock* events, which was the case in the Uart protocol.

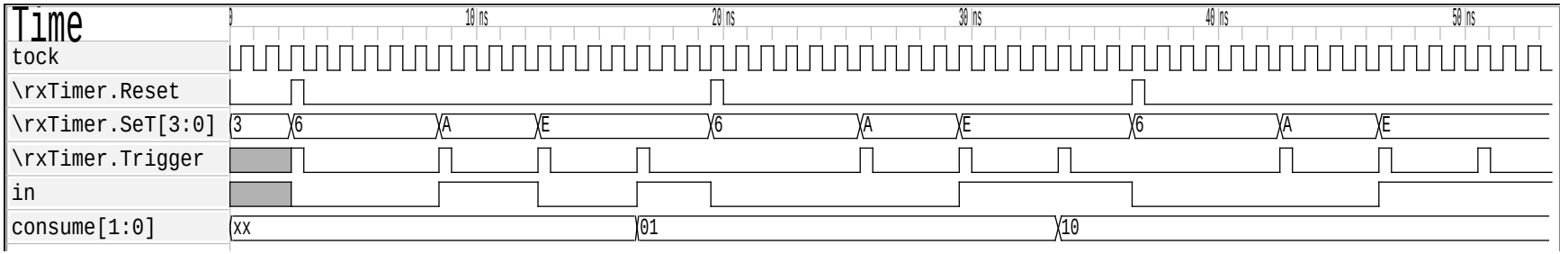


Figure 6.2 Waveform of $\text{UART}_{\text{SPEC-RX}}$ Process

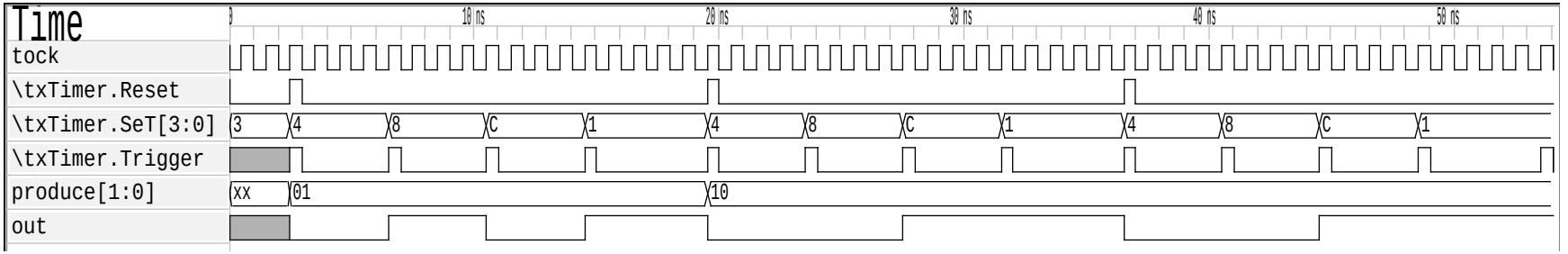


Figure 6.3 Waveform of $\text{UART}_{\text{SPEC-TX}}$ Process

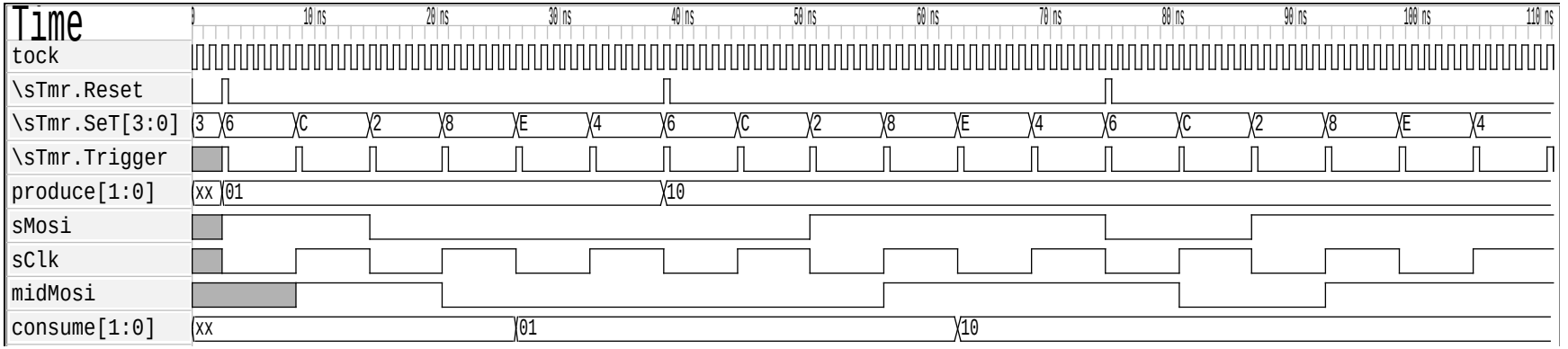


Figure 6.4 Waveform of $\text{SPI}_{\text{SPEC-MASTER}}$ and $\text{SPI}_{\text{SPEC-SLAVE}}$ in Synchronisation

6.5.3.1 Master

For simplicity, the number of *tock* events per bus clock cycle ($sCPB$) specified in Section 6.4.2 will be used in this section.

$$channel \quad \text{SPI}_{SpecClk}, \text{SPI}_{SpecMiso}, \text{SPI}_{SpecMosi} : Data \tag{6.119}$$

$$channel \quad \text{SPI}_{TM} : sTime \tag{6.120}$$

$$sSClk = \text{SPI}_{SpecClk} \tag{6.121}$$

$$sSMiso = \text{SPI}_{SpecMiso} \tag{6.122}$$

$$sSMosi = \text{SPI}_{SpecMosi} \tag{6.123}$$

$$\text{SSM}(timer) = produce?data \rightarrow timer!reset \rightarrow \tag{6.124}$$

$$SSMw(data, 0, \text{DS}, timer, 0) \tag{6.125}$$

When the master (SSM) receives a data word for transmission from the *produce* virtual channel, it initiates the SPI data word transmission process ($SSMw$).

$$SSMw(oD, iD, c, tr, t) = \begin{cases} SSMfinish(tr, t) & \text{if c} = 0 \\ SSMb(oD, iD, c, tr, t) & \text{otherwise} \end{cases} \tag{6.126}$$

$$SSMb(oD, iD, c, tr, t) = \tag{6.127}$$

$$sSClk!low \rightarrow sSMosi!(oD \bmod 2) \rightarrow \tag{6.128}$$

$$tr.set!(add(t, sCPB/2, 2^{sTS})) \rightarrow tr?trigger \rightarrow \tag{6.129}$$

$$sSClk!high \rightarrow sSMiso?b \rightarrow \tag{6.130}$$

$$tr.set!(add(t, sCPB, 2^{sTS})) \rightarrow tr!trigger \rightarrow \tag{6.131}$$

$$SSMw(oD/2, sright(iD, b), c - 1, tr, add(t, sCPB, 2^{sTS})) \tag{6.132}$$

$$SSMfinish(tr, t) = \tag{6.133}$$

$$sSClk!low \rightarrow tr.set!(add(t, sCPB/2, 2^{sTS})) \rightarrow \tag{6.134}$$

$$tr?trigger \rightarrow sSClk!high \rightarrow \tag{6.135}$$

$$tr.set!(add(t, sCPB, 2^{sTS})) \rightarrow tr!trigger \rightarrow \text{SSM}(tr) \tag{6.136}$$

The *SSMb* starts the bus cycle by driving the system clock low ($sSClk!low$), followed by transmitting the data bit ($sSMosi!(oD \bmod 2)$). Then, using the interface to the *SpecTimer*, the second phase of the cycle is started in Eqn. 6.129.

Once the timer triggers, the clock is driven high ($sSClk!high$) and the data input line is sampled ($sSMiso?b$). The timer is used again to signal the end of the cycle ($tr.set!(add(t, sCPB, 2^{sTS}))$) before the next cycle can start. This is repeated for the number of bits per data word (DS). Finally, an empty bus cycle is allowed before the transmission of the next data word can begin.

$$\textsc{Spi}_{\text{SPEC-MASTER}} = (\text{SSM}(\textsc{Spi}_{TM}) \underset{\{\!|\,\text{SPI}_{TM}\,|\!\}}{\|} SpecTimer(\textsc{Spi}_{TM})) \setminus \{\!|\,\textsc{Spi}_{TM}\,|\!\} \quad (6.137)$$

Eqn. 6.137 shows the construction of the top-level Spi master ($\textsc{Spi}_{\text{SPEC-MASTER}}$) by synchronising the SSM process with a *SpecTimer* process over the timer communication channel ($\textsc{Spi}_{TM}$).

6.5.3.2 Slave

The slave bus cycle is simpler, because timing information is explicitly exchanged using the *sSClk* signal and there is no need for the slave to maintain its own timing.

$$SSSw(c, oD, iD) = \quad (6.138)$$

$$\begin{cases} \left.\begin{array}{l} consume!iData \rightarrow sSClk?low \rightarrow \\ sSClk?high \rightarrow \textsc{Spi}_{\text{SPEC-SLAVE}} \end{array}\right\} & \text{if c} = 0 \\ \left.\begin{array}{l} sSClk?low \rightarrow sSMiso!(oD \bmod 2) \rightarrow \\ sSClk?high \rightarrow sSMosi?b \rightarrow \\ SSSw(c-1, oD/2, sright(iD, b)) \end{array}\right\} & \text{otherwise} \end{cases} \quad (6.139)$$

Using the same bus signals declared earlier (*sSClk*, *sSMiso* and *sSMosi*), the cycle starts by observing a falling edge of the clock signal ($sSClk?low$), followed by an output of the first data bit on the *sSMiso* signal. Then, at the rising edge of the clock ($sSClk?high$), the slave inputs the data bit arriving from the master ($sSMosi?b$). The process is repeated for all data bits in the transmission (DS times). Finally, when a full data word is received, the slave outputs that word on the virtual *consume* channel.

$$\textsc{Spi}_{\text{SPEC-SLAVE}} = SSSw(\text{DS}, 0, 0) \quad (6.140)$$

Eqn. 6.140 shows the top-level Spi slave process ($\textsc{Spi}_{\text{SPEC-SLAVE}}$) in which no *SpecTimer* process is needed.

Figure 6.4 shows a waveform synchronising SPI$_{\text{SPEC-MASTER}}$ in Eqn. 6.137 and SPI$_{\text{SPEC-SLAVE}}$ in Eqn. 6.140. The figure shows the transmission of three consecutive data words *0b01*, *0b10* and *0b10*. The construction of a communication pipeline involves the use of an intermediate process to hold the MOSI signal values and decouple the master from the slave. This is depicted in Figure 6.4 by the presence of two MOSI signals, *sMosi* triggered by the master and *midMosi* inputted by the slave. As the figure shows, there is a time lag between the two signals and hence the need for the decoupling process. Unlike the UART protocol, there is a need for only one *SpecTimer* process, since the timing in the SPI$_{\text{SPEC-SLAVE}}$ process is being managed by the exchanged *sSClk* signal. The parameters used for this demo were: $sCPB = 12$, $sTS = 4$ and DS = 2.

Finally, the exact signal names on the waveforms in Figures 6.2, 6.3 and 6.4 might differ slightly from those in the associated CSP processes described earlier, since the waveforms have been automatically generated from the CSP_m scripts.

6.6 Summary and Conclusion

The selected communication protocol specifications presented in this chapter serve a number of objectives. First and foremost, the protocol specifications presented in Section 6.4 demonstrate the ISA-*Oriented Specification* technique, which is based on the configurable system presented in Chapter 5 and its associated instruction set. It shows that a single protocol could be specified in a number of ways that vary in terms of the internal implementation details.

However, because the specifications presented in Section 6.4 include real-time requirements, two functionally equivalent specifications might differ from a performance perspective, as will be demonstrated in Sections 7.8 and 7.9. Using different configurable units in two independent specifications to specify the same protocol as demonstrated in Sections 6.4.1.1 and 6.4.1.3 shows how advanced constructs (such as the SBC unit) simplify the associated ISA-*Oriented Specification*s. Complex configurable units with simplified ISA-based interfaces provide powerful specification, exploration, and verification tools for designers of communication protocols. The support of an ISA interface for formal verification would be highly valuable for bridging the gap between the formal analysis community and the hardware design and implementation community.

The simplistic functional and real-time protocol specification technique discussed in Section 6.5 provides another outlook on the application of tock-CSP in the mod-

elling of a real-time system. The abstract specifications presented should be essential for verifying the correctness of the associated ISA-*Oriented Specification*s. Section 6.5 also highlighted the difference between modelling synchronous and asynchronous interfaces in tock-CSP, where in the latter both the transmitter and receiver have to synchronise on the built-in system timing event (*tock*), while this is not the case for synchronous interfaces, because an explicit clock signal is exchanged.

6.7 Future Work

Future work includes modelling all the technical details of the selected protocols and the investigation of the functional, performance and complexity issues that such details might introduce. Including the details of the idle state where communication interfaces are not engaged in transmission or reception is expected to have implications on the ISA-*Oriented Specification* instruction set, as well as the verification approaches used in Chapter 7. Also, adding the full details of the Spi protocol would subsequently require the construction and verification of a complex bidirectional communication channel.

It would also be interesting to investigate other communication protocols such as an asynchronous handshake protocol and the additional requirements those protocols would add to the modelled configurable units.

Finally, the ISA-*Oriented Specification*s could be further developed into an independent verification library. Such a library would have an ISA specification and verification interface and would rely on an underlying tock-CSP engine similar to that which has been presented in Chapter 5. Furthermore, it would be very interesting if the underlying tock-CSP engine was automatically translated into an HDL representation similar to the work done by Ostroumov et al. [78] and Treharne et al. [76]. Doing so would provide two cornerstones towards the grand challenge of building a formally verified stack as discussed by Hoare et al. [46] and Moore [80].

CHAPTER 7

Formal Verification using Refinement Model-Checking

7.1 Motivation

This chapter presents the formal verification of the concrete configurable communication system modelled in Chapter 5, using the relevant model-checking techniques, tools and abstractions. Through the detailed analysis of the modelled system and protocols, a great level of understanding of the relationship between the intended specifications and the formal model is established. However, it is also of great importance that the intended functions of the concrete configurable system are rigorously verified through model-checking each of the various models against abstract specifications of each modelled aspect or property.

The formal verification process progresses from basic timing and functional checks of the individual configurable units in Section 7.3 through to basic verification of the different protocol ISA-*Oriented Specifications* and abstract protocol specifications in Sections 7.4 and 7.5, respectively. Then, in Sections 7.6 and 7.7, complex communication channels are constructed, using a transmitter and a receiver of a communication protocol specified at the same abstraction level (i.e. ISA vs. abstract). Those complex constructions are model-checked and proved to be *equivalent* to a simple asynchronous abstraction of a communication protocol: a single-entry buffer.

In Section 7.8, the performance specifications of the complex channel constructions are verified. The abstract view of a communication protocol as an asynchronous single-entry buffer is modified to add clocking information. The intended latencies of the communication protocols are specified using this synchronous buffer. Then, in

Section 7.8.1, the latency specifications of the complex channels are checked against the intended latencies of the respective protocols. Section 7.8.1 also explores the possible use of the ISA-*Oriented Specification* methodology for providing the required latency specifications. The section briefly discusses the *Tock-CSP Waveform Generator*, which was instrumental in evaluating the performance specifications of the complex channels.

Another important performance aspect is the throughput of the constructed complex channels. This could be useful for checking the overhead that a specification style could introduce on the set-up of the transmission/reception process. Such an overhead might not be apparent in the latency specification. In addition, in some situations there might be a level of non-determinism in the latency specification, in which case the throughput specification could be used to assess the performance of the channel. Section 7.8.2 model-checks the throughput specifications of the complex communication channels.

In addition to independently verifying selected functional and performance aspects of the different specification styles independently, the conformance of the ISA-*Oriented Specification*s to the abstract protocol specifications is further analysed in Section 7.9. Different specification styles are multiplexed into constructing a set of complex conformance channels, which are then functionally verified in a way similar to the verification of the uniform complex channels demonstrated in Sections 7.6 and 7.7.

Finally, Sections 7.10 and 7.11 present a brief summary and the possible future work.

7.2 Background

The CSP model-checking framework provides a robust platform for verifying different aspects of a design progressively. It allows the specification and subsequent verification of a model at various abstraction levels: a model could be specified with minimum details; such specifications often serve as a top-level specification. Then the models evolve by adding various *implementation* details. Automatic verification through model-checking could check that the more complex models with added details still conform to the top-level specification. This is done by establishing a suitable refinement relation between the different models.

7.2.1 Refinement Models

The refinement models are algorithms and a set of laws that govern the checks performed in a model-checking framework. Each model addresses one or more refinement relations between two processes. This section describes selected refinement models, which were instrumental in the evaluation process of the configurable communication system.

7.2.1.1 Traces Refinement Model ($\mathcal{T}$)

In this model, each process is represented as a finite set of traces of events the process can execute. These sets of traces represent the behaviour of the process up to a certain point in time. This is typically written as a comma-separated sequence of events enclosed in angle brackets. For example:

- $traces(\text{STOP}) = \{\langle\rangle\}$ where $\langle\rangle$ is an empty sequence of events;
- $traces(a \rightarrow b \rightarrow \text{STOP}) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$ which means at the start, the process may not have communicated any events yet, may have communicated event a only, or may have communicated event a followed by event b; and
- $traces(a \rightarrow \text{STOP} \ \Box \ b \rightarrow \text{STOP}) = \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$ where the process has the choice of either executing event a or event b, or it could also not have communicated any event at all yet.

A model-checker can determine in advance the set of all possible traces of a process by automatically analysing the specifications of that process. The relationship between different processes with respect to the traces they can execute is called trace refinement. Hence, for checking that a process (Q) trace refines another process (P), the set of traces that process Q can execute should be a subset of all possible traces allowed by process P. Formally, this is written as:

$$\text{P} \sqsubseteq_{\text{T}} \text{Q} \Leftrightarrow traces(\text{P}) \supseteq traces(\text{Q}) \tag{7.1}$$

A trace refinement specification is said to be a safety specification, because it ensures that a process never executes events that it is not allowed to execute. On the other hand, the model cannot guarantee that any favourable events will happen. This can be achieved using the Stable Failures Model, which is discussed next. The Traces Refinement Model is well described by Brookes et al. [37] and Hoare [35].

7.2.1.2 Stable Failures Model ($\mathcal{F}$)

In addition to the traces that a process can execute, this model takes into account the set of events that a process can refuse (X) after executing a specific trace (tr) and reaching a stable state. A process is termed stable when it can no longer execute internal events (τ). The set of events X is called a stable refusals set.

The following rules are some of the algebraic laws used for analysing processes for their refusal sets.

- $\forall \mathrm{P} \Rightarrow \{\} \in refusals(\mathrm{P})$: the empty set is a refusal set of all possible processes. An empty refusal set simply means that a process does not refuse any event.
- $\Sigma \in refusals(\mathrm{STOP})$: the set of all possible events (Σ) is a refusal set for the STOP process. This simply means that the STOP process refuses to execute any event.
- $refusals(\mathrm{P} \,\Box\, \mathrm{Q}) \neq refusals(\mathrm{P} \sqcap \mathrm{Q})$: refusals can distinguish between internal and external choices. This is because an internal choice can decide to execute one event and refuse the other, while an external choice must offer both and it is up to the environment to decide which event to execute.

The tuple (tr, X) constituting the refusal set (X) of a process after executing a specific trace (tr) is called a stable failure (or simply *failure*). A process Q refines another process P under the Stable Failures Model, when the set of all stable failures of the process Q is a subset of all possible stable failures of the process P. This is written as:

$$\mathrm{P} \sqsubseteq_{\mathrm{F}} \mathrm{Q} \Leftrightarrow failures(\mathrm{P}) \supseteq failures(\mathrm{Q}) \tag{7.2}$$

A stable failures refinement specification is said to be a liveness specification, in addition to it being a safety specification, since it not only specifies that nothing illegal can happen, it also specifies that something favourable will happen. This model is useful for verifying the refinement relation between specification and implementation processes. It is also useful for verifying deadlock freedom of processes. When this property is performed abstractly without a reference to another *specification* process, then it ensures that the process must always be able to perform externally visible events (i.e. be live and not deadlocked). This model is also discussed in detail by Hoare [35].

7.2.1.3 Failures/Divergences Model (*FD*)

The Failures/Divergences Model is similar to the Stable Failures Model discussed earlier, but adds the notion of divergence or livelock: the possibility of the process to execute an infinite sequence of internal events (τ). A divergent process cannot execute any external events. Such a process is said to be livelocked or divergent.

Formally, a divergence trace is a finite trace of events during or after which the process can perform an infinite sequence of consecutive internal events.

The property that a process P is Failures/Divergences refined by the process Q is written as:

$$\mathrm{P} \sqsubseteq_{\mathrm{FD}} \mathrm{Q} \Leftrightarrow failures(\mathrm{P}) \supseteq failures(\mathrm{Q}) \tag{7.3}$$

$$\wedge\, divergences(\mathrm{P}) \supseteq divergences(\mathrm{Q}) \tag{7.4}$$

More details about this model are discussed by Brookes and Roscoe [81].

The Failures/Divergences Model is considered to be more powerful than the Traces Refinement Model or the Stable Failures Model because it can check a number of properties: livelock freedom, deadlock freedom, as well as checking the safety and liveness properties of a process if it is used in a refinement relation such as in Eqn. 7.3.

7.2.1.4 Tau Priority Model (τ)

This model is similar to the trace model described above, except that it adds the notion of priority of events. A set of events is defined to have lower priority than internal events (τ) and hence the process could not offer any event from this set, unless it is in a stable state and cannot execute any internal events.

This model is useful for modelling the passage of time where all communication events in the system must occur before the next clock event.

Consider this scenario as a motivation for the Tau Priority Model: because no process in a parallel composition should be allowed to run faster than another, all processes must synchronise on timing events (i.e. *tock*). Consequently, processes are allowed to be idle and hence execute the *tock* event infinitely. However, for many semantic reasons, including the avoidance of false divergences (i.e. ones where only the *tock* event is involved), some events must have priority over *tock*.

A minimal approach for supporting priority of events is the Tau Priority Model, which is currently implemented in FDR [18] Version 2.91, which is the latest release as of August 2012.

When using the Tau Priority Model, only internal events can have higher priority over a selected set of external events (*tock* events in the case of tock-CSP). Let us consider the following processes as an example:

$$P1' = a \rightarrow b \rightarrow tock \rightarrow P1' \tag{7.5}$$

$$P2' = a \rightarrow b \rightarrow tock \rightarrow P2' \,\Box\, tock \rightarrow P2' \tag{7.6}$$

$$P1 = P1' \setminus \{a\} \tag{7.7}$$

$$P2 = P2' \setminus \{a\} \tag{7.8}$$

One way to interpret the difference between the two processes *P1'* and *P2'* is that process *P2'* takes into account an idle state when event a is not available; in which case, it can allow *tock* events to happen. Process *P1'*, however, must always be able to execute event a immediately at the start, which makes event a an *urgent* event. The details of this model can be found in [19, 38]. In particular, Ouaknine [82] highlights the semantic difficulties that are encountered when syntactically translating *Timed CSP* into tock-CSP. Most of those difficulties are adequately dealt with in the Tau Priority Model.

Under the Tau Priority Model, the two processes above are understood to be equivalent because, when run independently, both processes are able to execute the event a at the start. Consequently, the choice to execute the *tock* event in process *P2'* is ignored. However, the situation becomes more interesting when those processes are composed in parallel in a wider system in which event a is not available at the start. In this case, process *P1'* would introduce what is called a *time-stop*: any point in the execution of a process when *tock* events cannot happen. The process *P2'* does not suffer such a limitation.

The event a was hidden from the top-level processes *P1* and *P2* in the above example since the Tau Priority Model is only able to prioritize internal events over *tock* events.

The ability for some events to have less priority than others is useful in the context of communication systems and signal propagation: all signals in the system need to propagate and settle before the next clock event can happen. Processes which do not have any signal change in that clock cycle should be able to execute clock events gracefully.

The tock-CSP approach is a promising one. It integrates the modelling and verification of real-time performance aspects of embedded systems within the well understood CSP functional model-checking techniques. This thesis presents an important

contribution through the application of such an approach to a complex performance-critical communication system.

Finally, the Tau Priority Model is somehow restrictive, since all events that should have higher priority than the clock event must be modelled as internal. The notion of urgent external events could be achieved under the Tau Priority Model by letting an internal event (τ) proceed any external event that needs to have higher priority than other external events (*tock* in this case). It would be interesting to see the Tau Priority Model evolve and introduce the notion of priority to externally visible events explicitly.

7.2.2 Verification of Timing Specifications

In this and subsequent sections, each CSP verification check is called an *assertion*. Each assertion is represented by a mathematical equation and is referenced using the word *Assertion*, similar to the use of the abbreviation *Eqn.* to reference mathematical equations.

The notation used for describing the performed *Assertions* uses both the CSP_M interface of FDR [36], as well as the LaTeX symbol macros for CSP, described in [40] to achieve best readability.

Roscoe [38] discusses how timing specifications and constraints could be modelled and verified using tock-CSP processes. In particular, Roscoe [83] discusses many possible checks that could be performed on such processes. A brief discussion of those checks follows.

Let us assume that *Proc* is a real-time process modelled using tock-CSP. Let us also assume that all CSP events apart from *special timing events* (*tock* in the case of this discussion) are classed as *normal* events ($normal = \Sigma \backslash \{tock\}$). The following assertions could be performed on such a process:

$$Proc \setminus (\Sigma \backslash \{tock\}) \ \ divergence\text{-}free \tag{7.9}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} Proc \setminus (\Sigma \backslash \{tock\}) \tag{7.10}$$

Assertion 7.9 checks that there cannot be an infinite number of *normal* events occurring between two *special timing events* in the system represented by the process *Proc*. The ability of a physical system to execute an infinite number of events between two finite clock events (i.e. in a finite time frame) is not possible and would defy the laws of physics.

Time-stops have been briefly discussed in Section 3.3.3. Assertion 7.10 checks the absence of time-stops in the system and that time is always able to pass consistently. In addition, Assertion 7.10 also checks that the system is *divergence-free*. The TOCKS process used in Assertions 7.10 to 7.12 has been presented earlier in Eqn. 4.10.

$$\text{TOCKS} \sqsubseteq_{\text{FD}} (Proc \underset{Delayed}{\parallel} Chaos(Delayed)) \setminus (\Sigma \setminus \{tock\}) \tag{7.11}$$

$$\text{TOCKS} \mid\mid\mid Chaos(Delayed) \sqsubseteq_{\text{FD}} Proc \setminus (\Sigma \setminus (Delayed \cup \{tock\})) \tag{7.12}$$

The *Delayed* set in Assertion 7.11 represents events that can potentially be delayed indefinitely, but could also be allowed to execute at any time. Assertion 7.11 uses the special CSP process *Chaos* to model the notion of delayable events. See [35, Chapter 3] for more information about *Chaos*. The assertion also uses the *failures-divergences* refinement model to ensure timing consistency of *Proc*, as well as the absence of time-stops. The *Chaos* process could choose not to block the execution of an event in the *Delayed* set. This should not affect the timing consistency of *Proc*: it should be able to execute *special timing events* consistently. In any case, Assertion 7.11 still checks that the execution of delayable events adheres to timing consistency, whereby an infinite number of delayable events cannot be executed in a finite amount of time. If Assertion 7.11 was modified to use the *failures* model, then it would only check for the absence of time-stops.

Finally, the conditions of Assertion 7.11 could be relaxed to split the *normal* events into two sets: delayable events (*Delayed*) which do not adhere to the timing consistency checks of Assertions 7.9 to 7.11 and urgent events ($\Sigma \setminus (Delayed \cup \{tock\})$) which do. At this level, *Delayed* events could potentially be delayed forever, but also an infinite number of *Delayed* events could potentially be executed in a finite amount of time. For this reason, *Delayed* events should adhere to higher level timing consistency checks and are exempt from the timing consistency checks at this level (i.e. in Assertion 7.12). Assertion 7.12 verifies that the passage of time should not rely on the execution of any *Delayed* event or the lack of such an execution, hence Assertion 7.12 checks for the timing consistency of urgent events, while excluding *Delayed* events from such a check.

Assertions 7.10 to 7.12 represent checks that are similar to the *bottom* law for refinement ($RUN \sqsubseteq_{\text{T}} \text{P}$), which is described by Schneider [39], except that they address timing properties rather than the progression of asynchronous events.

7.3 Basic Functional and Timing Verification

First, basic properties about key functional units, as well as the abstract specifications of protocols were checked. These include basic liveness and safety properties. The deadlock freedom liveness property was checked using the Stable Failures Model ($\mathcal{F}$) for all the functional units, as well as a selected construction of the top-level system.

$$\text{ISA} \quad \textit{deadlock-free}[\![\mathcal{F}]\!] \tag{7.13}$$

$$\text{REG} \quad \textit{deadlock-free}[\![\mathcal{F}]\!] \tag{7.14}$$

$$\text{STC} \quad \textit{deadlock-free}[\![\mathcal{F}]\!] \tag{7.15}$$

$$\text{SBC} \quad \textit{deadlock-free}[\![\mathcal{F}]\!] \tag{7.16}$$

$$\text{DDC} \quad \textit{deadlock-free}[\![\mathcal{F}]\!] \tag{7.17}$$

$$\text{SYSTEM}_{\textit{one-bit}} \quad \textit{deadlock-free}[\![\mathcal{F}]\!] \tag{7.18}$$

Assertions 7.13 to 7.18 check for strong deadlock freedom of the associated processes, which means the processes are not even allowed to successfully terminate.

With respect to basic safety checks, the timing consistency check described in Assertion 7.10 serves as a minimum timing check for essential timed processes. Let all events except *tock* be treated as urgent ($\Sigma \backslash \{tock\}$), then Assertions 7.19 to 7.23 verify the basic timing correctness of the functional units, as well as a one bit construction of the communication system.

$$\text{TOCKS} \sqsubseteq_{\text{FD}} (\text{ISA} \underset{aREG}{\|} \text{REG}) \setminus \Sigma \backslash \{tock\} \tag{7.19}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \text{STC} \setminus \Sigma \backslash \{tock\} \tag{7.20}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \text{DDC} \setminus \Sigma \backslash \{tock\} \tag{7.21}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \text{SBC} \setminus \Sigma \backslash \{tock\} \tag{7.22}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \text{SYSTEM}_{\textit{one-bit}} \setminus \Sigma \backslash \{tock\} \tag{7.23}$$

7.4 ISA-Oriented Specification Verification

Basic verification of the ISA-*Oriented Specification*s involved performing the following timing and deadlock assertions:

$$\textsc{Uart}_{\text{ISA-RX}} \quad \mathit{deadlock\text{-}free}[\![\mathcal{F}]\!] \tag{7.24}$$
$$\textsc{Uart}_{\text{ISA-TX}} \quad \mathit{deadlock\text{-}free}[\![\mathcal{F}]\!] \tag{7.25}$$
$$\textsc{Uart}_{\text{ISA-BUF-RX}} \quad \mathit{deadlock\text{-}free}[\![\mathcal{F}]\!] \tag{7.26}$$
$$\textsc{Uart}_{\text{ISA-BUF-TX}} \quad \mathit{deadlock\text{-}free}[\![\mathcal{F}]\!] \tag{7.27}$$
$$\textsc{Spi}_{\text{ISA-MASTER}} \quad \mathit{deadlock\text{-}free}[\![\mathcal{F}]\!] \tag{7.28}$$
$$\textsc{Spi}_{\text{ISA-SLAVE}} \quad \mathit{deadlock\text{-}free}[\![\mathcal{F}]\!] \tag{7.29}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\text{ISA-RX}} \setminus \Sigma\backslash\{\mathit{tock}\} \tag{7.30}$$
$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\text{ISA-TX}} \setminus \Sigma\backslash\{\mathit{tock}\} \tag{7.31}$$
$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\text{ISA-BUF-RX}} \setminus \Sigma\backslash\{\mathit{tock}\} \tag{7.32}$$
$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\text{ISA-BUF-TX}} \setminus \Sigma\backslash\{\mathit{tock}\} \tag{7.33}$$
$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Spi}_{\text{ISA-MASTER}} \setminus \Sigma\backslash\{\mathit{tock}\} \tag{7.34}$$
$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Spi}_{\text{ISA-SLAVE}} \setminus \Sigma\backslash\{\mathit{tock}\} \tag{7.35}$$

All the above assertions are valid, except the ones relating to a Uart receiver (Assertions 7.24, 7.26, 7.30, and 7.32) because a stand-alone receiver specification would halt if not connected with a transmitter, due to the possible error in the reception of the stop-bit. See the relevant ISA *Specification* in Section 6.4.1 for more details.

In addition, ISA-*Oriented Specification* verification involved checking the willingness of the unconfigured communication system to execute such a specification as follows:

$$\text{SYSTEM}_{\mathit{one\text{-}bit}} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\text{ISA-RX}} \tag{7.36}$$
$$\text{SYSTEM}_{\mathit{one\text{-}bit}} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\text{ISA-TX}} \tag{7.37}$$
$$\text{SYSTEM}_{\mathit{one\text{-}bit}} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\text{ISA-BUF-RX}} \tag{7.38}$$
$$\text{SYSTEM}_{\mathit{one\text{-}bit}} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\text{ISA-BUF-TX}} \tag{7.39}$$
$$\text{SYSTEM}_{\mathit{three\text{-}bit}} \sqsubseteq_{\text{FD}} \textsc{Spi}_{\text{ISA-MASTER}} \tag{7.40}$$
$$\text{SYSTEM}_{\mathit{three\text{-}bit}} \sqsubseteq_{\text{FD}} \textsc{Spi}_{\text{ISA-SLAVE}} \tag{7.41}$$

where:

$$\text{SYSTEM}_{three\text{-}bit} = (((\text{ISA} \underset{aReg}{\parallel} \text{REG}) \underset{aSbc}{\parallel} \text{SBC0}) \underset{aStc}{\parallel} \text{STC0}) \underset{aDdc}{\parallel} \text{DDC}_{3\text{-}bit} \quad (7.42)$$

Assertions 7.36 to 7.41 check the consistency of the ISA-*Oriented Specification* of a protocol with the relevant unconfigured communication system construction. These assertions still do not give any assurance of the functional correctness of the constructed system, nor of the correctness of the associated ISA-*Oriented Specification*. However, they merely check that all the ISA-*Oriented Specifications* of the protocols are consistent with the *Instruction Set* defined in Section 5.3.5.2.

If modelling rules were enforced so that all ISA-*Oriented Specifications* were written strictly using the *Instruction Set* defined in Section 5.3.5.2 then Assertions 7.36 to 7.41 would be satisfied using compositional semantics rules. This is true since the stable failures of the ISA process in Eqn. 5.136 are the union of the stable failures of all individual instruction sets. Assuming that an instruction (I) is an atomic unit, whose sequence of events is represented by the atomic event i ($I = i \rightarrow \text{SKIP}$), then the stable failures of an individual instruction (I) include the set of all atomic instructions except I. From the compositional semantics of the internal choice operator used to model the top-level ISA process, one could infer that the stable failures of the top-level ISA process is the union of the refusal sets of all individual instructions: i.e. ISA cannot refuse to execute any instruction in the instruction set. Hence, if an ISA-*Oriented Specification* of a protocol was strictly a sequence of the individual instruction sets, then the refusal set of an ISA-*Oriented Specification* at any point in time includes the set of all instructions, except the next instruction in the sequence. This refusal set is a subset of the refusal set of the top-level ISA process mentioned earlier. Hence, using theses rules, Assertions 7.36 to 7.41 would be satisfied using these compositional semantics rules of the stable failures model and typically there would not be a need for FDR to verify them.

Practically, Assertions 7.36 to 7.41 are used to enforce the modelling rule that all ISA-Oriented Specifications are indeed written strictly using the *Instruction Set* defined in Section 5.3.5.2.

7.5 Abstract Specification Verification

Deadlock checking and timing consistency checks were also performed on the abstract specifications of the communication protocols:

$$\textsc{Uart}_{\textsc{spec-rx}} \quad deadlock\text{-}free[\![\mathcal{F}]\!] \tag{7.43}$$

$$\textsc{Uart}_{\textsc{spec-tx}} \quad deadlock\text{-}free[\![\mathcal{F}]\!] \tag{7.44}$$

$$\textsc{Spi}_{\textsc{spec-master}} \quad deadlock\text{-}free[\![\mathcal{F}]\!] \tag{7.45}$$

$$\textsc{Spi}_{\textsc{spec-slave}} \quad deadlock\text{-}free[\![\mathcal{F}]\!] \tag{7.46}$$

The specification of $\textsc{Uart}_{\textsc{spec-rx}}$ enforces the input stop-bit value to be *high*, which is different from the respective ISA-*Oriented Specification*. For this reason Assertion 7.43 is valid, unlike Assertion 7.24. This is an interesting difference in specification styles because both are valid specifications and should be interoperable, as will be verified in Section 7.9.

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\textsc{spec-rx}} \setminus \Sigma\backslash\{tock\} \tag{7.47}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Uart}_{\textsc{spec-tx}} \setminus \Sigma\backslash\{tock\} \tag{7.48}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Spi}_{\textsc{spec-master}} \setminus \Sigma\backslash\{tock\} \tag{7.49}$$

$$\text{TOCKS} \sqsubseteq_{\text{FD}} \textsc{Spi}_{\textsc{spec-slave}} \setminus \Sigma\backslash\{tock\} \tag{7.50}$$

Assertions 7.47 to 7.50 verify the basic timing consistency check on all abstract specifications. All assertions are valid, except Assertion 7.50 because the process $\textsc{Spi}_{\textsc{spec-slave}}$ is an asynchronous process and does not execute the *tock* timing event.

7.6 ISA System-level Functional Verification

Roscoe [38, Section 4.3] describes functional verification checks of communication protocols. Roscoe proves the functional correctness of a communication protocol by verifying the equivalence of a complex communication channel constructed by connecting a sender and a receiver of such a protocol to a multi-entry buffer.

The refinement models described earlier verify the refinement relation between processes and not their equivalence. An equivalence relation is established using two-way refinement. This approach to equivalence model-checking has potential complexity implications when performed in FDR: each process would have to be compiled, explicated and traversed twice. Despite this complexity complication, for the purpose of establishing system-level correctness of the specifications, two-way refinements were used to establish equivalence relations throughout this chapter. Through the use of TCL scripting, it is possible for equivalence checks to be optimised by constructing

two *hypotheses* using the same compiled processes. For demonstration purposes, a two-way refinement symbol is used in this section:

$$\mathrm{P} \sqsubseteq \mathrm{Q} \wedge \mathrm{Q} \sqsubseteq \mathrm{P} \qquad \Leftrightarrow \qquad \mathrm{P} \equiv \mathrm{Q} \tag{7.51}$$

This implied equivalence was used to establish the functional correctness of all protocol specifications. In this section and the next one, channel construction uses a transmitter and a receiver of the same protocol at the same abstraction level. This is performed for both the ISA-*Oriented Specification*s discussed in Section 6.4, as well as the abstract specifications of the protocols discussed in Section 6.5. By doing so, the functional correctness of the specifications for all transmitters and receivers is established and data integrity is maintained through all communicating processes.

As discussed in Section 6.5, to establish a global notion of time between different processes interacting in a real-time transaction, a real-time event must appear in their synchronisation alphabet. Hence, in the construction of a complex Uart channel, the *tock* event is used to synchronise the transmitter and receiver. On the other hand, and because the Spi protocol is a synchronous protocol that explicitly exchanges a clock signal, the Spi clock signal is used to establish a global notion of time instead of explicit synchronisation using *tock*. This is justified by the fact that the $\text{Spi}_{\text{SPEC-SLAVE}}$ has been modelled as a completely asynchronous process, which does not execute *tock* events. All ISA-*Oriented Specifications* have to synchronise on *tock* events, because all ISA-*Oriented* system constructions are synchronous and execute the *tock* event.

First, an abstract simple buffer and its alphabets are defined as follows:

$$aBuffer = \{|produce, consume|\} \tag{7.52}$$

$$Buffer = produce?data \rightarrow consume!data \rightarrow Buffer \tag{7.53}$$

Then two Uart communication channels are constructed, one for the unbuffered specification and one for the buffered one:

$$\text{Uart}_{\text{ISA-TX}}\,' = \text{Uart}_{\text{ISA-TX}}\,[\![dRO0/pUART]\!] \tag{7.54}$$

$$\text{Uart}_{\text{ISA-RX}}\,' = \text{Uart}_{\text{ISA-RX}}\,[\![dRI0/pUART]\!] \tag{7.55}$$

$$\text{Uart}_{\text{ISA-CHAN}}\,' = \text{Uart}_{\text{ISA-TX}}\,' \underset{\{|pUART,tock|\}}{\|} \text{Uart}_{\text{ISA-RX}}\,' \tag{7.56}$$

$$\text{Uart}_{\text{ISA-CHAN}} = \text{Uart}_{\text{ISA-CHAN}}\,' \setminus \Sigma\backslash\, aBuffer \tag{7.57}$$

$$\textsc{Uart}_{\textsc{isa-buf-tx}}\,' = \textsc{Uart}_{\textsc{isa-buf-tx}}\,[\![^{dRO0}/_{pUART}]\!] \tag{7.58}$$

$$\textsc{Uart}_{\textsc{isa-buf-rx}}\,' = \textsc{Uart}_{\textsc{isa-buf-rx}}\,[\![^{dRI0}/_{pUART}]\!] \tag{7.59}$$

$$\textsc{Uart}_{\textsc{isa-buf-chan}}\,' = \textsc{Uart}_{\textsc{isa-buf-tx}}\,' \underset{\{|pUART,tock|\}}{\parallel} \textsc{Uart}_{\textsc{isa-buf-rx}}\,' \tag{7.60}$$

$$\textsc{Uart}_{\textsc{isa-buf-chan}} = \textsc{Uart}_{\textsc{isa-buf-chan}}\,' \setminus \Sigma\backslash\, aBuffer \tag{7.61}$$

Similarly, the ISA-based complex communication channel for the Spi protocol is constructed as follows:

$$aSPIchan = \{|pClk, pMiso, pMosi, tock|\} \tag{7.62}$$

$$\textsc{Spi}_{\textsc{isa-master}}\,' = \textsc{Spi}_{\textsc{isa-master}}\,[\![^{dRO0, dRI1, dRO2}/_{pClk, pMiso, pMosi}]\!] \tag{7.63}$$

$$\textsc{Spi}_{\textsc{isa-slave}}\,' = \textsc{Spi}_{\textsc{isa-slave}}\,[\![^{dRI0, dRO1, dRI2}/_{pClk, pMiso, pMosi}]\!] \tag{7.64}$$

$$\textsc{Spi}_{\textsc{isa-chan}}\,' = \textsc{Spi}_{\textsc{isa-master}}\,' \underset{aSPIchan}{\parallel} \textsc{Spi}_{\textsc{isa-slave}}\,' \tag{7.65}$$

$$\textsc{Spi}_{\textsc{isa-chan}} = \textsc{Spi}_{\textsc{isa-chan}}\,' \setminus \Sigma\backslash\, aBuffer \tag{7.66}$$

Then, using the failures/divergences refinement model, the complex channels are proven to be functionally equivalent to a single-entry buffer:

$$\textsc{Uart}_{\textsc{isa-chan}} \equiv_{\textsc{fd}} Buffer \tag{7.67}$$

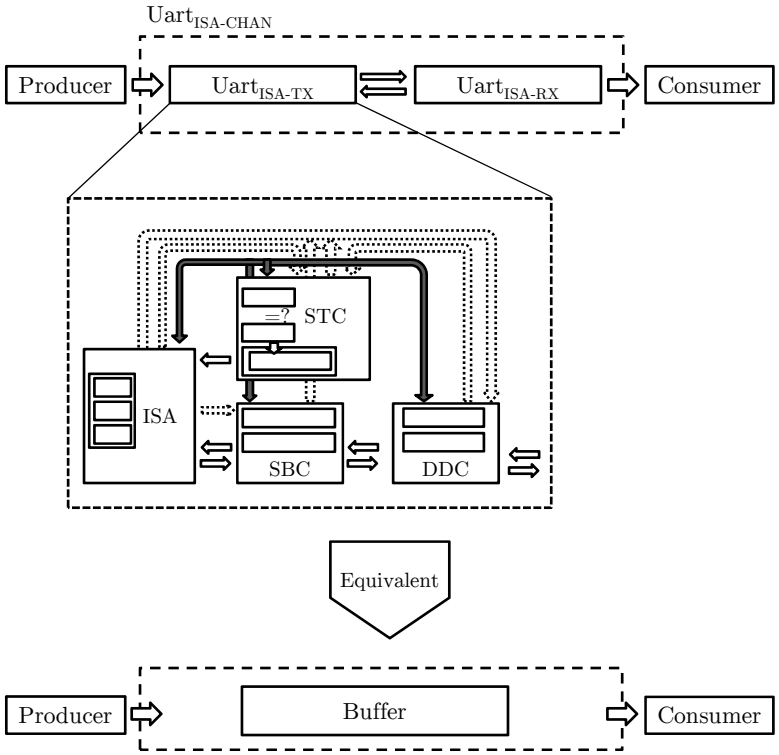
$$\textsc{Uart}_{\textsc{isa-buf-chan}} \equiv_{\textsc{fd}} Buffer \tag{7.68}$$

$$\textsc{Spi}_{\textsc{isa-chan}} \equiv_{\textsc{fd}} Buffer \tag{7.69}$$

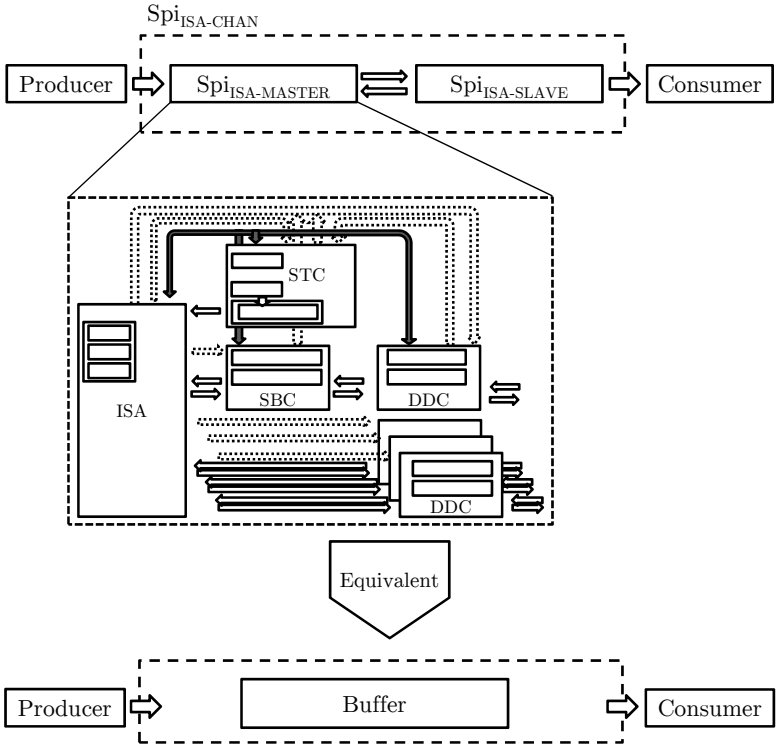
Figures 7.1a and 7.1b show block diagrams of the ISA-*Oriented* channels constructed by Eqn. 7.66 and Eqn. 7.57 respectively, as well as their equivalence to a buffer verified by Assertions 7.69 and 7.67.

7.7 Abstract System-level Functional Verification

A simple abstraction of the physical layer is specified to act as a signal register or physical wire. This is an asynchronous process: it does not execute *tock* events or synchronise on them. Its main function is to register the physical value of the interface set by a transmitter and make this value available to a receiver at any time. This



(a) Uart Protocol



(b) Spi Protocol

Figure 7.1 ISA Functional Verification

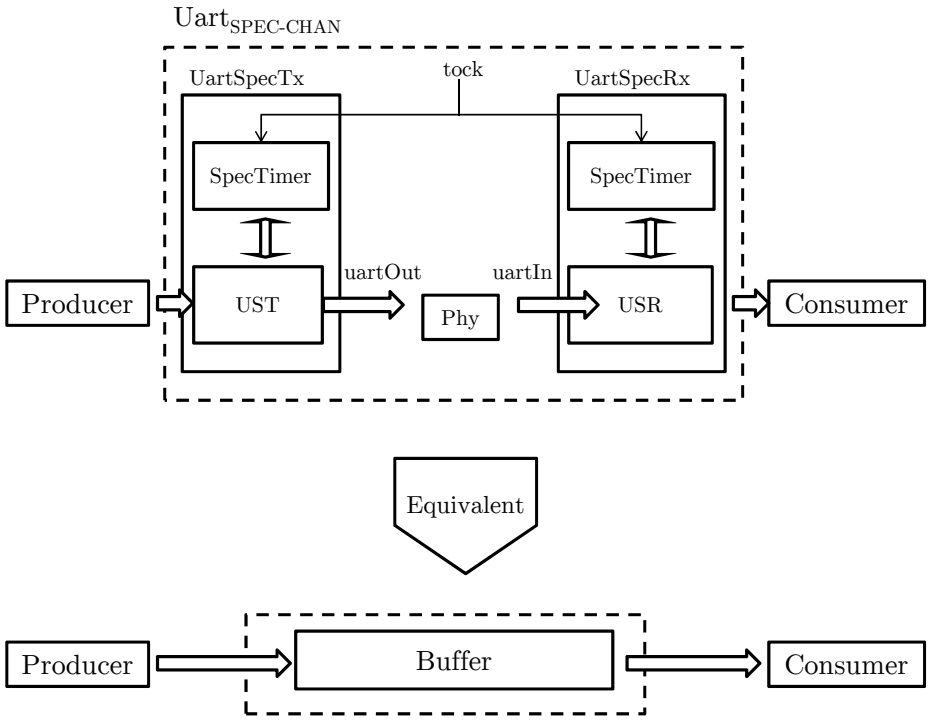


Figure 7.2 UART Specification Functional Verification

is equivalent to an abstract asynchronous 1 bit register. Its function is implicitly provided by the DDC functional unit described in Chapter 5.

$$Phy = Phy\prime(1) \tag{7.70}$$

$$Phy\prime(d) = in?x \rightarrow Phy\prime(x) \tag{7.71}$$

$$\Box\ out!d \rightarrow Phy\prime(d) \tag{7.72}$$

The default value of any physical connection at system initialisation is assumed to be 1 ($d = 1$ in Eqn. 7.71).

Complex communication channels were constructed using the abstract specification of the protocols. For example, a UART specification channel composed of the $\text{UART}_{\text{SPEC-RX}}$ and $\text{UART}_{\text{SPEC-TX}}$ called $\text{UART}_{\text{SPEC-CHAN}}$ is demonstrated in Figure 7.2. Similarly, an SPI specification channel composed of the $\text{SPI}_{\text{SPEC-MASTER}}$ and $\text{SPI}_{\text{SPEC-SLAVE}}$ called $\text{SPI}_{\text{SPEC-CHAN}}$ is demonstrated in Figure 7.3. Those complex specification channels are constructed in Eqns. 7.74 and 7.77, respectively.

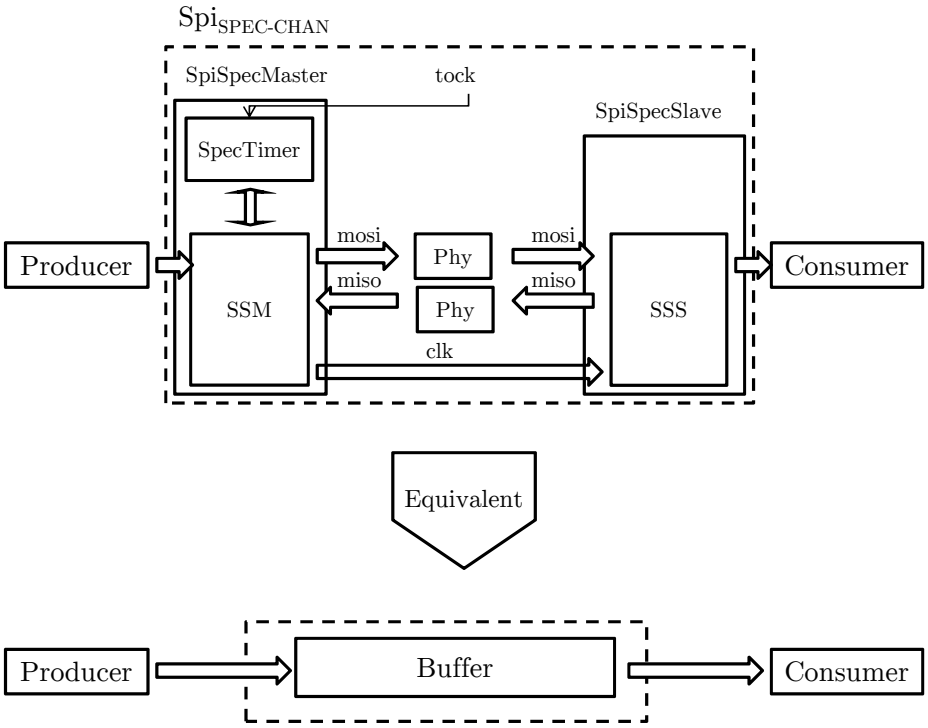


Figure 7.3 SPI Specification Functional Verification

$$\text{UART}_{\text{SPEC-RX}}\,' = \text{UART}_{\text{SPEC-RX}} \underset{\{|\,\text{UART}_{In}\,|\}}{\|} Phy[\![^{in,\,out}/\text{UART}_{In}, \text{UART}_{Mid}]\!] \tag{7.73}$$

$$\text{UART}_{\text{SPEC-CHAN}} = \tag{7.74}$$

$$\text{UART}_{\text{SPEC-RX}}\,' \underset{\{|\,\text{UART}_{Mid},tock|\}}{\|} \text{UART}_{\text{SPEC-TX}}\,[\![^{\text{UART}_{Out}}/\text{UART}_{Mid}]\!] \tag{7.75}$$

$$\text{SPI}_{\text{SPEC-MASTER}}\,' = \text{SPI}_{\text{SPEC-MASTER}} \underset{\{|sSMosi|\}}{\|} Phy[\![^{in,\,out}/sSMosi, midMosi]\!] \tag{7.76}$$

$$\text{SPI}_{\text{SPEC-CHAN}} = \tag{7.77}$$

$$\text{SPI}_{\text{SPEC-MASTER}}\,' \underset{\{|sSClk,midMosi|\}}{\|} \text{SPI}_{\text{SPEC-SLAVE}}\,[\![^{sSMosi}/midMosi]\!] \tag{7.78}$$

Finally, establishing the equivalence of the complex channels in Eqn. 7.74 and Eqn. 7.77 to a buffer verifies the functional correctness of the associated abstract specifications. This was not a simple task as the following discussion demonstrates.

Functional correctness of a specification channel was established using two separate and distinct assertions, unlike the functional verification checks of the ISA-*Oriented Specifications*, which used two similar refinement checks of the form:

$$Protocol_{\text{CHAN}} \sqsubseteq Buffer \wedge Buffer \sqsubseteq Protocol_{\text{CHAN}} \tag{7.79}$$

First, the following two assertions using the unclocked buffer were verified to be true:

$$\text{UART}_{\text{SPEC-CHAN}} \setminus \Sigma\backslash\, aBuffer \sqsubseteq_{\text{FD}} Buffer \tag{7.80}$$

$$\text{SPI}_{\text{SPEC-CHAN}} \setminus \Sigma\backslash\, aBuffer \sqsubseteq_{\text{FD}} Buffer \tag{7.81}$$

Assertions 7.81 and 7.80 verify that the complex specification channels can at least act as a single-entry buffer. However, because the Tau Priority Model is not used, there could be instances where the complex channels miss their real-time deadlines, so it is quite likely they could corrupt the data being buffered, as demonstrated by Assertion 7.82.

$$\text{SPI}_{\text{SPEC-CHAN}} \setminus \Sigma\backslash\, aBuffer \sqsubseteq_{\text{FD}} produce?0 \rightarrow consume!1 \rightarrow \text{SKIP} \tag{7.82}$$

Assertion 7.82 is true and hence verifying the specification channels using the *failures/divergences* refinement model is not sufficient.

In all abstract specifications, tock-CSP was used in a simplistic manner, as discussed in Section 6.5, where only the *SpecTimer* process was responsible for managing the real-time. For this reason, real-time communication events between *SpecTimer* and other specification processes (i.e. events *trigger*, *set*, *reset*) would need to have higher priority over *tock* events, if they are to be executed at the right time. For this reason, the Tau Priority Model was the model of choice for functional verification of the abstract specifications. However, under the Tau Priority Model only internal events can have higher priority over external events. If the *tock* event could not be hidden in a refinement check to maintain its lower priority, and in the lack of any performance specifications, then the abstract *Buffer* process could be allowed to execute an arbitrary number of *tock* events between an input and output:

$$Buffer\prime = Buffer \,|||\, \text{TOCKS} \tag{7.83}$$

However, if the complex specification channels hide all events except *tock*, *produce* and *consume*, then they become clocked complex channels. These constructions could be verified using the Tau Priority Model as follows:

$$\textsc{Uart}_{\textsc{spec-chan}}\,\prime = \textsc{Uart}_{\textsc{spec-chan}} \setminus \Sigma\backslash(\{tock\} \cup aBuffer) \tag{7.84}$$

$$\textsc{Spi}_{\textsc{spec-chan}}\,\prime = \textsc{Spi}_{\textsc{spec-chan}} \setminus \Sigma\backslash(\{tock\} \cup aBuffer) \tag{7.85}$$

$$Buffer\,\prime \sqsubseteq_{\mathrm{T}} \textsc{Uart}_{\textsc{spec-chan}}\,\prime[\![tau\text{-}priority\text{-}over]\!]\{tock\} \tag{7.86}$$

$$Buffer\,\prime \sqsubseteq_{\mathrm{T}} \textsc{Spi}_{\textsc{spec-chan}}\,\prime[\![tau\text{-}priority\text{-}over]\!]\{tock\} \tag{7.87}$$

Assertions 7.87 and 7.86 verify that the clocked complex channels in Eqns. 7.85 and 7.84 preserve the integrity of all the communicated data items, because they both refine a clocked single-entry buffer.

Assertions 7.87 and 7.86 lack any performance verification, which means they allow both processes on either side of the refinement check to execute an arbitrary number of clock cycles for each transmitted data item. Performance verification is discussed in Section 7.8.

Finally, Assertions 7.87 and 7.86 have striking similarities with the timing consistency check in Assertion 7.12, which takes into account delayable events (*produce* and *consume* in this case) that are never required to enable *tock*. This is quite the case for the *produce* event, but not precisely so for the *consume* event. In the clocked complex channels of Eqns. 7.85 and 7.84 *consume* has some sense of urgency. It is allowed to be delayed within a specific time interval. This interval is equivalent to 1 bit-time for process $\textsc{Uart}_{\textsc{spec-rx}}$ in Eqn. 6.108. At the end of that interval, the *consume* event becomes urgent, because the receiver must be able to start the reception of the next data word.

7.8 System-level Performance Verification

Performance specifications are an important aspect of real-time communication protocols and systems. Those specifications were modelled earlier in Chapter 5 for the whole communication system. Then, Section 6.4 used the ISA-*Oriented Specification* technique to specify both the functional and performance specifications of selected protocols. Also, Section 6.5 provided independent functional and performance specifications for the selected protocols. All real-time specifications were modelled using tock-CSP.

The high-level performance specifications of communication protocols could include the latency of communications and the maximum possible throughput an implementation can achieve. A more elaborate performance specification for an ISA-based communication system could also include lower-level performance specifications, such as instruction delays and the latencies associated with different I/O instructions.

This section addresses the verification of high-level performance aspects of the modelled protocols. Those performance aspects could be specified using standard tock-CSP or alternatively they could also be specified using the ISA-*Oriented Specification*, as demonstrated below.

Using the definition of the TOCK process in Section 5.3.1, a timed buffer could be defined:

$$\mathrm{SR} = sw0w?data \rightarrow sw0r!data \rightarrow \mathrm{SR} \tag{7.88}$$

$$aSR = \{|sw0w, sw0r|\} \tag{7.89}$$

$$Buffer_{Timed}\prime(n) = produce?data \rightarrow sw0w!data \rightarrow \mathrm{TOCK}(n); \tag{7.90}$$

$$sw0r?data \rightarrow consume!data \rightarrow Buffer_{Timed}\prime(n) \tag{7.91}$$

$$\Box\ tock \rightarrow Buffer_{Timed}\prime(n) \tag{7.92}$$

$$Buffer_{Timed}(n) = (Buffer_{Timed}\prime(n) \underset{aSR}{\parallel} \mathrm{SR}) \setminus aSR \tag{7.93}$$

$Buffer_{Timed}$ in Eqn. 7.93 buffers a data item received on the *produce* channel for an exact amount of clock cycles (n) before it makes it available for output on the *consume* channel. This is useful for measuring and verifying latency specifications for the communication protocols, using the complex communication channels that were constructed in Sections 7.6 and 7.7.

7.8.1 Latency Verification

First, the expected latency between input and output of a complex communication channel is defined in terms of the number of clocks per bit ($\textsc{Uart}_{ClkPerBit} = uCPB$ and $\textsc{Spi}_{ClkPerBit} = sCPB$) and the number of bits in a data word ($DataSize$):

$$\textsc{Uart}_{ClkWord} = (2 + DataSize) \times \textsc{Uart}_{ClkPerBit} + x \tag{7.94}$$

$$\textsc{Uart}_{BufClkWord} = (2 + DataSize) \times \textsc{Uart}_{ClkPerBit} + y \tag{7.95}$$

$$\textsc{Spi}_{ClkWord} = DataSize \times \textsc{Spi}_{ClkPerBit} - \textsc{Spi}_{ClkPerBit}/2 + z \tag{7.96}$$

Notice that one start-bit and one stop-bit were used for the UART specification and hence 2 is added to the *DataSize* in Eqns. 7.94 and 7.95. Also, the specification of the SPI protocol allows for the received data word to be transmitted on the *consume* channel once the rising edge of the external clock signal associated with the last data bit has been observed. For this reason, the value $\text{SPI}_{ClkPerBit}/2$ is subtracted from the expected latency of the data word. Finally, the constants x, y, z are used to adjust for small delays incurred due to latencies between the transmitter and receiver and also for allowing a delay overhead associated with few instructions in the ISA specifications.

Using the above definitions, the latencies of the complex communication channels could be verified as follows:

$$Buffer_{Timed}(\text{UART}_{ClkWord}) \sqsubseteq_{\text{T}} \text{UART}_{\text{ISA-CHAN}}\,\prime \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.97}$$

$$Buffer_{Timed}(\text{UART}_{BufClkWord}) \sqsubseteq_{\text{T}} \text{UART}_{\text{ISA-BUF-CHAN}}\,\prime \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.98}$$

$$Buffer_{Timed}(\text{SPI}_{ClkWord}) \sqsubseteq_{\text{T}} \text{SPI}_{\text{ISA-CHAN}}\,\prime \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.99}$$

In Assertions 7.97, 7.98 and 7.99, an abstraction of the complex channel is obtained where all events are hidden, apart from the buffer channels, as well as clocking events (*tock*). The assertions verify that all complex channels constructed using the ISA-*Oriented Specifications* of the protocols meet the specified latency requirements.

It was also found that the lowest *uCPB* for the unbuffered UART specification was 4. Using the SBC unit, a *uCPB* as low as 2 was fully functional. Hence, the SBC could potentially double the maximum achievable bit-rate by halving the lowest possible latency.

Verifying the performance of SPI, as specified by assertion 7.99 was tricky. Functional verification of SPI using $sCPB = 2$ and $TimeSize = 2$ was possible. However, through the examination of the ISA-*Oriented Specification*s in Section 6.4.2 it was clear that *Master* devices need to execute about 10 instructions each bus cycle. This information eliminates the need for performance analysis of SPI for $sCPB < 10$. Analysis of the performance for $sCPB \geq 10$ is best demonstrated using the *Tock-CSP Waveform Generator*, which is discussed in the following section.

7.8.1.1 Tock-CSP Waveform Generator

Waveform visualisation is an essential part of hardware design development cycles. It can be used for providing real-time specifications and also for analysing faults in the

design through a timeline view of changes to the selected signals. For this reason, the addition of a waveform generator to the model-checker for added visualisation of tock-CSP traces would help bridge the gap between formal specification and verification approaches and the hardware implementation and verification tools. Hence, a trace analyser and Verilog Value Change Dump (VCD) generator have been integrated into the FDRlei plug-in. The full documentation of FDRlei can be found in Appendix B. In this section, a brief discussion of the VCD generation process is presented along with a demonstration of its use.

At least two processes are provided to FDRlei. Optionally, the refinement model, witness number, analysed process name and VCD file name can also be provided. FDRlei then uses the TCL interface to an FDR server to construct and execute a refinement check, which is assumed to fail, in which case the failing traces could be identified and analysed. A top-level process trace information could be interpreted as a tree of failed traces: one for each subprocess involved in the construction of that top-level process. Hence, an optional third process could be used to identify the node in the trace tree for analysis. FDRlei then constructs the trace tree interactively with the FDR server and searches through this trace tree to identify the required trace.

Once a failing trace has been identified, the following assumptions are made in order to interpret the trace in real-time and convert it into the VCD file format.

- All events (even complex ones) are interpreted as *typeless* events, unless the value appearing after the last dot operator is a numeric value. For example, $input.10.Data$ is interpreted as a simple event, while $input.Data.10$ is interpreted as a complex event with value 10.

- All simple events are interpreted as "wires" in Verilog. Once a simple event occurs in a tock-CSP cycle, the wire value is changed to 1 for half of the respective cycle.

- A complex CSP event is represented in Verilog as a number of wires (or bus). The occurrence of a complex event at any cycle changes the bus value at the start of the cycle (i.e. just after the rising edge of the clock). It does not change until the complex event appears again in the CSP trace.

An ambiguity might occur between a typeless event and a complex event which is 1 bit wide. For this reason, a typeless event is only activated in the VCD trace for half the tock-CSP cycle, and goes back to zero in the second half allowing for its occurrence in the next cycle to be clearly identifiable in the waveform. A complex

event which is of width 1 changes its value at the start of the cycle (just like any other complex event) and does not change again unless it appears again in the CSP trace.

Once FDRlei has finished interpreting the CSP trace, a VCD file is generated. A waveform viewer, such as GTKWave [84] can then be used to view and analyse the generated waveform. FDRlei will attempt to instantiate GTKWave, providing it with the name of the generated VCD file.

Finkelstein et al. [84] present a user guide to GTKWave. Most importantly, Finkelstein et al. present a TCL interface to GTKWave which inspired automated interaction with GTKWave by FDRlei, similar to its interaction with the FDR server. This enabled an interactive mode in which FDRlei is able to change the trace view point in the debug tree at run time.

Figure 7.4 demonstrates the use of the waveform generator in analysing a failing trace of Assertion 7.30. The figure shows the sequence of instructions leading to the failing instruction (the TEST instruction identifying the stop-bit).

Only a selected subset of the *Verilog* standard [85] has been addressed by the waveform generator for use in this framework. Future work would include addressing more complex aspects of the standard. In particular, the notion of scopes could be used to enable the interpretation of the full debug tree of a top-level process, rather than only one subprocess at a time.

Not only can the waveform generator be useful for analysing failing traces, it could also be used to produce specification waveforms of the expected behaviour of a CSP specification. Those specification waveforms can guide the implementation process of such a specification into physical hardware.

7.8.1.2 Waveform Analysis of Latency Verification

As mentioned earlier, an Spi bus cycle takes about 10 instructions. Assuming each arithmetic instruction takes exactly 1 cycle and that all I/O instructions commit immediately and hence also take exactly 1 cycle, then theoretically the maximum possible performance is achieved when $sCPB = 10$, which requires a $TimeSize \geq 4$. Using these assumptions in the latency verification of the Spi complex channel using Assertion 7.99, it is evident that setting $sCPB = 10$ does not meet the specified latency requirement. Figure 7.5 shows the performance of the $\text{Spi}_{\text{ISA-MASTER}}$ analysed using the waveform generator.

Because $sCPB = 10$, the specification aims to generate a duty cycle of 50% for the bus clock (*mid-clk* in Figure 7.5). The figure shows that the falling edge of the

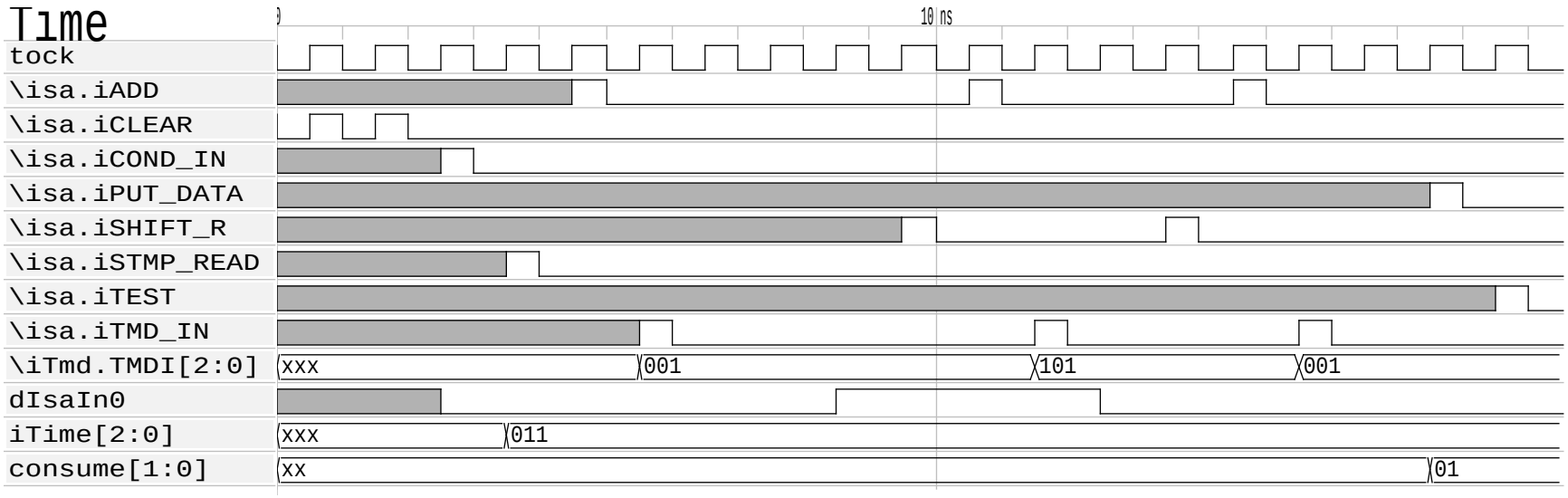


Figure 7.4 Waveform of a failing trace for Assertion 7.30

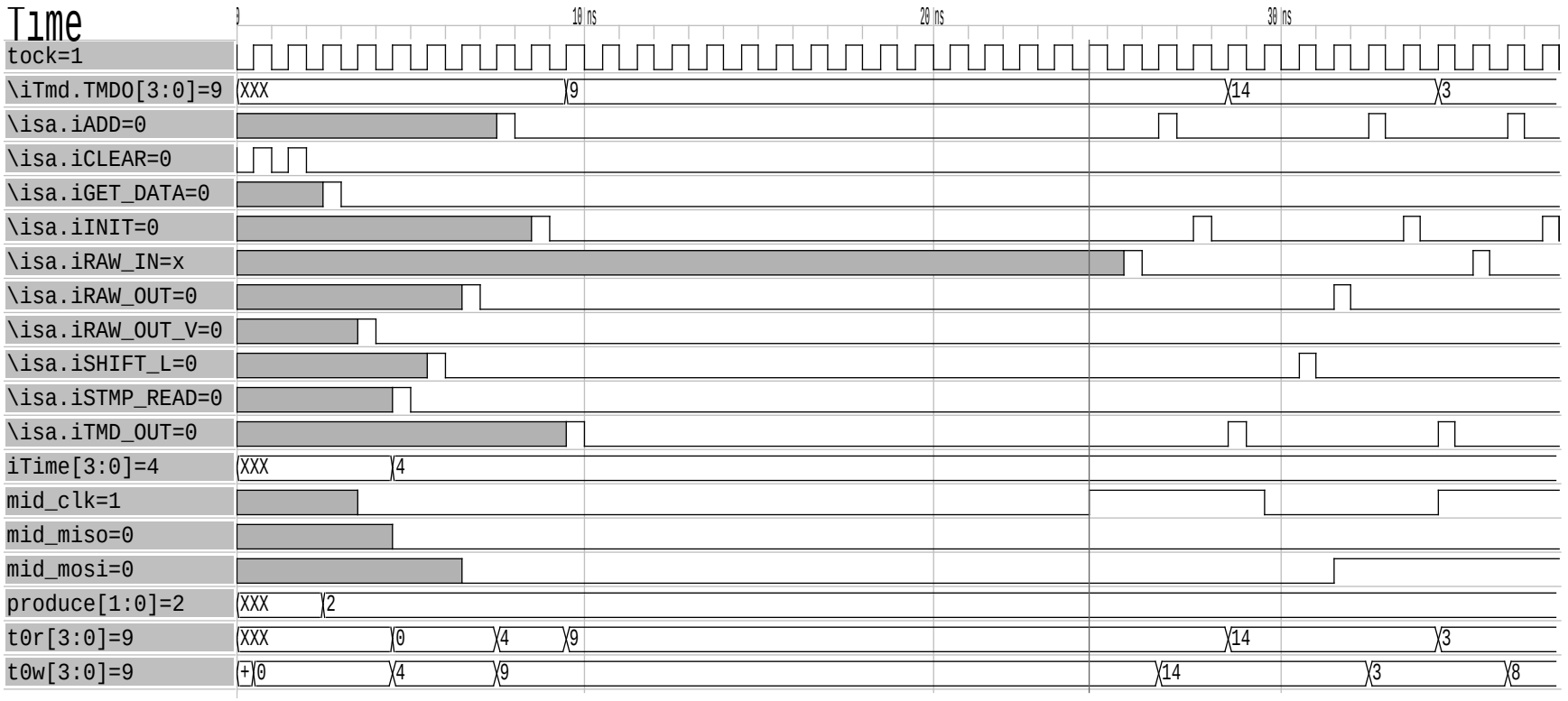


Figure 7.5 Waveform of Assertion 7.99 using $sCPB = 10$

first bit has a timestamp of 4 ($iTime = 4$). Then the rising edge was initiated using a timed output instruction (TMD_{OUT}) at the 9^{th} clock cycle (TMDO .9). However, this TMD_{OUT} instruction was posted at the 10^{th} clock cycle which meant it missed its deadline and had to wait for the time register in the STC unit to wrap. This takes 16 cycles and the falling edge of the bus clock takes place at the 25^{th} cycle stretching the bus clock by 16 system cycles.

Using $sCPB \geq 12$, the latency verification of Spi succeeds. Figure 7.6 shows the latency waveform for process $\text{Spi}_{\text{ISA-MASTER}}$ using $sCPB = 12$. It is evident from the waveform that the bus clock is now regular and the stretching that occurred when using $sCPB = 10$ no longer takes place, achieving the optimum bus clock of exactly 12 system clock cycles per 1 Spi bus cycle.

Finally, performance specifications could also be defined using ISA-*Oriented Specifications*. For example, the latencies expected could be defined in terms of the number of NOP instructions that could be executed between the start of a transmission of a data word until it is received and forwarded by the receiver:

$$\begin{aligned}
\text{Uart-ISA-latency} = {} & \text{GET}(produce, wReg0); && (7.100)\\
& \text{NOPS}((\text{Uart}_{ClkWord} - IDelay)/IDelay); && (7.101)\\
& \text{PUT}(consume, wReg0); && (7.102)\\
& \text{Uart-ISA-latency} && (7.103)\\
\Box\ & \text{NOP}\ ;\text{Uart-ISA-latency} && (7.104)
\end{aligned}$$

Hence, Assertion 7.97 demonstrated earlier could also be written as:

$$\begin{aligned}
&\text{Uart-ISA-latency} \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \sqsubseteq_{\text{T}} && (7.105)\\
&\qquad \text{Uart}_{\text{ISA-CHAN}}{}' \setminus \Sigma\backslash(aBuffer \cup \{tock\}) && (7.106)
\end{aligned}$$

Similarly, ISA-based latency specifications and assertions could be defined for $\text{Uart}_{\text{ISA-BUF-CHAN}}$ and $\text{Spi}_{\text{ISA-CHAN}}$.

7.8.2 Throughput Verification

Another measure of the performance of a communication channel is the maximum possible throughput of such a channel. This is useful when events are allowed to happen within a well-defined time window, but the exact cycle at which events occur is unknown. For example, the exact clock cycle for forwarding the data word to

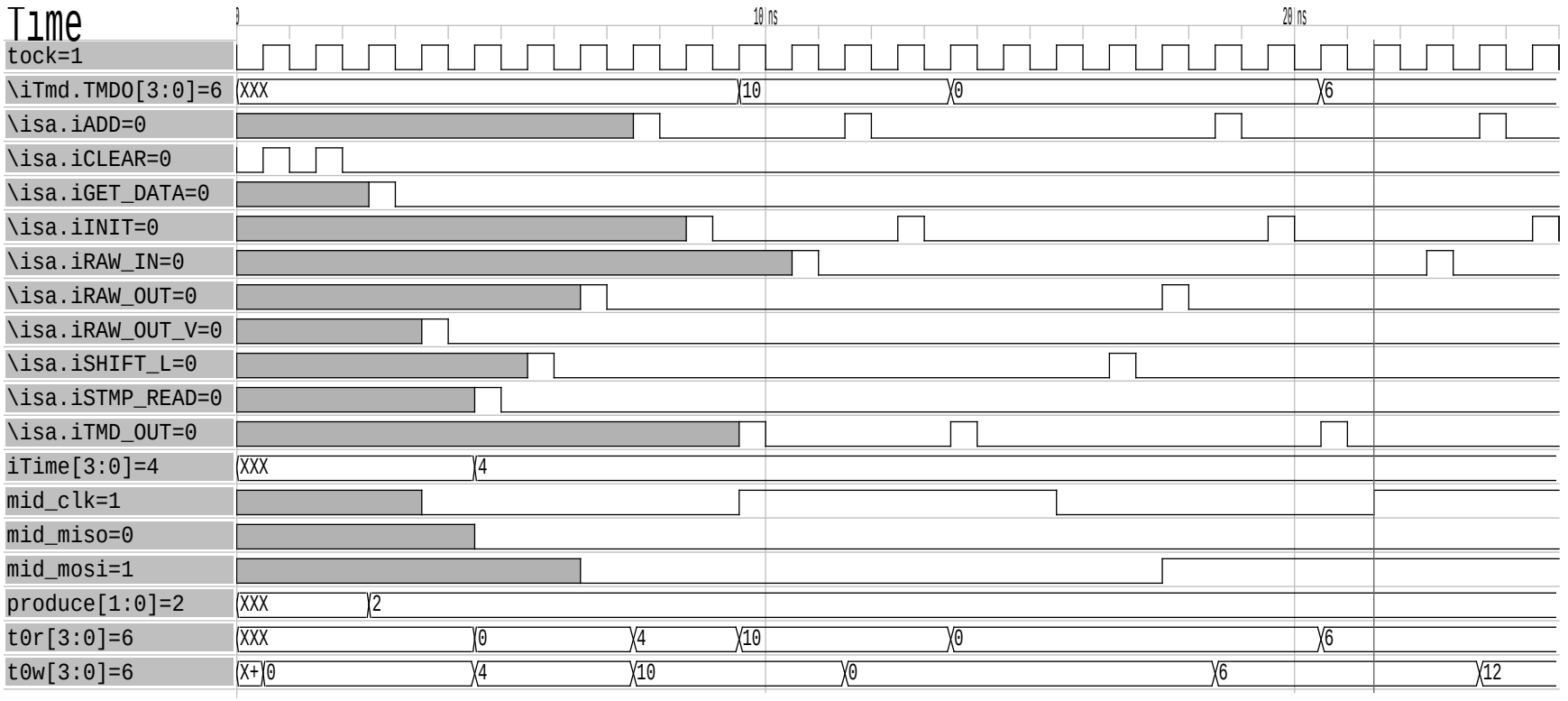


Figure 7.6 Waveform of Assertion 7.99 using $sCPB = 12$

the *consume* channel in the abstract specification of Section 6.5 is not fixed. Hence, verifying the latency of the complex channels of the abstract specifications could be tricky. However, measuring the total number of clock cycles between two consecutive transmissions would provide good evidence on the performance of such a complex channel. This delay shall be called the *throughput delay* and measuring it between two successive transmissions helps determine the practical bit-rate or *throughput* of the channel, as opposed to the nominal bit-rate defined in Eqns. 6.1 or 6.4.

This delay could be verified using the following abstraction:

$$Produce_{Delay}(n) = produce?data \rightarrow \text{TOCK}(n)\,;\, Produce_{Delay}(n) \tag{7.107}$$

$$\Box\ tock \rightarrow Produce_{Delay}(n) \tag{7.108}$$

Because there is no overhead between two consecutive data transmissions in the Uart specification, it turns out that the practical bit-rate is equivalent to the nominal bit-rate:

$$\text{Uart}_{ThroughputDelay} = uTD = \text{Uart}_{ClkWord} \tag{7.109}$$

$$\text{Uart}_{BufThroughputDelay} = uBTD = \text{Uart}_{BufClkWord} \tag{7.110}$$

However, because the Spi protocol specification allows for an empty bus cycle between two consecutive data words, the throughput delay for Spi is:

$$\text{Spi}_{ThroughputDelay} = \text{Spi}_{TD} = (DataSize + 1) \times sCPB \tag{7.111}$$

The following assertions verified the throughput of the specification channels:

$$Produce_{Delay}(uTD) \sqsubseteq_{\text{T}} \text{Uart}_{\text{SPEC-CHAN}} \setminus \Sigma\backslash(\{|produce|\} \cup \{tock\}) \tag{7.112}$$

$$Produce_{Delay}(\text{Spi}_{TD}) \sqsubseteq_{\text{T}} \text{Spi}_{\text{SPEC-CHAN}} \setminus \Sigma\backslash(\{|produce|\} \cup \{tock\}) \tag{7.113}$$

Because $Produce_{Delay}(n)$ forces at least n clock cycles between two consecutive *produce* events, but allows for an additional arbitrary number of clock cycles after n and before the next produce event, it follows that if assertion 7.112 is true, then so is:

$$Produce_{Delay}(uTD - \text{D}) \sqsubseteq_{\text{T}} \text{Uart}_{\text{SPEC-CHAN}} \setminus \Sigma\backslash(\{|produce|\} \cup \{tock\}) \tag{7.114}$$

For all $uTD \geq \mathrm{D} \geq 0$ but *not* for $\mathrm{D} < 0$.

While the latency checks in Section 7.8.1 implicitly verify functional correctness too, the throughput checks discussed in this section do not. Hence, it is important that the functional correctness of the complex channels is always verified, as discussed in Sections 7.6 and 7.7.

Similar checks were performed to verify the throughput of the ISA-*Oriented Specifications*:

$$Produce_{Delay}(uTD + 1) \sqsubseteq_{\mathrm{T}} \textsc{Uart}_{\textsc{isa-chan}}\,' \setminus \Sigma\backslash(\{|produce|\} \cup \{tock\}) \quad (7.115)$$

$$Produce_{Delay}(uBTD + 2) \sqsubseteq_{\mathrm{T}} \textsc{Uart}_{\textsc{isa-buf-chan}}\,' \setminus \Sigma\backslash(\{|produce|\} \cup \{tock\}) \quad (7.116)$$

$$Produce_{Delay}(\textsc{Spi}_{TD} + 2) \sqsubseteq_{\mathrm{T}} \textsc{Spi}_{\textsc{isa-chan}}\,' \setminus \Sigma\backslash(\{|produce|\} \cup \{tock\}) \quad (7.117)$$

Assertions 7.115 to 7.117 show a slight different (1 or 2) clock cycles overhead between the throughput performance of the ISA-*Oriented Specification* and the abstract one for the same protocol.

It can be a little tedious to search for the boundary values of performance specifications, especially in cases where those performance requirements could not be mathematically estimated, as previously demonstrated. In such a case, a construction could be made to extract these performance metrics automatically using FDR (see Section 14.6 of Roscoe [38] for more details).

7.9 Protocol Conformance Verification

The final set of checks was performed to establish that the ISA-*Oriented Specification* of a protocol conforms to the abstract specification of the same protocol. Conformance verification of communication protocols has evolved as a dedicated field of research through an ISO standard [86], which was subsequently formalised into a conformance testing framework by Tretmans et al. [87]. In addition, Krichen and Tripakis [88] further developed conformance testing for real-time systems based on timed automata [89]. In this section, a more ad-hoc approach to conformance verification is taken where conformance is checked through the construction of a set of complex channels. One end of the complex channel is a transmitter specified in one abstraction and the other end is a receiver of the same protocol specified using another abstraction.

Two Uart conformance channels were constructed as follows:

$$\textsc{Uart}_{\textsc{conf-chan-1}}'' = \tag{7.118}$$

$$\textsc{Uart}_{\textsc{isa-tx}}[\![dRO0/\textsc{Uart}_{Out}]\!] \underset{\{|\,\textsc{Uart}_{Out}\,|\}}{\|} Phy[\![in, out/\textsc{Uart}_{Out}, \textsc{Uart}_{In}]\!] \tag{7.119}$$

$$\textsc{Uart}_{\textsc{conf-chan-1}}' = \textsc{Uart}_{\textsc{conf-chan-1}}'' \underset{\{|tock,\textsc{Uart}_{In}\,|\}}{\|} \textsc{Uart}_{\textsc{spec-rx}} \tag{7.120}$$

$$\textsc{Uart}_{\textsc{conf-chan-1}} = \textsc{Uart}_{\textsc{conf-chan-1}}' \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.121}$$

$$\textsc{Uart}_{\textsc{conf-chan-2}}'' = \textsc{Uart}_{\textsc{spec-tx}} \underset{\{|\,\textsc{Uart}_{Out}\,|\}}{\|} Phy[\![in, out/\textsc{Uart}_{Out}, \textsc{Uart}_{In}]\!] \tag{7.122}$$

$$\textsc{Uart}_{\textsc{conf-chan-1}}' = \tag{7.123}$$

$$\textsc{Uart}_{\textsc{conf-chan-2}}'' \underset{\{|tock,\textsc{Uart}_{In}\,|\}}{\|} \textsc{Uart}_{\textsc{isa-rx}}[\![dRI0/\textsc{Uart}_{In}]\!] \tag{7.124}$$

$$\textsc{Uart}_{\textsc{conf-chan-2}} = \textsc{Uart}_{\textsc{conf-chan-2}}' \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.125}$$

Notice the use of the *Phy* process described in Section 7.7 to decouple a receiver process from a transmitter process. The Uart conformance channels were verified through a series of refinement checks to establish their correctness. At the time of writing, basic deadlock and timing consistency checks as well as system-level functional checks were carried out, similar to the those in Sections 7.6 and 7.7 in which the Tau Priority Model was used.

Similar Spi conformance channels were constructed:

$$\textsc{Spi}_{\textsc{spec-master}}'' = \textsc{Spi}_{\textsc{spec-master}}[\![sSMosi, sSClk/iMosi, iClk]\!] \tag{7.126}$$

$$\textsc{Spi}_{\textsc{isa-slave}}'' = \textsc{Spi}_{\textsc{isa-slave}}[\![dRI0, dRI2/iClk, iMosi]\!] \tag{7.127}$$

$$\textsc{Spi}_{\textsc{conf-chan-1}}' = \textsc{Spi}_{\textsc{spec-master}}'' \underset{\{|iMost,iClk,tock|\}}{\|} \textsc{Spi}_{\textsc{isa-slave}}'' \tag{7.128}$$

$$\textsc{Spi}_{\textsc{conf-chan-1}} = \textsc{Spi}_{\textsc{conf-chan-1}}' \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.129}$$

$$\textsc{Spi}_{\textsc{spec-slave}}'' = \textsc{Spi}_{\textsc{spec-slave}}[\![sSMosi, sSClk/midMosi, iClk]\!] \tag{7.130}$$

$$\textsc{Spi}_{\textsc{conf-chan-2}}'' = \textsc{Spi}_{\textsc{spec-slave}}'' \underset{\{|midMosi|\}}{\|} Phy[\![in, out/iMosi, midMosi]\!] \tag{7.131}$$

$$\textsc{Spi}_{\textsc{isa-master}}'' = \textsc{Spi}_{\textsc{isa-master}}[\![dRO0, dRO2/iClk, iMosi]\!] \tag{7.132}$$

$$\textsc{Spi}_{\textsc{conf-chan-2}}' = \textsc{Spi}_{\textsc{conf-chan-2}}'' \underset{\{|iMost,iClk|\}}{\|} \textsc{Spi}_{\textsc{isa-master}}'' \tag{7.133}$$

$$\textsc{Spi}_{\textsc{conf-chan-2}} = \textsc{Spi}_{\textsc{conf-chan-2}}' \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.134}$$

It was possible to establish the functional correctness of $\text{SPI}_{\text{CONF-CHAN-2}}$ without using the Tau Priority Model because the construction did not involve the use of the *SpecTimer* process on the contrary to the functional verification of $\text{SPI}_{\text{CONF-CHAN-1}}$.

The DDC units in the $\text{SPI}_{\text{ISA-SLAVE}}$ process act as a *Phy* buffering physical interface changes and decoupling the ISA process from such changes, hence no *Phy* process was required in the construction of $\text{SPI}_{\text{CONF-CHAN-1}}$ in Eqn. 7.129. On the contrary, the construction of $\text{SPI}_{\text{CONF-CHAN-2}}$ in Eqn. 7.134 required a *Phy* process, because physical interface changes triggered by the $\text{SPI}_{\text{ISA-MASTER}}$ process needed to be decoupled from the $\text{SPI}_{\text{SPEC-SLAVE}}$ process, which is modelled simplistically and hence does not integrate the *Phy* mechanism that exists in the ISA system.

Finally, as already demonstrated, many different specifications (ISA or abstract) are considered equivalent if the selected functional or performance properties are found to be equivalent using the selected equivalence algorithm. For example, Assertions 7.67 and 7.68 verified the functional equivalence of two different ISA-*Oriented Specification*s of the UART protocol. They were both found to be functionally equivalent to a single-entry buffer. This meant that a conformance channel construction, which uses two different ISA-*Oriented Specification*s of the UART protocol is possible: one that makes use of an SBC unit, and another that does not. Using the respective specifications defined in Section 6.4.1 and their interfaces defined in Section 7.6, the following two conformance channels were constructed:

$$\text{UART}_{\text{CONF-CHAN-3}}\,\prime = \text{UART}_{\text{ISA-RX}}\,\prime \underset{\{\!|tock,pUART|\!\}}{\|} \text{UART}_{\text{ISA-BUF-TX}}\,\prime \tag{7.135}$$

$$\text{UART}_{\text{CONF-CHAN-3}} = \text{UART}_{\text{CONF-CHAN-3}}\,\prime \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.136}$$

$$\text{UART}_{\text{CONF-CHAN-4}}\,\prime = \text{UART}_{\text{ISA-BUF-RX}}\,\prime \underset{\{\!|tock,pUART|\!\}}{\|} \text{UART}_{\text{ISA-TX}}\,\prime \tag{7.137}$$

$$\text{UART}_{\text{CONF-CHAN-4}} = \text{UART}_{\text{CONF-CHAN-4}}\,\prime \setminus \Sigma\backslash(aBuffer \cup \{tock\}) \tag{7.138}$$

$\text{UART}_{\text{CONF-CHAN-3}}$ and $\text{UART}_{\text{CONF-CHAN-4}}$ were verified for deadlock freedom, timing consistency and functional correctness through establishing their equivalence to a buffer.

7.10 Results and Summary

The model-checking approach demonstrated in this chapter verifies essential functional and timing properties of the developed configurable system in Sections 7.3

to 7.5. In addition, the power of the modelled configurable system along with its ISA interface is demonstrated through the system-level functional and performance verification in Sections 7.6 and 7.8. The two independent specification techniques demonstrated in Chapter 6 are used in an interoperable manner to construct complex communication channels in Section 7.9. The model-checking of those complex channels against predefined functional and performance specifications gives further proof of the equivalence between the two independent specification styles. This is essential for verifying the validity of the configurable system and the ability of its ISA interface for specifying and verifying communication protocols.

The evaluation in this chapter was performed using similar hardware/software configurations to the ones used for the metrics extraction in Chapter 5: FDR 2.91 Academic Release with the added optimisations described in Section 4.5, Linux kernel version 2.6.32, and a PC with 8 Cores each of which is a 3.16 GHz Intel Xeon processor. All processors share 40 GB of main memory.

The total number of assertions used through the evaluation process was 122. Throughout the functional evaluation of the system, the following configuration was sufficient:

$$TimeSize = 3 \tag{7.139}$$

$$DataSize = 2 \tag{7.140}$$

However, the performance evaluation of the SPI protocol required at least the following configuration:

$$TimeSize = 4 \tag{7.141}$$

$$DataSize = 2 \tag{7.142}$$

$$\text{SPI}_{ClkPerBit} = 12 \tag{7.143}$$

Attempts to run all assertions sequentially in a single instance of FDR with $TimeSize = 4$, $DataSize = 2$ and a system with 3 DDC units proved unsuccessful. Memory profiling showed that FDR was not designed for verifying systems with such a large number of assertions. This is evident in the ever-increasing memory used by FDR. By attempting to verify all 122 assertions in one run, FDR exhausted all the available 40 GB of memory before it was terminated by the host kernel. It was noticed that FDR was not releasing the used memory after finishing the verification of an assertion before starting the next one.

It was also noticed that performing all the checks using a single SYSTEM construction that guarantees all assertions to be true was not possible. This would entail running all checks on processes similar to $\text{SYSTEM}_{three\text{-}bit}$ using the maximum requirement for all parameters (i.e. $TimeSize = 4$). By doing so, results for a set of 8 system-level assertions could never be obtained. The expected cause for this is the fact that current implementation of FDR always explicates the left process in a refinement check. An experiment for verifying Assertion 7.40 from Section 7.4 using $TimeSize = 4$ and $DataSize = 2$ ran for over 14 hours, consumed 34.83 GB of main memory and explicated 522.8 million states before the kernel ran out of memory and FDR was terminated gracefully.

Hence, for the purpose of memory optimisation, the large number of assertions were split into 9 goal-based sets of assertions. Each set of assertions was placed into a standalone CSP script which could be run by FDR. This helped reduce the memory profile for the following two reasons.

1. The memory used by one set of assertions could be freed once FDR had finished evaluating them since the whole instance of FDR would terminate.

2. In order to reduce memory consumption and also verification time, it was possible to configure each set of assertions with suitable values for *TimeSize*, *DataSize*, $\text{SPI}_{ClkPerBit}$ and $\text{UART}_{ClkPerBit}$; in addition to the appropriate system construction.

Running the 9 different sets of assertions in sequence took 10 minutes and 23 seconds. However, since the PC used had 8 identical 3.16 GHz Intel Xeon processors, the evaluation script was changed to dispatch more than one set of assertions in parallel. This is done with the condition that no more than R instances of FDR could run at the same time. Setting R = 6 would consume at most 75% of the available processing power. Under this configuration, the whole suite took 3 minutes and 6 seconds where all 122 assertions were found to be true. This is an improvement of 437 seconds or 70% over the sequential approach.

If FDR (latest release version 2.91, as of August 2012) did not always explicate the left hand side process and explicated parts of the process when required instead, then the refinement checks in Section 7.4 could have been possible for larger configurations ($TimeSize \geq 4$ and $DataSize \geq 2$). This could have also been useful for the UART protocol. Typical implementations of this protocol use either 8 or 16 data bits. The evaluation in this chapter was limited to a maximum of 3.

The verification of the configurable system was only possible through the identification of interesting constructions and their respective ISA-*Oriented Specification*s and the subsequent verification of specific functional and performance properties of such constructions. The correctness of the system with larger parameters is inferred through the verification of such a system using more practical parameters.

7.11 Future Work

The performance specifications and their verification performed in Section 7.8 were cycle accurate. The constructed throughput and latency specifications of the complex channels were specified to the exact cycle. The throughput verification could be adjusted to allow for a window of non-determinism.

Future work includes the addition and verification of such non-determinism for latency and potentially other performance specifications. This would also facilitate the performance verification of the constructed conformance channels where the different specification styles are likely to introduce a small window of non-determinism.

Future work also includes the specification and verification of *multiple tock domains* which would entail building processes by compiling different subprocesses using different *tock* events. This should provide a great tool for the verification of systems with multiple clock domains. For example, an Spi slave could use the clock generated by the master process as its internal *tock* event, while the master uses explicit *tock* events.

The identification of further functional and performance specifications and their verification would provide an endless source of possible extensions to the framework presented in this thesis.

CHAPTER 8

Conclusions

8.1 Motivation and Chapter Structure

As the last chapter, this chapter presents the overall conclusions to the thesis. First, Section 8.2 presents a brief summary to all previous chapters. Then, Section 8.3 discusses the overall contribution of the thesis. Possible future work is discussed in Section 8.4. Finally, Section 8.5 presents the concluding notes and remarks.

8.2 Thesis Summary

To manage the inherent complexity of a configurable performance-critical communication system, a design and verification framework for such a system was demonstrated. This was achieved through the separation of functional aspects into independent hardware building blocks with clear interfaces and functions, which could be verified at the block level.

Chapter 1 presented a brief introduction to the challenges faced by configurable real-time communication systems, which inspired the inception of this thesis. The chapter also summarised the challenges in the form of a problem statement and presented a brief background to the overall proposed modelling and verification framework.

Chapter 2 presented a brief theoretical background and a structured analysis of the requirements of configurable communication systems. The chapter focused on the semantic and orthogonality analysis of common features of selected communication protocols and aimed to establish an orthogonal set of requirements.

Then, Chapter 3 presented the first attempt at modelling the proposed configurable communication system, along with its configurable blocks. A brief background which highlights the different possible approaches for providing hardware configurability was presented. Then the chosen specification and modelling language (CSP) was briefly discussed. The chapter continued to present a *Data and Control Multiplexing* approach for providing the needed configurability. It used a bidirectional pipeline which multiplexed control tokens along with a universal data-type. The modelling of the different configurable units was aided greatly with the visualisation of CSP processes compiled into state machines and then automatically converted into graphic form. These state machine visualisation techniques were also presented in Chapter 3. Finally, Chapter 3 presented the metrics of the demonstrated models. The chapter concluded with the need of complexity analysis techniques for hardware design and model-checking.

Chapter 4 analysed the complexity of the model-checking technology through analysing the complexity of the underlying state machine metrics. The chapter demonstrated how such metrics could be transformed into complexity equations, which could then be analysed in terms of their asymptotic behaviour with respect to a specific configuration parameter. The chapter also addressed the possible fallacy of making conclusions about the complexity of the formal models based on the performance of the model-checking tool. The chapter demonstrated the use of the proposed complexity analysis techniques in analysing the complexity of the data and control multiplexing approach to configurability, which was discussed in Chapter 3.

Chapter 5 discussed the final modelled configurable communication system. First, the evolution from a pipeline of control tokens to the final hierarchical control of the functional blocks was discussed. Then, a formal specification of each modelled configurable unit was presented along with the ISA interface for configuring the functional units and performing data I/O. The possible system constructions that meet the demands of the modelled protocols were then discussed. The chapter then presented the metrics of a basic system construction with respect to two different configuration parameters along with a complexity analysis of those metrics. The chapter concluded with a summary and possible future work.

Chapter 6 demonstrated ISA-*Oriented Specification* as a technique for configuring the underlying formal system. ISA-*Oriented Specification* was also demonstrated as a useful technique for providing functional and performance specifications of communication protocols. Chapter 6 also discussed the abstract specifications of communication protocols which are independent of the configurable system and its ISA interface.

Those independent specifications are intended as reference models for checking the conformity of the respective ISA-*Oriented Specification*s.

Finally, Chapter 7 discussed the verification of the modelled system through model-checking. Basic functional and timing properties were first checked. The specifications of the communication protocols discussed in Chapter 6 were verified independently. Then, those specifications were verified at the system level through the construction and functional verification of complex communication channels composed of a transmitter and a receiver at the same abstraction level. Performance aspects of different specifications of the protocols were verified through the verification of the performance of the constructed complex channels. Chapter 7 also briefly discussed and demonstrated the *Tock-CSP Waveform Generator*, which was instrumental in checking the functional and performance properties of the complex channels. Chapter 7 demonstrated an ad hoc approach to protocol conformity checking, whereby abstract specifications of a protocol were verified with respect to an instance *configuration* of the modelled configurable communication system. The conformity checks that were carried out not only confirm protocol functional conformance, but also its timing conformance.

8.3 Contributions

This section first outlines the overall contribution of this integrated modelling and model-checking framework, then contributions of high academic interest are outlined in Section 8.3.1 and finally the general contributions are highlighted in Section 8.3.2. Those general contributions are considered to have high impact on knowledge transfer and adoption challenges facing formal methods in general and model-checking techniques in particular.

The individual aspects of analysis of communication protocols [23], hardware decomposition into functional blocks [90], protocols standardisation and formal specification [24, 42], conformance checking of protocols [87] and timing requirements validation [91] are techniques that have been studied individually for many years.

The novelty of the work presented in this thesis rests in the overall integrated formal framework for the design and model-checking of functional and performance specifications of communication systems and protocols. It contributes to the ever-increasing interest in the verification of real-time communication systems.

8.3.1 Academic Contributions

The model-checking complexity analysis techniques presented in Chapter 4 enabled the complexity of CSP processes to be expressed in terms of configurable parameters, data-types or state variables. The resulting complexity formulae can then be analysed asymptotically, which gives great insight on how each analysed configuration parameter affects the overall complexity of such a process. These state machine complexity analysis techniques are considered to be one of the major contributions of this thesis, as they:

- help in the objective definition of the state-space explosion phenomena well-known to model-checking techniques;
- enable subsequent optimisations to the associated formal models;
- provide an objective mechanism for assessing the dependence of a configurable unit on a data-type, configuration parameter or shared variable; and
- help solve the scalability issues that typically hinder the application of model-checking techniques and enable model-checkers to handle much larger specifications.

Such issues have been the focus of academic research for a number of years.

The ISA-*Oriented Specification* technique modelled in Chapter 5 and later demonstrated in Chapters 6 and 7 for the modelling and verification of functional and performance aspects of communication protocols represent an extendible and generic formal modelling interface, which minimises the efforts and technical expertise needed for producing detailed formal models. It uses the underlying configurable blocks for the specification and verification of the protocols. This approach is seen as a significant contribution in the use of CSP for the verification of large and complex state-of-the-art communication systems.

8.3.2 General Contributions

The integration of performance modelling and model-checking in a general CSP framework is a promising approach. This is achieved through the use of abstract tock-CSP performance specifications and also ISA-*Oriented Specification*s for modelling performance aspects of protocols. Then, by using the relevant refinement model including the Tau Priority Model, performance checks can be carried out at an early stage of

the design cycle. This provides verified and cycle-accurate performance specifications of the system. The thesis demonstrates this integrated approach and is believed to be the first project to use the newly released Tau Priority Model.

The complexity and grand scale of recent design problems, especially in multi-core parallel systems with multiple clock domains highlight the importance of formal design and verification and the automation of its tools. Though FDR proved useful for designing systems and protocols in the past, it requires a good knowledge of CSP and its theories. With the rising acceptance of formal methods in the industry, evidenced by the increasing adaptation of such techniques for hardware design and verification, there is a great opportunity for increasing the adaptation of CSP if the usability gap is bridged. With the addition of the Tau Priority Model model it becomes even more important to address the usability of CSP. The modelling visualisation techniques depicted in the State Machine Visualiser, which was presented in Chapter 3 and the verification visualisation techniques depicted in the Tock-CSP Waveform Generator, which was presented in Chapter 7 are seen as considerable contributions to the usability and adoption challenges facing formal design and verification in general and refinement-based model-checking in particular.

8.4 Future Work

Future work of the individual chapters has been discussed in detail in the respective chapters. See Sections 4.9, 5.7, 6.7 and 7.11 for more details.

This section discusses overarching future work that transcends individual aspects. Such future work includes addressing the scalability issues of the suggested model-checking framework. For example, in addition to complete complexity analysis through the analysis of the full complexity semantics of CSP and the incorporation of those semantics into a static complexity analyser, it would also be useful to perform further performance and memory profiling and optimisation to the full model-checking tool chain. This would help identify any other complexity issues to the refinement algorithms and verification tools used.

In addition, a more scalable CSP model-checker, which benefits from the ever-increasing parallelism existing in state-of-the-art computer systems is desirable. The notion of employing a technological advancement in the development of the next technological advancement is called bootstrapping. Such an approach has been used in the development of computer architectures for many decades. The development of model-checking algorithms and tools that take advantage of the recent trend of

parallel computing is seen as an essential step in scaling the analysed systems after exhausting all the scalability and optimisation options for the single-threaded model-checker.

Future work that is more general would address implementation aspects of the configurable hardware system. For example, the autogeneration of abstract state machines, as opposed to the fully enumerated state machines presented in Section 3.5. In addition, the investigation of a CSP to HDL interface similar to [76] and [78] which would help integrate the modelling and specification framework discussed here to the current hardware implementation technologies.

Finally, abstract waveforms could be automatically generated to act as blueprints for a protocol specification or hardware implementation. An abstract waveform compiler could analyse a set of traces or CSP specifications and produce a set of abstract waveforms, where some data values could be designated the value *do not care*. By doing so, a single waveform could represent a large set of traces or a full CSP process.

8.5 Final Words

This thesis presents a modelling and model-checking framework for the specification and verification of a complex configurable communication system. It examines many aspects of the model-checking technology from scalability and complexity analysis of the modelled system, the orthogonality of different building blocks with respect to a specific function or data-type, scalability of the model-checker itself, the model-checking algorithms used and the modelling and model-checking interfaces. The thesis demonstrated that there are issues and problems to be solved and lessons to be learnt in every examined aspect of the technology. For decades model-checking has been seen as a *push button* technology [92]. By writing this thesis, the author hopes that the work presented constitutes a step towards achieving that ultimate goal of a fully automated modelling and model-checking framework for real-world complex systems.

Bibliography

[4] Electron. Industries Assoc. (EIA), "Recommended Standard (RS)-232: Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange," EIA, Tech. Rep., 1969.

[5] Freescale Semiconductor Inc., *Motorola PowerPC 5121e's Serial Peripheral Interface (SPI)*, Texas, USA, 2007.

[6] Int. Organization for Standardization (ISO), *Std. 11898-1 - Road Vehicles - Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling*, ISO, 2003.

[7] Inst. of Elect. and Electron. Eng. (IEEE), "Carrier Sense Multiple Access with Collision Detection - Access Method and Physical Layer Specifications," IEEE, Tech. Rep., 2008.

[8] Philips Semiconductors, *Inter-Integrated Circuit (I^2C) Bus Specification*, Amsterdam, Netherlands, 1982 (Original) and 2007 (Version 3).

[9] Philips Semiconductors, *Inter-Integrated Circuit Sound (I^2S) Bus Specification*, Amsterdam, Netherlands, 1986 (Original) and 1996 (Revised).

[10] Intel Corporation, "Universal Serial Bus Transceiver Macrocell Interface Specification," Intel, Tech. Rep., March 2001.

[11] Gregor von Bochmann, "Finite State Description of Communication Protocols," *Comput. Networks*, vol. 2, pp. 361–372, 1978.

[12] F. Müffke, "A Better Way to Design Communication Protocols," Ph.D. dissertation, University of Bristol, May 2004.

[13] P. Böhm and T. Melham, "A Refinement Approach to Design and Verification of On-Chip Communication Protocols," in *Proc. 8th Int. Conf. Formal Methods in Comput.-Aided Design*, 2008.

[14] Advanced RISC Machine (ARM), "Advanced Microcontroller Bus Architecture (AMBA^{TM}) Advanced Peripheral Bus Protocol," ARM, Tech. Rep., 2010.

[15] M. Leuschel and M. Butler, "ProB: A Model Checker for B," in *Formal Methods Europe (FME)*. Springer, 2003, pp. 855–874.

[16] A. N. Parashkevov and J. Yantchev, "ARC - A Tool for Efficient Refinement and Equivalence Checking for CSP," in *Int. Conf. Algorithms and Architectures for Parallel Process.*, 1996, pp. 68–75.

[17] J. Sun *et al.*, "Model Checking CSP Revisited: Introducing a Process Anal. Toolkit," in *Int. Symp. Leveraging Applicat. of Formal Methods, Verification and Validation*, 2008, pp. 307–322.

[18] Formal Syst. (Europe) Ltd., *Failures-Divergence Refinement FDR2, User Manual*, 2nd ed., May 2010.

[19] J. Ouaknine, "Discrete Anal. of Continuous Behaviour in Real-Time Concurrent Syst." Ph.D. dissertation, Oxford University, 2001.

[20] H. A. Simon, "The Organization of Complex Syst." in *Hierarchy Theory - The Challenge of Complex Syst.*, H. Pattee, Ed. New York: George Braziller, 1973, pp. 1–27.

[21] H. Zimmermann, "The ISO Model of Architecture for Open Syst. Interconnection," *IEEE Trans. Commun.*, vol. 28, pp. 425 – 432, 1980.

[22] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *Proc. Assoc. for Computing Machinery (ACM) Symp. Commun. Architectures & Protocols*, 1990, pp. 200–208.

[23] S. Furber and J. Spars, *Principles of Asynchronous Circuit Design: A Syst. Perspective*. Kluwer Academic Publishers, Boston, 2001.

[24] Open Microprocessor Initiative (OMI), *Peripheral Interconnect (PI)-Bus*, OMI Std. 324, 1994.

[25] Maxim Integrated Products, Inc., *+5V-Powered, Multichannel RS-232 Drivers/Receivers*, California, USA, 1987.

[26] Robert Bosch Gmbh, *CAN Specification*, 2nd ed., Postfach 50, D-7000 Stuttgart 1, Germany, 1991.

[27] Microchip Technology Inc., *High-Speed CAN Transceiver Datasheet*, Arizona, USA, 2003.

[28] F. Hartwich and A. Bassemir, "The Configuration of the CAN Bit Timing," in *Proc. 6th Int. CAN Conf.*, November 1999.

[29] D. May, *The XMOS XS1 Architecture*, XMOS Ltd., Bristol, United Kingdom, October 2009.

[30] Microchip Technology Inc., *Peripheral Interface Controller 16F87X Datasheet*, Arizona, USA, 2001.

[31] Altera Corporation, *Stratix GX Field-Programmable Gate Array (FPGA) Family Datasheet*, 2004.

[32] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, pp. 203–215, 2007.

[33] E. L. Horta *et al.*, "Dynamic Hardware Plugins in an FPGA with Partial Run-Time Reconfiguration," in *Proc. 39th Annu. Design Automation Conf.*, 2002.

[34] D. May *et al.*, *XS1 Ports: Use and Specification*, 1st ed., XMOS Ltd., Bristol, United Kingdom, November 2008.

[35] C. A. R. Hoare, *Communicating Sequential Processes*, J. Davies, Ed. Prentice Hall Int., 1985 and 2004.

[36] B. Scattergood, "The Semantics and Implementation of Machine-Readable CSP," Ph.D. dissertation, University of Oxford, 1998.

[37] S. D. Brookes *et al.*, "A Theory of Communicating Sequential Processes," *ACM*, vol. 31, pp. 560–599, 1984.

[38] A. W. Roscoe, *Understanding Concurrent Syst.*, 1st ed., ser. Texts in Comput. Sci. (TCS). Springer, May 2010.

[39] S. Schneider, *Concurrent and Real Time Syst.: The CSP Approach.* New York, USA: John Wiley & Sons, Inc., 1999.

[40] T. Mazur, *LaTeX Symbol Macros for CSP*, 1st ed., University of Oxford, March 2009.

[41] A. W. Roscoe and G. M. Reed, "A Timed Model for Communicating Sequential Processes," in *Theoretical Comput. Sci.*, 1988, pp. 314–323.

[42] K. Seidel, "Case Study: Specification and Refinement of the PI-Bus," in *FME*, 1994, pp. 532–546.

[43] M. Leuschel and M. Fontaine, "Probing the Depths of CSP_M: A New FDR-Compliant Validation Tool," in *Proc. 10^{th} Int. Conf. Formal Methods and Software Eng.* Berlin, Heidelberg: Springer, 2008, pp. 278–297.

[44] L. Freitas and J. Woodcock, "FDR Explorer," *Electr. Notes Theor. Comput. Sci.*, vol. 187, pp. 19–34, 2007.

[45] J. Woodcock *et al.*, "Formal Methods: Practice and Experience," *ACM Computing Surveys*, vol. 41, pp. 19:1–19:36, 2009.

[46] T. Hoare and J. Misra, "Verified Software: Theories, Tools, Experiments (VSTTE) Vision of a Grand Challenge Project," in *Proc. 1^{st} Int. Conf. VSTTE*, 2005, pp. 1–18.

[47] J. C. Bicarregui *et al.*, "The Verified Software Repository: A Step Towards the Verifying Compiler," *Formal Aspects of Computing*, vol. 18, pp. 143–151, 2006.

[48] E. R. Gansner and S. C. North, "An Open Graph Visualization System and its Applicat. to Software Eng." *Software - Practice and Experience*, vol. 30, pp. 1203–1233, 2000.

[49] E. Koutsofios and S. North, *Drawing Graphs with Dot*, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, 1991.

[50] D. Benson, *JGraph and JGraph Layout Pro User Manual*, 5th ed., JGraph Ltd., Northampton, United Kingdom, December 2006.

[51] M. Leuschel and E. Turner, "Visualising Larger State Spaces in ProB," in *Proc. 4th Int. Conf. Formal Specification and Develop. in Z and B*, ser. Lecture Notes in Comput. Sci. (LNCS), vol. 3455. Springer, November 2005, pp. 6–23.

[52] A. W. Roscoe *et al.*, "Hierarchical Compression for Model-Checking CSP," in *Proc. 20th Int. Conf. Tools and Algorithms for the Construction and Anal. of Syst.* Springer, 1995, pp. 133–152.

[53] R. Lazic *et al.*, "On Model Checking Data-Independent Syst. with Arrays with Whole-Array Operations," in *25 Years Communicating Sequential Processes*, 2004, pp. 275–291.

[54] V. L. Allis, "Searching for Solutions in Games and Artificial Intell." Ph.D. dissertation, University of Limburg, 1994.

[55] A. C.-C. Yao, "Some Complexity Questions Related to Distributive Computing (Preliminary Rep.)," in *Proc. 11th Annu. ACM Symp. Theory of Computing*, New York, USA, 1979, pp. 209–213.

[56] E. M. Clarke *et al.*, "Symbolic Model Checking: 10^{20} States and Beyond," in *IEEE Symp. Logic in Comput. Sci.*, 1990, pp. 428–439.

[57] E. M. Clarke and F. Lerda, "Model Checking: Software and Beyond," *J. of Universal Comput. Sci.*, vol. 13, pp. 639–649, 2007.

[58] E. M. Clarke *et al.*, "Symbolic Model Checking with Partitioned Transition Relations," in *Proc. Int. Conf. Very Large Scale Integration*. North-Holland, 1991, pp. 49–58.

[59] E. Turner *et al.*, "Symmetry Reduced Model Checking for B," in *Proc. 1st Int. Symp. Theoretical Aspects of Software Eng.* IEEE Comput. Soc., 2007, pp. 25–34.

[60] R. Nalumasu and G. Gopalakrishnan, "PV: An Explicit Enumeration Model-Checker," in *Formal Methods in Comput.-Aided Design*, 1998, pp. 523–528.

[61] R. Pelanek, "BEEM: Benchmarks for Explicit Model Checkers," in *Int. Symp. Model Checking of Software*. Springer, 2007, pp. 263–267.

[62] G. J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 279–295, 1997.

[63] L. Prechelt, "An Empirical Comparison of Seven Programming Languages," *IEEE Computer*, vol. 33, pp. 23 –29, 2000.

[64] M. Avriel, *Nonlinear Programming: Anal. and Methods.* Courier Dover Publications, 2003.

[65] A. M. Brown, "A Step-by-Step Guide to Non-linear Regression Anal. of Experimental Data Using a Microsoft Excel Spreadsheet," *Comput. Methods and Programs in Biomedicine*, vol. 65, pp. 191–200, 2001.

[66] L. S. Lasdon *et al.*, "Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming," *ACM Trans. Math. Softw.*, vol. 4, pp. 34–50, 1978.

[67] D. Fylstra *et al.*, *Frontline Solvers User Guide*, 11th ed., Frontline Syst. Inc., Incline Village, Nevada, USA, 1996.

[68] D. Fylstra *et al.*, "Design and Use of the Microsoft Excel Solver," *Interfaces*, vol. 28, pp. 29–55, 1998.

[69] W. P. Bowen and J. C. Jerman, "Nonlinear Regression Using Spreadsheets," *Trends in Pharmacological Sciences*, vol. 16, pp. 413–417, 1995.

[70] J. L. Hennessy and D. A. Patterson, *Comput. Architecture: A Quantitative Approach (3^{rd} Edition).* Morgan Kaufmann, 2003.

[71] B. Scattergood and P. Armstrong, CSP_M - *A Reference Manual*, Formal Syst. (Europe) Ltd., January 2011.

[72] K. Paliwoda and J. Sanders, "The Sliding-Window Protocol in CSP," Oxford University Comput. Laboratory, Programming Research Group, Tech. Rep., 1988.

[73] D. May *et al.*, "Designing Chips that Work," in *Philosph. Trans.: Physical Sciences and Eng.* The Roy. Soc., 1992, vol. 339, pp. 3–20.

[74] G. Barrett, "The Semantics and Implementation of Occam," Ph.D. dissertation, Oxford University, 1988.

[75] G. Barrett, "Model Checking in Practice: The Transputer 9000 Virtual Channel Processor," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 69–78, 1993.

[76] H. Treharne *et al.*, "Experiments in Translating CSP ‖ B to Handel-C," in *Proc. 31^{st} Conf. Communicating Process Architectures*, 2008, pp. 115–133.

[77] E. Turner *et al.*, "Automat. Generation of CSP ‖ B Skeletons from xUML Models," in *Proc. 5th Int. Colloq. Theoretical Aspects of Computing.* Berlin, Heidelberg: Springer, 2008, pp. 364–379.

[78] S. Ostroumov and L. Tsiopoulos, "VHDL Code Generation from Formal Event-B Models," in *Proc. 14th Euromicro Conf. Digital Syst. Design.* IEEE Comput. Soc., 2011, pp. 127–134.

[79] G. Lowe, "Casper: a Compiler for the Anal. of Security Protocols," in *Proc. 10th Comput. Security Foundations Workshop*, June 1997, pp. 18 –30.

[80] J. S. Moore, "A Grand Challenge Proposal for Formal Methods: A Verified Stack," in *10th Anniversary Colloq. United Nations University/International Institute for Software Technology*, ser. LNCS. Springer, 2002, pp. 161–172.

[81] S. D. Brookes and A. W. Roscoe, "An Improved Failures Model for Communicating Processes," in *Seminar on Concurrency*, ser. LNCS. Springer, 1985, vol. 197, pp. 281–305.

[82] J. Ouaknine, "A Framework for Model-Checking Timed CSP," in *Inst. of Elect. Eng. Colloq. Applicable Modelling, Verification and Anal. Techniques for Real-Time Syst.*, 1999.

[83] A. W. Roscoe, *Timed Syst. 1: tock-CSP*, 1st ed., ser. TCS. Springer, May 2010, ch. 14, pp. 321–343.

[84] U. Finkelstein *et al.*, *GTKWave 3.3 Wave Analyzer User's Guide*, 3rd ed., December 2011.

[85] D. E. Thomas and P. R. Moorby, *Verilog Hardware Description Language*, IEEE Std. 1364, 1995.

[86] ISO, "Conformance Testing Methodology and Framework," Open Syst. Interconnection, Int. Std. (IS) 9646, 1991.

[87] J. Tretmans *et al.*, "Protocol Conformance Testing: A Formal Perspective on ISO IS-9646," in *Proc. 4th Int. Workshop Protocol Test Syst.* North-Holland Publishing Co., 1992, pp. 131–142.

[88] M. Krichen and S. Tripakis, "Black-Box Conformance Testing for Real-Time," *Form. Methods Syst. Des.*, vol. 34, pp. 238–304, 2009.

[89] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theor. Comput. Sci.*, vol. 126, pp. 183–235, 1994.

[90] K. Yue, "Validating System Requirements by Functional Decomposition and Dynamic Anal." in *Proc. 11th Int. Conf. Software Eng.* New York, USA: ACM, 1989, pp. 188–196.

[91] G. Lowe and J. Ouaknine, "On Timed Models and Full Abstraction," *Electr. Notes Theor. Comput. Sci.*, vol. 155, pp. 497–519, 2006.

[92] E. M. Clarke *et al.*, "Model Checking: Algorithmic Verification and Debugging," *Commun. ACM*, vol. 52, pp. 74–84, 2009.

Appendices

APPENDIX A

Visual Basic Macros using Excel Solver

This chapter lists all the Visual Basic subroutines used to automatically analyse the complexity of the CSP models. These subroutines have been used to generate the results of the case study discussed in Chapter 4.

A.1 Sheet Solving Wrappers

```
'assumptions:
'first row in the sheet lists the values for the independent variable
'those variables are shared between initialisation and solving subroutine
Public F As Integer, P As Integer, D As Integer, S As Double, _
  X As Double, tempCell As Range, StartedCollection As Boolean
Public Const n As Integer = 5
Public Const maxData As Integer = 10
Public Const FS As Integer = 2
Public Const SS As Integer = 12

'solves whole sheet using the iterative GRG subroutine
'and displays consumed time at the end
Sub MacroSolveGeneric()
  Call MacroSolveGenericWrap(True, True)
End Sub

Sub MacroSolveGenericNoIterate()
  Call MacroSolveGenericWrap(True, False)
End Sub

'solves whole sheet
```

```
Sub MacroSolveGenericWrap(Time, Iterate)

  Dim time1 As Double, time2 As Double
  time1 = Timer

  Dim TestString As String, TestArray() As String
  Dim Row As Integer, Result As Integer

  Row = 1

  Result = getResultRow()

  TestString = ActiveSheet.Cells(Row, 1).Text

  While TestString <> "END"

    If TestString = "" Then
      GoTo LoopNext
    End If

    TestArray = Split(TestString)

    If TestArray(0) = "Metric" Then
      Call SolveMatrix(Row, 2, Result, Iterate)
      Result = Result + 1
    End If

LoopNext:
    ' prepare for next iteration
    Row = Row + 1
    TestString = ActiveSheet.Cells(Row, 1).Text

  Wend

EndSolving:
  time2 = Timer
  If Time = True Then
    MsgBox "Time Taken " & Format(time2 - time1, "0.00 \s\ec")
  End If

End Sub
```

A.2 Top-Level Solving Subroutine

```
Sub SolveMatrix(Row, Col, Result, Iterate)

  Call InitMatrix(Row, Col, Result)
```

```
  If Iterate = True Then
    Call RoundAndSolve
  Else
    Call SolveOnce(Range(Cells(F, FS), Cells(F, n + 1)))
  End If

  Call CollectFactors(Row, Col, Result)

End Sub
```

A.3 Initialisation Phase

```
Sub InitMatrix(Row, Col, Result)

  'Row:     measured (dependent) data row
  'Col:     measured data column
  'Result:  row where the results are to be collected
  'P:       predictions row
  'D:       difference row
  'F:       factors row
  'X:       independent variable row
  'S:       column number of SS cells

  P = Row + 1
  D = Row + 2
  F = Row + 3
  X = 1

  Dim fCell(n) As Range

  For I = 0 To (n - 1)
    Set fCell(I) = ActiveSheet.Cells(F, Col + I)
  Next

  Dim xCell As Range
  Dim formula As String

  For I = 0 To (maxData - 1)

    If LTrim(ActiveSheet.Cells(Row, I + FS).Text) = "" Then
      GoTo FinishInit
    End If

    Set xCell = ActiveSheet.Cells(X, I + FS)

    'initialise formulae
```

```
    formula = "=" & fCell(0).Address

    For J = 1 To (n - 1)
      formula = formula & "+" & fCell(J).Address & _
              "*2^(" & xCell.Address & "*" & J & ")"
    Next

    ActiveSheet.Cells(P, Col + I) = formula

    'initialise the error cells
    ActiveSheet.Cells(D, Col + I) = _
      "=(" & ActiveSheet.Cells(P, Col + I).Address & _
      "-" & ActiveSheet.Cells(Row, Col + I).Address & ")^2"

  Next

FinishInit:

  'initialise all factors to 0
  For I = 0 To n - 1
    If RTrim(LTrim(ActiveSheet.Cells(F, Col + I).Text)) = "" Then
      ActiveSheet.Cells(F, Col + I) = 0
    End If
  Next

  'initialise the SS cell
  ActiveSheet.Cells(D, SS) = _
    "=sum(" & ActiveSheet.Cells(D, Col).Address & _
    ":" & ActiveSheet.Cells(D, Col + (maxData - 1)).Address & ")"

  'initialise results area
  Dim TestArray() As String
  ' TestArray(2) is process name
  ' TestArray(3) is metric name
  TestArray = Split(ActiveSheet.Cells(Row, 1).Text)
  ActiveSheet.Cells(Result, 1) = _
    "Factors : " & TestArray(2) & " " & TestArray(3)

  StartedCollection = False

End Sub
```

A.4 Standard GRG Subroutine

```
Sub SolveOnce(ChangeRange)

  SolverReset
```

```
  SolverOptions MaxTime:=100,iterations:=1000,Precision:=0.000001, _
    AssumeLinear:=False, StepThru:=False, Estimates:=1, _
    Derivatives:=1, SearchOption:=1, IntTolerance:=5, _
    Scaling:=False, Convergence:=0.0001, AssumeNonNeg:=False
  SolverAdd CellRef:=Range(Cells(F, FS), Cells(F, n + 1)), _
    Relation:=3, FormulaText:="0"
  SolverOk SetCell:=Cells(D, SS), MaxMinVal:=2, ValueOf:="0", _
    ByChange:=ChangeRange
  SolverSolve True

End Sub
```

A.5 Modified Iterative GRG Subroutine

```
Sub RoundAndSolve()

  Dim iToRound  As Double

  For c = n + (FS - 1) To FS Step -1

    Call SolveOnce(Range(Cells(F, FS), Cells(F, c)))

    iToRound = ActiveSheet.Cells(F, c).Value
    Cells(F, c).Select
    ActiveCell.FormulaR1C1 = Round(iToRound, 0)

  Next c

End Sub
```

A.6 Results Collection for One Complexity Equation

```
Public Function getResultRow() As Integer
  Dim Result As Integer
  Result = 1
  While ActiveSheet.Cells(Result, 1).Text <> "END"
    ' prepare for next iteration
    Result = Result + 1
  Wend

  Result = Result + 5

  getResultRow = Result

End Function

Sub CollectFactors(Row, Col, Result)
```

```
  For c = n To 1 Step -1

    cc = c + 1
    Cells(F, cc).Select

    If StartedCollection = False And _
        ActiveSheet.Cells(F, cc).Value <> "0" Then
      ActiveSheet.Cells(Result, cc).FormulaR1C1 = ActiveCell.FormulaR1C1
      StartedCollection = True
    Else
      If StartedCollection = True Then
        ActiveSheet.Cells(Result, cc).FormulaR1C1 = ActiveCell.FormulaR1C1
      Else
        ActiveSheet.Cells(Result, cc).FormulaR1C1 = ""
      End If
    End If
  Next c

End Sub
```

APPENDIX B

FDR Plugin (FDRlei)

B.1 Introduction and Background

Freitas and Woodcock [44] developed an extension to FDR called *FDR Explorer*, which comes on the form of a set of TCL scripts. It provides useful insights on the TCL interface of FDR and has an interesting set of methods including: Compilation, Extraction, Helper and Auxiliary methods. These provide good knowledge about internal FDR structures accessible through the TCL interface. Perhaps the most interesting functionality of *FDR Explorer* is the graph visualisation function, which uses *JGraph* [50]. However, attempts of using that visualisation function have failed since it uses an old version of the *JGraph* library, which is no longer supported.

The initial motivation behind extending the capabilities of FDR through its TCL interface was the need of better observability of the internal structures of processes and their compositions. In particular, the analysis of the system complexity and how each system component contributed to such complexity was of great interest.

The *transitions* function of the ISM object [18] provided some observability into how a CSP process was compiled. Once the system grows and the number of transitions grows over hundreds of transitions, examining the complexity of those processes simply by analysing their transitions becomes infeasible.

Because examining the transitions of each system component at the command line is tedious, an automated approach was desirable. Sequences of TCL commands were collected into scripts to extract a transitions list of specific processes and perform routine checks on those processes. The scripts also evolved into performing post processing of the transitions lists, including a statistics collection, such as total number

of states and transitions. This gave a quick indication about the complexity of the individual processes, as well as the overall system. It also gave an indication of how each individual process or state variable affected such complexity. However, the more processing and functionality that was added to those TCL scripts, the slower they became. It was apparent at that stage that the TCL scripting approach is not effective in evaluating larger processes with millions of states and transitions. This gave rise to a more efficient approach of having a standalone application for post processing of the information extracted from FDR using the TCL interface. This application was called FDRlei. As more functionality was shifted from the TCL scripts to this standalone application, the TCL script interface evolved into a Middleware. It worked only as a mediator, extracting information from an FDR server and forwarding it to FDRlei, as depicted in Figure B.1.

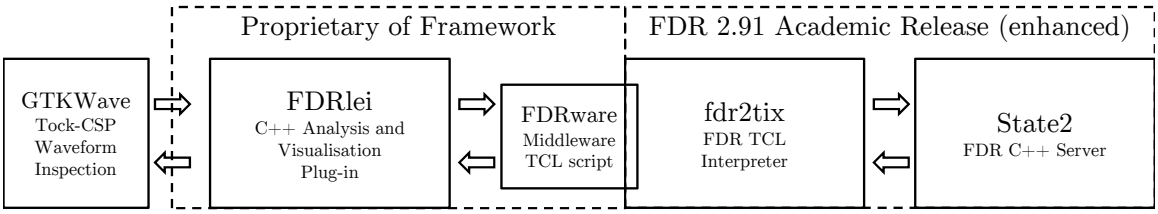


Figure B.1 Structure of the FDRlei Plugin

The structure depicted in Figure B.1 enabled the offloading of almost all processing functions to the FDRlei C++ application.

This C++ *plugin* approach was taken, since it proved more effective in analysing and manipulating large data sets, as opposed to performing the same analysis using TCL. By benchmarking seven programming and scripting languages, Prechelt [63] shows that C++ consistently outperforms TCL in terms of runtime and memory consumption, both of which are valuable resources in a model-checking platform. This trade-off becomes more apparent when handling large data sets. The analysis of large data sets was instrumental to the complexity analysis of large CSP processes.

The implementation approach of FDRlei was to mirror the needed Classes and Objects in the FDR server and augment them with the needed functionality. Additionally, a Class was needed to handle the bidirectional flow of information between FDRlei and external applications such as the FDR server. This Class was called Stream. Refer to Figure B.2 for more information about the object model of FDRlei. Only the objects and functions in FDR that were relevant to FDRlei were implemented. Those helped in importing the desired structures and information from

FDR. They were then augmented with additional objects and functions to provide for the extra analysis and observability.

B.2 Object-Oriented API

The functional interface of the object model in Figure B.2 is discussed in the following sections. This interface is only made available through the external interface of the FDRlei plugin, represented by the FDRlei object which is discussed in Section B.2.8. A brief description of selected objects and functions that are implemented in FDRlei follows.

B.2.1 Stream

This object encapsulates a socket to an external application. It keeps information about the file descriptors for input, output, and error. It has the ability to forward commands to the attached application and, if necessary, collect the response for those commands. It can also be configured to collect error messages. The provided functions are listed below.

- *Execute (command)* $\Rightarrow$ *response*: takes a single parameter which is the command to send to the attached application. It reads back the response from the application and returns this response.
- *Execute (command, expect)* $\Rightarrow$ *result*: can be used if the calling object expects a specific response, but does not need to perform any processing on such a response. The returned boolean value specifies whether the application responded as expected or not.
- *Execute (command)*: can be used if no response at all is expected from executing the command. It is a low-level function and it only sends the command on the output channel attached to the application. The calling object is either not expecting any response back, or needs to manually access the input channel and parse the response one token at a time.
- *Exit*: terminates the external application by sending an exit command.

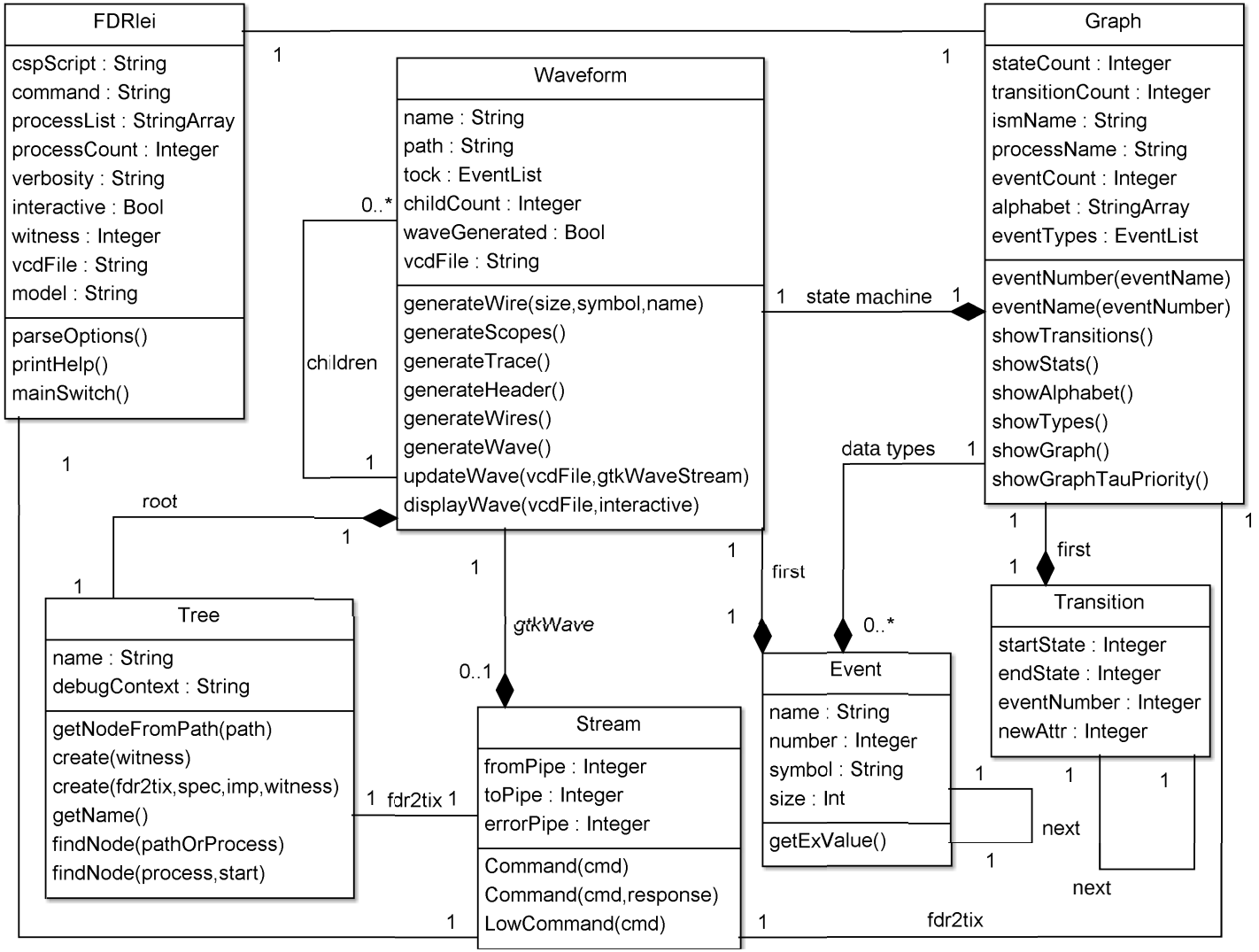


Figure B.2 UML Representation of the Object Model of FDRlei

B.2.2 Transition

An object that encapsulates a single transition of the state machine. It holds information about the starting state, the event involved, and the end state. An ISM can then be represented by a list of Transition objects. This is an abstract data-type with no executable functions.

B.2.3 Event

An object that holds information about a specific event that can be executed by a CSP process. The held information includes the alphabetical event name and the corresponding enumerated event number. It was also extended to hold the symbolic representation of the event, as well as the size of the associated data-type. This additional information was needed for the generation of a waveform abstraction of a failing trace on the form of a Verilog [85] VCD file, as discussed earlier in Section 7.8.1.1. Only one function is associated with this object.

- *GetDataValue* $\Rightarrow$ *data*: checks whether or not the event represented by this object is a compound event of the form $c.t \mid t \in TY$, where TY is the *type* of channel c. This function returns the value t associated with this specific event.

The *GetDataValue* function was useful in dynamically determining the type of each event involved in a failing trace, by analysing the failing trace itself and without access to the CSP source script. This analysis allows for a quick conversion of the failing trace to the desired VCD representation, which can then be examined using a waveform viewer, such as GTKWave. Also, CSP allows for a Cartesian generalisation of the "dot" separator for sets. For example, $c.\mathrm{S}.\mathrm{E}.\mathrm{T} = c.s.e.t | s \in \mathrm{S} \wedge e \in \mathrm{E} \wedge t \in \mathrm{T}$. At the time of writing, only the simple case of $\mathrm{X}.t \mid t \in \mathrm{T}$ is implemented where X can be any channel type, including compound ones.

B.2.4 Waveform

This object is similar to the *Behaviour* object of FDR. It encapsulates the behaviour of a single process and the events list associated with a trace that the process executes or contributes to an overall execution trace. The added functions were used to collect additional information about the executed trace. This additional information was instrumental in translating the trace into a VCD waveform.

- *DisplayWave (fileName, active)* $\Rightarrow$ *gtkStream*: is used to convert the execution trace contained by this node into VCD format, which is then saved into the file name specified by the first parameter. After the wave generation is complete, this function will attempt to invoke the GTKWave application to display the generated waveform. The second parameter specifies whether or not future interaction with GTKWave is required. If so, a Stream object is constructed to keep an I/O socket for interaction with GTKWave. This Stream object is then returned by the *DisplayWave* function.

- *UpdateWave (fileName, gtkStream)* $\Rightarrow$ *status*: if FDRlei was invoked in interactive mode, then GTKWave will also be invoked in interactive mode. This means that interaction with GTKWave will be through its TCL interface. The reader is referred to the User Guide of GTKWave [84] for more information about its TCL interface. This function is called once a new trace has been interactively identified, which can be done either by changing the displayed process name or the failing trace number. The first parameter to this function is the VCD file, and the second parameter is a reference to the Stream object obtained by an earlier call to *DisplayWave*. Finally, the function returns the status of the update.

- *GenerateWave* $\Rightarrow$ *status*: is called by both *DisplayWave* and *UpdateWave*. If the generation process succeeds then the VCD file is saved.

- *GenerateHeader*: generates the header section of the VCD file, including date, time, tool version and timescale.

- *GenerateScopes*: generates the scopes section of the VCD file. At the time of writing, only a top-level scope was supported, which represents the top-level process under scrutiny.

- *GenerateWires* $\Rightarrow$ *status*: generates a symbolic name for each event and stores it in a structure called *eventTypes* held by the *Graph* object. It also finds the type of each event (in case of compound events) and its size. Then it generates a single VCD variable of type *wire* for each event that appears in the trace. Timing events (*tock*s) are essential to generating VCD waveforms and if none are found an error message is displayed and the function fails.

- *GenerateTrace*: converts the trace information into the VCD format. It is assumed that the clock frequency is $1GHz$ and a *tock* event would start the

cycle with the associated *tock* signal being high. Then all events in the trace are generated so that the value of simple events is assumed to be high at the start of the clock cycle and all compound events have their exact value changed, as specified by the trace. Then at the halfway point of the clock cycle ($500ps$) the *tock* is changed to low along with all simple events. Finally, the time is advanced another $500ps$ to finish the cycle. The process continues until there are no more events in the trace.

B.2.5 Tree

This object is similar to the *DebugTree* object of FDR. It encapsulates the top-level of an execution trace as a root *Waveform* object. In addition, it includes information about all the subprocesses that the top-level process is constructed from and their contribution to the top-level trace. Available functions are:

- *Create (stream, spec, imp, witness)* $\Rightarrow$ *tree*: is used to create the tree by running the refinement relation *spec* $\sqsubseteq$ *imp*. If the refinement relation succeeds, then the function fails to produce any traces and subsequently the tree creation fails. If the assertion result is either *xtrue* or *xfalse* then, using the associated *DebugContext* and *DebugTree* objects of FDR, a *Tree* object is constructed.

- *Create (witness)* $\Rightarrow$ *tree*: can be executed using an existing *DebugTree* object when the trace refinement has already been checked by an earlier call to the *Create(stream, spec, imp, witness)* function. It is used for exploring a different witness trace in interactive mode. It is similar to the *Create(stream, spec, imp, witness)* function, except that it uses the *DebugContext* already saved and does not run the refinement assertion again.

- *FindNode (process)* $\Rightarrow$ *node*: takes a process name as a string and finds the node in the current tree that corresponds to that process. It does so by recursively traversing the tree and once a node has been identified, that node is returned.

B.2.6 Graph

This object imports many aspects and functions from the ISM object in FDR. It also adds functions that are useful to the development of the overall framework. Selected functions are briefly discussed here, but are also further discussed in Section 3.5.

- *EventNumber (name)* $\Rightarrow$ *number*: returns the numerical event number when given the alphabetical event name.

- *EventName (number)* $\Rightarrow$ *name*: returns the alphabetical name of the event when given the numerical value of that event. This function is useful when translating state machines and trace information into different visual formats.

- *ShowGraph* $\Rightarrow$ *pdfGraph*: automatically transforms the transitions list into graphical form through the use of the *Dot* language and the associated layout and automatic rendering tools, described by Gansner and North in [48] and also by Koutsofios and North in [49]. FDRlei generates a *Dot* file formatted according to the *Dot* language specification. Then by using the associated toolkit, the *Dot* file is transformed into a PDF file. Section 3.5 provides more details about this function.

- *ShowStats*: prints statistical information about the process. Those statistics include the total number of states, transitions and unique events. The events count includes all possible communications along a compound channel. This function was useful for iteratively analysing the complexity of CSP processes, as discussed in Chapter 4.

B.2.7 SessionLei

As the name implies, this object is used for accessing the corresponding *Session* object in FDR. In addition, a top-level interface to the waveform generation mechanism, as well as the interactive interface with GTKWave was implemented into this object. For brevity, this object is not shown in Figure B.2.

- *ShowWaveform(spec, imp, process, witness, active)* $\Rightarrow$ *vcdWave*: takes a specification process and an implementation process to be checked for refinement. By default it performs the refinement check under the Tau Priority Model. Then, using the provided process name and witness number, the function identifies a *Waveform* object in the debug tree from which a trace is obtained. Finally, using the *DisplayWave* function of the identified *Waveform* object, the trace is converted into the VCD format and subsequently displayed for inspection using GTKWave. The function also has the ability to run in interactive mode, in which case the function waits for further input identifying another trace. This could be done using a process name or a witness number. GTKWave is

automatically updated using the *UpdateWave* function of the *Waveform* object described earlier in Section B.2.4.

B.2.8 FDRlei

This is the top-level object, which provides the command line interface with a number of commands and optional arguments. Those commands wrap and sometimes combine functions in the object model described so far. It first parses the command line arguments and then a main switch selects from a set of possible responses.

- *ParseArguments(argumentCount, argumentList)*: expects at least three arguments: the CSP script file name, the *command* to execute, and a *process list* on the form of "$\{P1, P2, P3, \ldots\}$". See the *SwitchCommand* function below for more information on the possible commands. This object supports a number of optional arguments.
  - v: is used to configure the *verbosity* of the associated FDR server. It is also used by FDRlei to control the amount of reporting generated. Possible values for -v are: "auto", "none", "medium" and "full".
  - f: is used to specify the output file name. This is only useful if the *command* was *graph* or VCD. In the case of the *graph* command, the provided name would be used for the PDF file. In the case of the *vcd* command, the provided name is used for the VCD file.
  - i: specifies interactive mode. At the time of writing, interactive mode was only supported for the *vcd* command.
  - w: is used to specify the witness number for VCD debugging. It should be followed by a numeric value between 0 and 99. If this option is missing, the witness number is assumed to be 0.
  - m: specifies the refinement model to be used in the compilation of processes and the execution of any assertions.
- *SwitchCommand(command)* $\Rightarrow$ *status*: is the top-level switch. The following *command*s are implemented.
  - *stats*: each process in the *process list* is compiled and formatted, then the statistics of each process are displayed.

- *graph*: each process is compiled and the graph representing the state machine of each process is generated and displayed.
- *graph_tau_priority*: similar to the *graph* command, except that each process is compiled using the Tau Priority Model. Then the state machine for each process is generated and displayed.
- *vcd*: assumes that a refinement check is to be carried out, the first process in the *process list* is the *specification* process, the second is the *implementation* process, and the third process (if it exists) is assumed to be a process in the tree of the *implementation* process to be used for debugging. If a third process was not provided, then the debugged process will be the root process (i.e. the *implementation* process itself). Finally, the control is passed to the *ShowWaveform* function of the *SessionLei* object, using the relevant parameters.

The object model is presented here as proof of concept of the functions that were essential in the development of the configurable communication system discussed in this thesis. By providing a detailed documentation of FDRlei, it is hoped that the contribution of this tool to the overall framework is highlighted. Future work would possibly include the integration of those objects and functions into the FDR object model. Alternatively, the FDRlei plugin could be further developed into an independent tool for the debugging, analysis and visualisation of formal models and state machines.