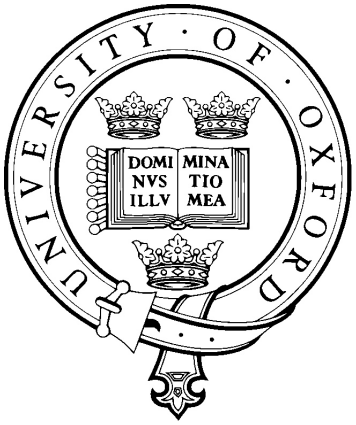
Incremental Modelling for Verified Communication Architectures



Peter C. Böhm

Balliol College

University of Oxford,
Department of Computer Science

Submitted in partial fulfilment for the degree of
Doctor of Philosophy

Trinity 2011

Incremental Modelling for Verified Communication Architectures

Peter Böhm, Balliol College, University of Oxford

Submitted for DPhil. Computer Science, Trinity Term 2011

Abstract

Modern computer systems are advancing from multi-core to many-core designs and System-on-chips (SoC) are becoming increasingly complex while integrating a great variety of components, thus constituting complex distributed systems. Such architectures rely on extremely complex communication protocols to exchange data with required performance. Arguing formally about the correctness of communication is an acknowledged verification challenge.

This thesis presents a generic framework that formalises the idea of incremental modelling and step-wise verification to tackle this challenge: to control the overall complexity, features are added incrementally to a simple initial model and the complexity of each feature is encapsulated into an independent modelling step. Two main strategies reduce the verification effort. First, models are constructed with verification support in mind and the verification process is spread over the modelling process. Second, generic correctness results for framework components allow the verification to be reduced to discharging local assumptions when a component is instantiated. Models in the framework are based on abstract state machines formalised in higher order logic using the Isabelle theorem prover.

Two case studies show the utility and breadth of the approach: the ARM AMBA Advanced High-performance Bus protocol, an arbiter-based master-slave bus protocol, represents the family of SoC protocols; the PCI Express protocol, an off-chip point-to-point protocol, illustrates the application of the framework to sophisticated, performance-related features of current and future on-chip protocols.

The presented methodology provides an alternative to the traditional monolithic and post-hoc verification approach.

Contents

List of Figures

List of Tables

Preface

The research results presented in this dissertation can roughly be split into three main parts: (i) a generic modelling framework together with correctness results; (ii) the application of the framework to the ARM AMBA Advanced High-performance Bus protocol; and (iii) the application of the modelling and verification methodology to the PCI Express protocol.

The individual work on each of these three parts has been published in both peer-reviewed conference papers and journal articles: a paper covering the overall framework was presented at the tenth International Conference on Formal Methods in Computer-Aided Design (FMCAD'10) [Böh10a] in Lugano, Switzerland.

The results of the AMBA case study were accepted at the same venue two years earlier, the eighth International Conference on Formal Methods in Computer-Aided Design (FMCAD'08) [BM08] in Portland. Finally, the application of the framework to PCI Express was published in the eighth ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'09) [Böh09] and the IEEE Transactions on Computer-Aided Design for Integrated Systems and Circuits (TCAD) [Böh10b]. Although not covered in this thesis, the method has also been applied to the PCI Express data-link layer in addition to the transaction layer. The TCAD contribution

covers these results as well.

This dissertation combines the major results from these contributions into a coherent whole with a presentation independent of the text of the papers. The research has been funded by the Engineering and Physical Sciences Research Council (EPSRC) and a donation from Intel Corporation.

Acknowledgements

First and foremost, I would like to thank Tom Melham for his invaluable support during my research for this work. His advice helped me focusing on a coherent research path and remembering the overall picture. Elaborate discussions during our meetings made me look at my work from different perspectives, discovering new aspects, and fully comprehend the work. I am very grateful to Tom for giving me the opportunity to conduct my research in his group.

I am grateful to Intel Corporation for funding my research as well as supporting me with opportunities to present and discuss this work. I thank Jin Yang and Carl Seger from Intel's Strategic CAD Labs for their valuable feedback, many fruitful discussions, and for hosting me various times at Intel to present my work at different stages. I would also like to thank John O'Leary, Jim Grundy, and Sava Kristić for their support.

I am also thankful for the great working environment at the Computer Science Department in Oxford. Many interesting discussions with colleagues and co-workers contributed to and influenced my research, especially with Ziyad Hanna, Sara Adams, Shamal Faily, and John Lyle. I thank Warren Hunt, Kevin Jones, Steve Levitan, Mary Sheeran, Satnam Singh, and Joe Stoy for excellent discussions and comments during their visits to Oxford or meetings at conference. I am

grateful to Steve McKeever and Joël Ouaknine for reading my transfer and confirmation reports and providing helpful feedback during the examinations. I also would like to thank Balliol College and the Computer Science Department for their accommodating environments and their contributions to conference travel expenses. I thank the Engineering and Physical Sciences Research Council (EPSRC) for funding my tuition fees.

Finally, I would like to thank my parents for supporting and encouraging me over many years, and my sister for her patience with my absence and unavailability.

Chapter 1

Introduction

1.1 Problem Statement and Motivation

Modelling and verifying of on-chip communication architectures for modern high-performance chip designs such as many-core processors or complex System-on-Chips (SoCs) is a well-established research challenge in Computer Science [BBC+06]. Such designs integrate large numbers of distributed computation units—CPU cores in the former case and heterogeneous components in the latter—and the design goal is to provide high-performance by parallelizing and distributing computation. As a consequence, correct computation and execution relies on correct and reliable communication among the components. Thus, a verified communication architecture is a highly desirable goal. Because of the concurrent nature of such systems, they are hard to model and verify at the same time. It is well known that concurrency heavily contributes to the state explosion problem in verification [HKV02].

In addition to concurrency, the individual components of the communication

system are complex by themselves. Since the system has to provide error-free, high-performance communication to meet performance requirements, the implemented protocol has to support a broad variety of sophisticated features which also increases the verification complexity significantly. This combination of concurrency and complexity makes the formal verification of high-performance on-chip communication architectures usually infeasible using traditional verification approaches.

The term *traditional verification approach* is used to refer to the well-established methodology of creating a monolithic model that is proven correct using post-hoc verification. Thereby, both, monolithic modelling and post-hoc verification, significantly contribute to the infeasibility of a formal correctness argumentation. This work presents a new methodology that aims at revising this workflow by combining modelling and verification into a single, structured process. The idea is to encapsulate the complexity of protocol features into independent modelling steps and add features incrementally to an initially simple model, hence *incremental modelling*. At the same time, the verification effort is reduced using two main strategies: on the one hand, the verification process is spread over the modelling process and in each step only the newly added part has to be verified. On the other hand, generic correctness results for the framework components allow the verification to be reduced to discharging local assumptions when they are instantiated.

This dissertation answers the following main questions:

- How can we make formal verification of high-performance on-chip communication architectures more feasible, and by doing so, can we also improve the modelling process such that it can be leveraged for the

verification?

- Can we develop a methodology that combines modelling and verification in a homogeneous, well-structured way to create a uniform process?
- Can communication architectures be modelled in a compositional way to allow a straightforward design of protocol families with different, application-specific feature sets?
- Can this be done for "real world" communication systems?

1.2 The Modelling Approach

The main goal of the modelling approach is to improve traditional monolithic modelling and to provide a basis for combining modelling and verification in order to avoid post-hoc verification. To achieve this, incremental modelling is used to construct a complex model as a sequence of well-structured modelling steps, each of which has only a limited, tractable increase in complexity. This step-wise modelling process is interleaved with a verification process that is spread over the modelling steps. The idea is sketched in Figure 1.1.

As the model provides the basis for any verification attempt, reducing the modelling complexity is a natural starting point to increase the feasibility of a verification challenge. Incremental modelling tackles this by splitting a monolithic model into a series of models with increasing complexity such that the complexity gain between two *successive* models is limited and controllable. This modelling chain is constructed using a set of well-defined constructions rules, called *transformations*. Each transformation implements a specific protocol

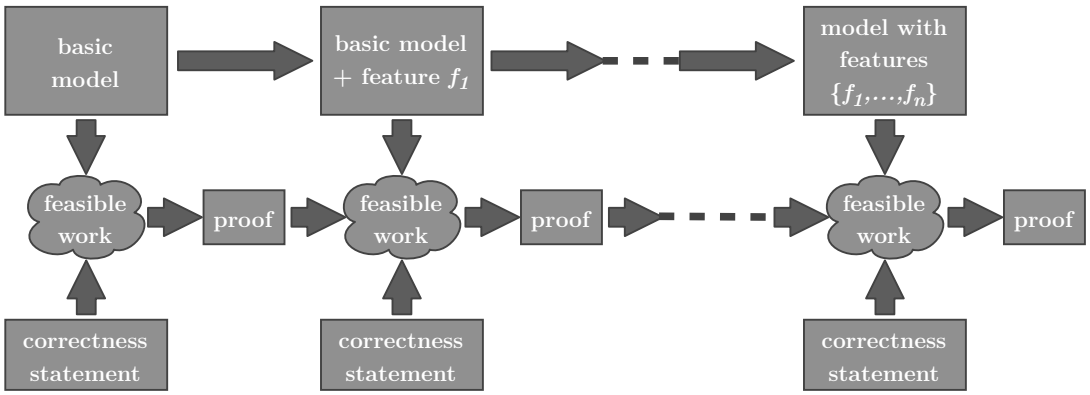


Figure 1.1: Modelling and Verification Approach

feature and this feature's complexity is captured in the transformation. This way a model with a specific feature set can be constructed in a well-structured way.

The generic framework formalises the key concepts of incremental modelling. Every component in the framework is modelled as a Mealy machine and, in order to compose them, the standard Mealy machine model is extended with a synchronous model of communication. On the lowest abstraction level, the framework consists of *abstract components* and *composition rules*. The former are the core building blocks which provide a carefully defined input-output interface to guarantee compositionality, such as buffers or data modification blocks. The latter define how to combine abstract components to more complex constructs, in the simplest case *parallel* and *sequential composition*. Abstract components and composition rules are defined in a generic way using parametrised datatypes and uninterpreted functions. Transformations are constructed using basic components and provide a feature-oriented abstraction.

Using the framework, the process of creating a concrete modelling chain can be summarized by the following steps: initially, a simple model is constructed by instantiating a small number of abstract components with the required datatypes

and concrete functions, and combining them using simple composition operators such as parallel and sequential composition. The main purpose of this simple model is to implement rudimentary data transmission in a model as simplified as possible. The protocol implemented by this model can be quite far from the complete, final protocol. Then, the model is extended with advanced features using transformations.

As illustrated in Figure 1.1, verification is spread over the modelling process to create a homogeneous process of modelling and verification; one of two main strategies to reduce the verification effort. By combining modelling and verification is this way, after each modelling step, already proven properties can be *reused*, together with generic correctness from the framework, to show the correctness of the new, more complex model. This way most of the verification complexity can be reduced to discharging local assumptions on the building blocks.

The development of the framework was driven by the work on the two case studies. Besides using the framework to model and verify crucial parts of two important communication protocols, the diversity of the protocols—bus protocol versus point-to-point protocol, bit-level control signal-based communication versus packet-based communication, etc—highlights the utility and breadth of the approach.

This case study driven research approach also dominated the chronological process which led to the work presented in this thesis. The rest of this section is dedicated to briefly summarising the "research journey" that ultimately resulted in this dissertation. Whereas the natural way of presenting the research results is to start with the generic framework and then illustrate its application using case studies, the research was actually conducted in an almost reversed order:

starting with the idea of a methodology that combines modelling and verification, an initial case study was used to explore different options and starting points. For this purpose, the ARM AMBA 2 Advanced High-performance Bus (AHB) protocol [ARM99] seemed to be a very suitable choice because of its popularity and the free availability of its specification.

The research results of this case study led to initial ideas of a more generic and extensive framework: a framework based on incremental, transformation-based modelling steps with an interleaved verification process. However, this initial "framework" was still very specific to bus protocols. Still, many important insights had already been gained at this stage and fundamental ideas, such as the controlled parallel execution of two copies of a state machine, had been developed.

To further explore these promising initial research results, a more sophisticated case study was needed to examine the usability and potential of the methodology. The PCI Express protocol [PS06] seemed to be a challenging choice: a successful application of the approach to a packet-based point-to-point protocol would highlight the versatility of the framework. Moreover, a transformation-based specification of complex protocol features as implemented by the PCI Express transaction layer is a research challenge in its own right.

During the PCI Express case study, the framework evolved immensely and grew more generic with every feature in order to provide sufficient flexibility. To ensure the development of a unified, generic framework two key aspects turned out to be crucial: even though the PCI Express case study turned out to be much more extensive than the AMBA one, keeping the latter in mind while adapting the framework to PCI Express helped increasing the generality of the framework. It also eased the second important point: the revision of the AMBA case study after

having modelled PCI Express in order to finalise the framework. This revision process increased the maturity of the framework significantly.

1.3 Contributions

The following four paragraphs briefly summarise the main research contributions presented in this dissertation.

The Framework. A framework for the incremental modelling and formal verification of on-chip communication protocols that consists of two main parts: basic components and composition rules, and generic correctness results. The former includes operators that are specific enough to obtain significant correctness results, but generic enough so that only a few of them are sufficient to derive a variety of concrete transformations. The latter reduces the verification effort for an instantiation to discharging local assumptions.

The Transformations. A set of transformations specifying important protocol features independently from the specific protocol. Transformations encapsulate the complexity of the features and cover both bus protocol specific as well as point-to-point protocol specific features. The set of transformations is derived from the features required for the case studies.

The Case Studies: PCI Express and AMBA AHB. Two major case studies to illustrate the breadth and utility of the framework. The case studies where chosen to cover two major protocol classes: point-to-point and shared bus communication architectures. Both topologies implement rather different feature sets as well.

The Formalisation and Automation. A complete formalization of the framework and the case studies in higher order logic using the Isabelle/HOL [NPW02] theorem prover (Isabelle 2009-1 [Isa]) and the integration of NuSMV [CCG+02] into the verification process to reduce manual theorem proving and to ease the applicability of the framework. Technically, the model checker is integrated using the oracle based interface IHaVeIt [Tve05], which has been ported to Isabelle 2009-1 as part of this thesis.

1.4 Outline

Chapter 2 overviews background and key concepts related to the presented work together with the tool environment used for formalisation and automation. Theorem proving and model checking are introduced briefly and their application in the context of this dissertation, thus the concrete tool environment and how the tools are integrated, is detailed. Since concrete models are specified in an executable subset of higher-order logic, the chapter also explicates how executable specifications are exported using the Isabelle code generation framework [Haf09] and how they can be simulated for straightforward sanity checks. The chapter concludes with surveying related work on general protocol verification, especially using state machines.

Chapter 3 introduces mathematical concepts that are used throughout this thesis, and specifies the basic modelling approach for components of the framework: communicating Mealy machines. The actual framework with its basic components is detailed in Chapter 4 which covers the abstract components, the composition operators, and the generic correctness results for both. Chapter 4 concludes with

introducing the concept of a transformation. Transformations extend existing components with specific features and provide a feature-oriented abstraction level between the basic framework components and the actual application of the framework.

Chapters 5 and 6 present the AMBA AHB and the PCI Express case studies. Both chapters define basic models for the respective communication architecture, followed by the steps of the incremental modelling process together with the correctness argumentation.

Finally, Chapter 7 concludes the dissertation by summarising and highlighting the key aspects of the work. Some potential further research directions based on this work are then pointed out, as well.

Chapter 2

Background and Tools

2.1 Theorem Proving – Isabelle/HOL

Theorem proving refers to the process of proving a mathematical theorem deductively using a computer program to check the reasoning steps. Isabelle [NPW02] is an interactive, LCF-style theorem proving framework; LCF [Sco93, Mil72, Gor00]—*Logic for Computable Functions*—is a type-theoretical deductive system and theorem proving logic going back to Scott and Milner which provides terms from the typed λ-calculus and predicate calculus formulae. In general, LCF-style systems are provers, implemented in a strongly-typed language, in which theorems are of a special abstract datatype with only primitive inference rules. The type system then guarantees that new objects of that datatype can only be created using either the primitive inference rules or rules derived from them. This allows for large libraries of theorems with only a small, trusted kernel.

The Isabelle system is generic in the sense that it can be used with a variety of different logics [Pau09a, NPW09, Pau09b] such as first-order logic (FOL), higher-

order logic (HOL), or Zermelo-Fraenkel set theory (ZF). Here, Isabelle is used in version 2009-1 with its HOL instantiation: Isabelle/HOL 2009-1 [Isa]. Note that due to active development of the proof assistant, Isabelle theories are not necessarily backwards compatible.

Isabelle/HOL provides a formal specification language that is inspired by the functional programming language Standard ML [MTH90] and includes constructs to specify datatypes, inductive definitions, recursive functions, and functions with complex pattern matching. Moreover, the language allows the use of polymorphism and uninterpreted functions.

Proofs are written in the structured proof language Isar [Wen09] to provide human readable proof text, and individual subgoals are proven correct using short, tactic-based proof scripts. Proof scripts make use of Isabelle's integrated automatic tools such as the simplifier, the Metis [Hur] automatic theorem prover, or the sledgehammer [Sle] tool, which is used as one of two main efforts to integrate proof automation into the verification workflow. Sledgehammer integrates first-order automatic theorem provers (ATPs) into the Isabelle/HOL system: in Isabelle 2009-1 these are E [Sch02], Spass [Wei99], and Vampire [RV02]. If sledgehammer is invoked, the ATPs are executed in parallel and their execution time is limited by a user configurable time-out. Invoking sledgehammer within a proof causes the current subgoal to be passed to the ATPs together with facts from the current theory context. Facts are selected heuristically and are filtered by relevance. If one of the ATPs finds a proof for a given subgoal, it returns an Isabelle proof script that invokes the Metis prover. Metis is fully integrated into Isabelle/HOL with interfaces going through the kernel. This ensures the correctness and soundness of any proof script generated by sledgehammer that is successfully applied to the

current subgoal.

To interact with Isabelle interactively, the Isabelle/Isar version of Proof General [Asp00], a generic front-end for interactive theorem provers in Emacs, is used.

2.2 Model Checking – NuSMV 2 and IHaVeIt

In contrast to theorem proving, model checking represents a complementary or orthogonal approach to verifying properties of a mathematical representation of a system: it is based on an exhaustive state space exploration to check a property of a system. The system is usually specified as a transition system or state machine, and the property to check is given by a temporal logic formula, an idea going back to Clarke and Emerson [EC80, CE82], and Queille and Sifakis [QS82]. In contrast to interactive theorem proving, model checking is a fully automatic process. A key issue with model checking is the state explosion problem which results in very large transition systems even for reasonably small models. Symbolic model checking tries to tackle the problem by representing the system implicitly using quantified propositional logic. In general, the state explosion problem has been very well studied and there is a large literature on various technical approaches to it. NuSMV 2 [CCG$^+$02] is an open source symbolic model checker integrating BDD-based and SAT-based model checking for the formal verification of finite state systems. Its input language provides constructs to specify finite state machines and to express properties in computational tree logic (CTL) and linear temporal logic (LTL).

To integrate the model checker into the Isabelle/HOL system and to make it

usable as a proving tool, the IHaVeIt [Tve05] tool is used. IHaVeIt is an oracle-based interface written in PolyML. Originally designed to work with Isabelle 2005, it makes external tools such as NuSMV, various SAT solvers, and a Verilog code generator available as tactics. Due to various changes in the ML-backend of Isabelle from version 2005 to version 2009-1, the original IHaVeIt code had to be ported to the more recent version of Isabelle for this work. As the work in this dissertation just relies on the NuSMV interface, only this part has been fully ported. However, the other parts are likely to be portable with minor modifications because of the modular design of IHaVeIt.

IHaVeIt consists of six main components: an Isabelle/HOL parser, a component to eliminate uninterpreted functions, a data abstraction component, pretty printers for NuSMV and SAT solvers, the Verilog code generator, and a translator for the tools' results back to Isabelle/HOL. The effort of porting the tool can be summarised in two main tasks: adapting the Isabelle/HOL parser to handle changes in the internal representation of theorems and data structures, such as the removal of a *set* datatype from Isabelle's internal type system, and fixing errors in the code caused by changes in the ML library of Isabelle's backend, such as parametrising the set membership with an equality function. To adapt IHaVeIt to Isabelle 2009-1, the former task has been completed, and the latter has been accomplished for the parser, the elimination of uninterpreted functions, the data abstraction, the NuSMV pretty printer, and the NuSMV part of the result translator.

2.3 Executing Specifications

The models constructed using the framework are specified in the executable subset of HOL [BN02]. By restricting the models to this subset, the Isabelle code generator [Haf09] can be used to export specifications as functional programs. The code generator is a built-in, generic framework for generating functional programs from executable Isabelle/HOL specification. It is generic in the sense that the target language is not fixed but can be any functional programming language. Isabelle 2009-1's code generator supports Standard ML, OCaml [Ler], and Haskell [Jon03]. Code is generated from a set of raw code equations using three sequential steps: (i) *preprocessing*, (ii) *translation*, and (iii) *serialisation*.

The preprocessor produces a structured collection of code equations applying a chain of definitional substitution steps to the set of raw code equations using a *simpset* and a *function transformer*. The former applies rewrite rules to the right hand side and to the arguments of the left hand side of code equations. The constant heading the left hand side is only rewritten in special cases such as replacing non-executable constructs with executable definitions. The latter is a type and heading-constant preserving mechanism to transform a list of function theorems into another one. The preprocessor also deals with the implicit existence of equality in Isabelle's logic by generating explicit equality functions where necessary.

The translation step produces a program in an abstract intermediate language from the code equations output by the preprocessor. This intermediate program consists of four different statements: one to represent datatypes, one to represent the actual code equations, and two for type classes.

The serialisation process transforms the intermediate program into actual source

code in the target language. This final step only maps concrete syntax to the statements of the intermediate language. Note that the preprocessor is written in Isabelle's logic and only the translation and the serialisation are performed outside of the logic, which keeps the amount of code to trust small.

Since all the models in the framework are described using deterministic Mealy machines, a model can be simulated simply by executing the Mealy machine with some inputs assignments. A simulation outcome is represented as a list of state and output elements, thus for a Mealy machine with state space S, input alphabet I, and output alphabet O, a simulation trace sim_{trc} is a list of $(S \times O)$ elements:

$$sim_{trc} \in (S \times O)\, list \tag{2.1}$$

Mealy machines are detailed in Chapter 3, but to illustrate the simulation environment, $s^k \in S$ refers to the machine state after applying the transition function δ(next-state function) of the machine k times to an initial state s^0. Similarly, $o^k \in O$ denotes the value of the output function ω given a state s^k and a value from the input alphabet $i \in I$. Thus, given a list of input values is of length n, then:

$$sim_{trc}[k] = (s^k, o^k) \qquad \text{for } k < n \tag{2.2}$$

Since this thesis does not focus on simulation or test case generation, the list of input signals is assumed to be provided as a manually-specified argument to the simulation function, and not, for example, randomly generated. Then, the following recursive function **run** computes k steps of the simulation trace starting

from a state $s \in S$.

$$\textbf{run}\ \delta\ \omega\ s\ (i :: ins)\ 0 = (s,\ \omega\,(s, i)) \tag{2.3}$$

$$\textbf{run}\ \delta\ \omega\ s\ (i :: ins)\ k = (s,\ \omega\,(s, i)) :: \textbf{run}\ \delta\ \omega\ (\delta\,(s, i))\ ins\ (k-1) \tag{2.4}$$

where $i :: ins$ denotes a list with at least one element i and a (possibly empty) rest ins.

The following function **sim** simulates a Mealy machine M with transition function δ, output function ω, and initial state *s0* for n steps given a list of input signals ins with **length** $ins \geq n$.

$$\textbf{sim}\ M\ ins\ n = \textbf{run}\ M.\delta\ M.\omega\ M.s0\ ins\ n \tag{2.5}$$

2.4 Related Approaches and Protocol Verification

This section is dedicated to the discussion of related approaches and general protocol verification work. It is not intended to be complete, but to provide an overview and to outline the differences between existing work and the work presented here. Related approaches can broadly be categorised into the following areas:

- Verification, especially hardware verification, based on step-wise refinement
- Communication protocol formalisation and verification using state machines or I/O automata
- Incremental or correctness-preserving, step-wise modelling
- Hybrid verification approaches

Since this dissertation is structured such that many chapters discuss related work at the end, modelling and verification using state machines or I/O automata is discussed in Chapter 3 together with the formalisation of communicating state machines. Existing work on bus protocol verification tackling a specific protocol is discussed in Chapter 5 with the AMBA case study, and work related to the PCI and PCI Express protocol is surveyed in the PCI express chapter (Chapter 6), respectively. This leaves step-wise refinement, generic frameworks for verified protocols, incremental modelling, and hybrid verification approaches to be discussed here.

Verification by step-wise refinement. Verification by step-wise refinement is, in general, of course not new and there is a rich literature going at least back to Dijkstra [Dij68, Dij76] and Wirth [Wir71]. Also the application of refinement checking to hardware verification has a long history. Abadi and Lamport [AL91] show the existence of refinement mappings in their widely-cited article. McMillan [McM97] proposes a compositional rule for hardware verification based on local refinements which can be efficiently model-checked. Aagaard *et al.* [ACDJ01] present a framework for microprocessor correctness statements based on simulation relations. Chen *et al.* [CGG07] propose a modular, refinement-based approach to verify transaction-based hardware implementations against their specification models. They use a cache coherency protocol to illustrate their methodology. Although, the basic idea seems similar, the focus of this work is different in various aspects: their contributions aim at verifying implementations against specifications, at generating VHDL code, and at transistor-level representations.

Frameworks for verified protocols. In recent work, Abu Kharmeh *et al.* [KEM11] present a design-for-verification framework for a configurable performance-critical communication interface. The work is formalised in Hoare's Communicating Sequential Processes algebra and the FDR model-checker is used to verify certain aspects of the model. To handle protocol complexity, the authors decided to decompose the communication controller into blocks of different functions. Even though closely related to this thesis, the work aims at a slightly different application area—configurable interfaces, which is crucial for their approach—and the authors apply a very different approach: process algebra with model-checking instead of theorem proving with integrated automatic tools, and separating functional components of the controller instead of composing a verified model incrementally.

Schmaltz *et al.* [SB06] present initial work on a generic network on chip model as a framework for correct on-chip communication. They identify key constraints on architectures and show protocol correctness provided these are satisfied. However, the approach relies on a post-hoc verification of the key constraints and the work is tailored towards routing networks rather than verified communication controllers implementing a specific protocol or feature set. So this work is different in aim and approach from this thesis as it does not attempt to tackle a feature-rich communication controller and it relies on a post-hoc verification approach.

Müffke [Müf04] presents a framework for the design of communication protocols. He provides a dataflow-based language for protocol specification, and decomposition rules for interface generation relating dataflow algebra and process algebra. Aside from noting that correct and verified protocol design is still an unsolved problem, Müffke does not address the verification aspect in

general. Claiming that the generated interfaces are correct by construction in terms of their specification, he neither addresses the protocol correctness itself nor the verification of the implementation against the specification.

Finally, there is a rich literature on protocol description languages with corresponding property verifiers, such as Murφ from Dill *et al.* [DDHY92]. This class of related work usually uses an event-based description of a protocol instead of an actual architecture model, and relies on a post-hoc verification using model checking, both of which are key differences.

Incremental or correctness-preserving modelling. The basic idea of the presented approach is similar to Intel's integrated design and verification (IDV) system [Seg06], a system for verified (gate-level) hardware designs. The system can replace parts of a hardware design with a new design, i. e. with a different circuit, while ensuring that the new circuit provides the same input-output properties as the old one. The IDV system justifies its transformations by a *local proof* using simple equivalence checking.

Suhaib *et al.* [SMSB05] propose an incremental methodology for developing formal models called XFM. Their intention and basic idea is very similar to ours, but they use a different approach to the problem. An extendable set of LTL properties is used to incrementally create a model that satisfies the set of properties. Their approach focuses on building prescriptive formal models that capture the behaviour of natural language specifications. Thus, thee approach is mainly tailored to control-dominated application areas, whereas the framework in this dissertation is tailored to a more specific application area, but providing a greater flexibility in that area, for example also covering datapath-dominated

specifications. Moreover, our methodology tries to capture specific features in independent models.

Finn and Fourman [FF93] present the tool set LAMBDA, a refinement based general-purpose design assistant using mathematical logic to represent and manipulate system behaviour. Their system also applies transformations to modify a current design. In that aspect their work is closely related to ours, but the tool, being a formal synthesis tool, aims at transforming higher-order logic specifications into HDL code. Our work is different in terms of application area, and especially in focusing on specifying complex protocol features independently and even in a re-usable way.

Another related approach is the B Method [Abr96], an event-based method for a refinement-based specification, design, and implementation of software components. Abrial *et al.* [ACM03] apply the method to the incremental development of the IEEE 1394 tree identify protocol. Although this work uses an incremental development approach, the event-based approach and the choice of protocol distinguishes this work from ours. We contribute a methodology, and not "just another" specific protocol verification.

Hybrid verification. The combination of Isabelle/HOL and NuSMV using the IHaVeIt tool has been applied to a variety of hardware verification instances. Schmaltz [Sch07] applies it to the area of clock domain crossing and the time-triggered hardware implementing it. Alkassar *et al.* [ABK08] use the tool to show the correctness of a fault-tolerant real-time scheduler and its hardware implementation. In both cases, the authors apply a similar strategy: they use theorem proving to argue about real-time, asynchronous properties of the system,

and the model checker to prove properties of finite state machines which are used to model the hardware implementation.

As research on hybrid verification approaches is not the core of my thesis, this short discussion is only meant to point out some work which uses the same tool chain. It is not meant to be an exhaustive overview nor a broad discussion of hybrid verification. However, a broader and more general overview of hybrid verification approaches can be found in the survey from Bhadra *et al.* [BAWR07].

Chapter 3

Communicating State Machines

This chapter is dedicated to introducing the fundamentals required for the specification of the framework components. Section 3.1 introduces some non-standard mathematical notations. As the framework is based on communicating Mealy machines, Section 3.2 details the representation and formalisation of Mealy machines in the scope of this thesis, which is extended with a model of communication in Section 3.3. Finally, Section 3.4 discusses previous work on modelling using state machines and formalising Mealy machines in theorem provers.

3.1 Notation and Basics

The main tool to specify and formalise the framework is discrete mathematics and logic. The reader is assumed to be familiar with basic Boolean algebra—briefly summarised in Table 3.1—and quantified propositional logic using universal ($\forall$) and existential ($\exists$) quantification over variables. As the framework reverts

Symbol	Description
T	the constant *true*
F	the constant *false*
$\neg x$	unary *not* operator
$x \wedge y$	binary *and* operator
$x \vee y$	binary *or* operator
$x \implies y$	implication
$x \equiv y$	equivalence

Table 3.1: Basic Boolean Operators

to higher order logic to define operations in an abstract and generic way, quantifications can also range over functions. If not further specified, numbers are assumed to be naturals including the number zero, referred to as $\mathbb{N}$. The following notation is used to specify intervals of naturals $a, b \in \mathbb{N}$:

$$[a, b] = \{n \in \mathbb{N} \mid a \leq n \wedge n \leq b\}$$

$$(a, b) = \{n \in \mathbb{N} \mid a < n \wedge n < b\}$$

$$[a, b) = \{n \in \mathbb{N} \mid a \leq n \wedge n < b\}$$

$$(a, b] = \{n \in \mathbb{N} \mid a < n \wedge n \leq b\}$$

Option Type

The *option datatype* is a well-known concept from function programming languages, often also called the *maybe value*. Its main purpose is to specify a possibly undefined value (**undef**) in a mathematically sound way: assume x is either undefined or an element of some set $\mathcal{S}$. For the former, the special symbol **None** is employed. For the later, x is wrapped with **Some** to indicate that it is defined. So the domain $\mathcal{S}\,option$ is the set that contains the special symbol **None** and

Some x for all elements $x \in \mathcal{S}$.

Definition 3.1 (Option Type)

Given a set $\mathcal{S}$*, the corresponding option set* $(\mathcal{S})option$ *is defined as:*

$$(\mathcal{S})option = \{\textbf{Some } x \mid x \in \mathcal{S}\} \cup \{\textbf{None}\} \tag{3.1}$$

The semantics of $\hat{x} \in (\mathcal{S})option$ *is given by:*

$$\hat{x} = \begin{cases} \textbf{Some } x & : x \in \mathcal{S} \\ \textbf{None} & : x = \textbf{undef} \end{cases} \tag{3.2}$$

As a (partial) inverse, the selection operator **the** maps elements from $(\mathcal{S})option$ to $\mathcal{S}$. It is undefined if it is applied to **None**, and returns the element x if it is applied to **Some** x.

Definition 3.2 (Selection Operator)

The partial function $\textbf{the} : (\mathcal{S})option \to \mathcal{S}$ *is defined as:*

$$\textbf{the}(\hat{x}) = \begin{cases} x & : \hat{x} = \textbf{Some } x \\ \textbf{undef} & : \hat{x} = \textbf{None} \end{cases} \tag{3.3}$$

Labelled Tuples

Labelled tuples simplify the use of Cartesian product domains by assigning names to tuple components, similar to a vector space where an element $\vec{x} \in \mathcal{D}^n$ is usually referred to as $(x_1, \ldots, x_n)$ and the i-th component is labelled x_i. Given n sets $\mathcal{S}_1, \ldots, \mathcal{S}_n$, the standard Cartesian product $\mathcal{S}_1 \times \ldots \times \mathcal{S}_n$ is given by

$\{(s_1, \ldots, s_n) \mid s_i \in S_i \text{ for } i \in [1, n]\}$. The corresponding set of labelled tuples is given in Definition 3.3.

Definition 3.3 (Set of Labelled Tuples)

Given n sets $\mathcal{S}_i$ *for* $i \in [1, n]$*, a set of n* labels $\mathcal{L} = \{l_i \mid i \in [1, n]\}$*, and a* labelling function $\mathbf{l} : \{\mathcal{S}_i \mid i \in [1, n]\} \to \mathcal{L}$*, the* set of labelled tuples $\mathcal{S}_\mathbf{l}$ *is given by:*

$$\mathcal{S}_\mathbf{l} = [\mathbf{l}(\mathcal{S}_1)]\mathcal{S}_1 \times \ldots \times [\mathbf{l}(\mathcal{S}_i)]\mathcal{S}_i \times \ldots \times [\mathbf{l}(\mathcal{S}_n)]\mathcal{S}_n \tag{3.4}$$

which yields the following components:

- *The set* $\mathcal{S}$ *containing all standard tuples.*

$$\mathcal{S} = \{(s_1, \ldots, s_n) \mid s_i \in S_i \text{ for } i \in [1, n]\} \tag{3.5}$$

- *Labelled accessor functions* $l_i : \mathcal{S} \to S_i$ *for* $i \in [1, n]$ *such that*

$$l_i((s_1, \ldots, s_n)) = s_i \tag{3.6}$$

To specify a concrete instance of a set of labelled tuples, the following Isabelle-inspired, simplifying notation is used:

$$\mathcal{S} = (\!|\, l_1 : \mathcal{S}_1, \ldots,\ l_i : \mathcal{S}_i, \ldots,\ l_n : \mathcal{S}_n \,|\!) \tag{3.7}$$

Note that the labelling function $\mathbf{l}$ is implicitly given. Additionally, $s.l_i$ is used as a shorthand for $l_i(s)$; and for an element in $\mathcal{S}$, the following notation is used to

make labels explicit:

$$s = (\!| \, l_1 = s_1, \ldots, l_s = s_i, \ldots, l_n = s_n \, |\!) \tag{3.8}$$

To use labelled tuples similarly to normal tuples, some of the standard operations for Cartesian products are lifted, and extended with some labelled tuple-specific operators. For a label l_i, the element operation $l_i \tilde{\in} \mathcal{S}_\mathbf{l}$ checks if a label is used in a labelled tuple. It is defined as:

$$l_i \; \tilde{\in} \; \mathcal{S}_\mathbf{l} \equiv l_i \in \mathcal{L} \tag{3.9}$$

The set $\mathbf{dom}[\mathcal{S}_\mathbf{l}](l_i)$ denotes the domain set of the tuple component with label l_i. The function $\mathbf{dom}[\mathcal{S}_\mathbf{l}] : \mathcal{L} \to \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ is given by:

$$\mathbf{dom}[[l_1]S_1 \times \ldots \times [l_n]S_n](l_i) = S_i \tag{3.10}$$

Given a concrete labelled tuple $s = (s_1, \ldots, s_n) \in [l_1]S_1 \times \ldots \times [l_n]S_n$, the update of component l_i by a value $s'_i \in \mathcal{S}_\rangle$ is denoted $s(\!| \, l_i := s'_i \, |\!)$ and defined as:

$$s(\!| \, l_i := s'_i \, |\!) = (l_1 = s_1, \ldots, l_{s-1} = s_{i-1}, l_s = s'_i, l_{s-1} = s_{i+1}, \ldots, l_n = s_n) \tag{3.11}$$

To update *distinct* components l_i, l_j by values v_i, v_j, the shorthand $s(\!| \, l_i := v_i, \; l_j := v_j \, |\!)$ denotes $s(\!| \, l_i := v_i \, |\!)(\!| \, l_j := v_j \, |\!)$ (where the order is irrelevant as $l_i \neq l_j$). Analogously to the usual Cartesian product for sets, that is given by

$$\prod\nolimits_{i \in [1,n]} \mathcal{S}_i = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n,$$

the product operator for sets of labelled tuples is defined as:

$$\widetilde{\prod}_{i\in[1,n],\mathbf{l}}\mathcal{S}_i = [l_1]\mathcal{S}_1 \times \ldots \times [l_n]\mathcal{S}_n \quad \text{where } l_i = \mathbf{l}(\mathcal{S}_i) \tag{3.12}$$

Finally, we define a *disjoint label union* and a *concatenation* operator for sets of labelled tuples. The former is given by the set that contains all the labels of the sets of labelled tuples:

$$\widetilde{\biguplus}_{i\in[1,n]}\mathcal{S}_i = \{l_{i,j} \mid l_j \tilde{\in} \mathcal{S}_i\} \tag{3.13}$$

For sets of labelled tuples $\mathcal{S}_i = (\!|\, l_{i,0} : \mathcal{S}_{i,0}, \ldots, l_{i,m_i} : \mathcal{S}_{i,m_i} \,|\!)$, the concatenation is given by:

$$\widetilde{\bigodot}_{i\in[0,n]}\mathcal{S}_i = (\!|\, l_{0,0} : \mathcal{S}_{0,0}, l_{0,1} : \mathcal{S}_{0,1}, \ldots, l_{n,m_n} : \mathcal{S}_{n,m_n} \,|\!) \tag{3.14}$$

Lastly, $\mathcal{S}_0 \tilde{\times}_{\mathcal{L}} \mathcal{S}_1$, $\mathcal{S}_0 \tilde{\uplus} \mathcal{S}_1$, and $\mathcal{S}_0 \tilde{\odot} \mathcal{S}_1$ are used for the binary variants of Cartesian product, disjoint union, and concatenation.

Signals

Intuitively, a *signal* is a variable of a domain $\mathcal{D}$ which is assigned a value at a given time. Since this dissertation deals with discrete, synchronous systems, time is modelled using naturals. Thus, a signal is a function from time to the domain space of the signal.

Definition 3.4 (Signal)

A signal *sig is a function from discrete time to a signal domain $\mathcal{D}$: $sig : \mathbb{N} \to \mathcal{D}$. The value of the signal at time $t \in \mathbb{N}$ is denoted $sig^t \in \mathcal{D} = sig(t)$.*

3.2 Mealy Machines

Components in the framework are modelled using standard *Mealy machines* [Mea55], that is a state machine where the output depends on the current state and input. It is specified by a 6-tuple: three sets define the *state space*, the *input alphabet*, and the *output alphabet*. The other three components are an *initial state*, a *transition function*, and an *output function*.

Definition 3.5 (Mealy Machine)

A Mealy machine is given by a 6-tuple $(S, I, O, s0, \delta, \omega)$ *where*

- S *is the set of possible states.*
- I *is the input alphabet, that is the set of possible input values.*
- O *is the output alphabet, that is the set of possible output values.*
- $s0 \in S$ *is the initial state.*
- $\delta : S \times I \to S$ *is the transition function that defines the next state for a given state and input.*
- $\omega : S \times I \to O$ *is the output function that defines the output for a current state and input.*

Given a Mealy machine, the next state s' *for a state* $s \in S$ *and an input assignment* $i \in I$ *is given by*

$$s' = \delta(s, i) \tag{3.15}$$

and the current output is given by $\omega(s, i) \in O$.

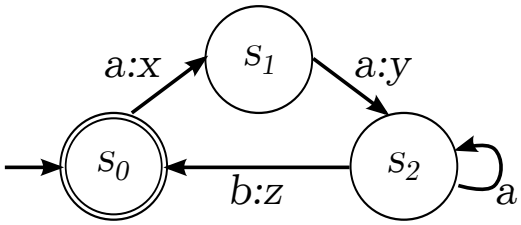


Figure 3.1: A Simple Mealy Machine

Figure 3.1 depicts a simple example of a Mealy machine in a graph-based representation. Using Definition 3.5, this state machine is defined by:

$$S = \{s_0, s_1, s_2\}$$

$$I = \{a, b\}$$

$$O = \{x, y, z\}$$

$$s0 = s_0$$

$$\delta = \{((s_0, a), s_1), ((s_1, a), s_2), ((s_2, a), s_2), ((s_2, b), s_0)\}$$

$$\omega = \{((s_0, a), x), ((s_1, a), y), ((s_2, b), z)\}$$

In the following, S, I, and O are usually given as sets of labelled tuples to simplify referring to a specific input or output using a label or name.

To model and argue about the behaviour of a state machine over time, the notions of an *execution trace* and an *output signal* are introduced. The former is a sequence of configurations such that the first element of the sequence is the initial state, and an element at position $i + 1$ ($i \in \mathbb{N}$) is equal to the transition function applied to element i and the input values at time i. The output signal is the signal representing the value of the output function at any time $t \in \mathbb{N}$. To define both concepts formally, the signal representing the input assignments at

time $t \in \mathbb{N}$ of a state machine M is required, called *input signal*. Usually, it is denoted $in_M^t \in I$, r simply in^t if smM is clear from the context.

Definition 3.6 (Execution Trace and Output Signal)
Given a Mealy state machine $M = (S, I, O, s_0, \delta, \omega)$ and an input signal in^t, the execution trace of machine M with input in, $\tau_{M,in} : \mathbb{N} \to S$, is given by:

$$\tau_{M,in}^t = \begin{cases} s_0 & : t = 0 \\ \delta(\tau_{M,in}^{t-1}, in^{t-1}) & : t > 0 \end{cases} \tag{3.16}$$

Similarly, the output signal, $out_{M,in} : \mathbb{N} \to O$, is given by $out_{M,in}^t = \omega(\tau_{M,in}^t, in^t)$. If the context is clear, the indices are omitted: τ^t and out^t.

3.3 Model of Communication

Next, a model of communication among state machines is introduced which will be used to define composition operators later in this chapter. Uni-directional communication from a *sender* (S) to a *receiver* (R) is modelled by connecting an output of the source to an input of the destination. This 'connection' is modelled by defining the value of the input using the output function of the source, instead of modelling the input as an environment input. To illustrate the general approach, assume a communication from output $x \in O_s$ to input $y \in I_r$. Given the sender's inputs $in_s^t \in I_s$, the receiver's input signal y is then given by:

$$in_r^t.y = (\omega_s(\tau_{s,in_s}^t, in_s^t)).x \tag{3.17}$$

$$= out_{s,in_s}^t.x \tag{3.18}$$

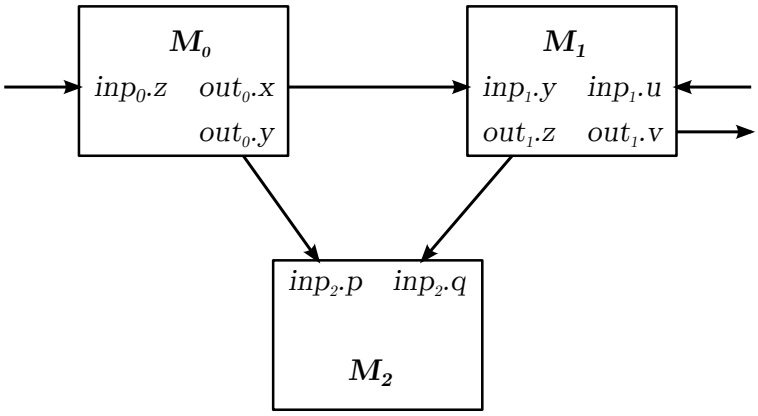


Figure 3.2: A System of Communicating State Machines

Such an input is called *internal input*; the output is called *internal output*.

Figure 3.2 depicts an example of three communicating state machines M_0, M_1, and M_2. The communication system defines a new state machine where the inputs (outputs) of the system are the union of all inputs (outputs) minus the internal signals. In order to fully specify the *global* communication system in terms of the three individual machines, a closed-form definition for every internal output has to be specified. In this framework, this is achieved by requiring a set of assumptions that prevents loops in the construction. The next chapter discusses the issue in detail when composition operators are introduced (Section 4.2). The communication in Figure 3.2 is given by the following set of assignments:

$$in_1^t.y = out_0^t.x$$

$$in_2^t.p = out_0^t.y$$

$$in_2^t.q = out_1^t.z$$

Such a set of assignments defines a global, partial communication function, $\mathbf{com}_{\mathcal{M}}$,

which specifies all internal signals among a set of state machines $\mathcal{M}$:

$$\mathbf{com}_{\mathcal{M}} : \widetilde{\biguplus}_{M_i \in \mathcal{M}} I_i \to \widetilde{\biguplus}_{M_i \in \mathcal{M}} O_i.$$

It maps any state machine input to an output iff it is an internal input. In this example, $\mathbf{com}_{\mathcal{M}} = \{(in_1^t.y, out_0^t.x), (in_2^t.p, out_0^t.y), (in_2^t.q, out_1^t.z)\}$. The full communication system is given by the set of participating state machines $\mathcal{M}$ and the communication function $\mathbf{com}_{\mathcal{M}}$. This is generalised in Definition 3.7.

Definition 3.7 (Global Communication Function)

Given a set of state machines $\mathcal{M} = \{M_0, \ldots, M_n\}$*, communication is specified as a partial function* $\mathbf{com}_{\mathcal{M}} : \widetilde{\biguplus}_{i \in [0,n]} I_i \to \widetilde{\biguplus}_{i \in [0,n]} O_i$ *such that*

$$\mathbf{com}_{\mathcal{M}}(y_i) = \begin{cases} x_j & : O_j.x \text{ of } M_j \text{ is connected to } I_i.y \text{ of } M_i \\ \mathbf{undef} & : otherwise \end{cases} \tag{3.19}$$

An input y *of* M_i external *is called* external *with respect to* $\mathcal{M}$ *iff* $\mathbf{com}_{\mathcal{M}}(y_i) =$ **undef** *and* internal *otherwise.*

In order to specify composition operators using this model and in order to define the global state machine given by the communication system, a standard interface between state machines is introduced. A simple *handshaking* protocol with three signals is used to implement uni-directional communication between two state machines. The sender provides a *valid* and a *data* signal, and the receiver provides a *busy* signal. Intuitively, to initiate communication, the sender provides data on the data signal and raises the valid signal to indicate valid data. If the receiver cannot receive, the busy signal is active and the sender has to wait. If the busy

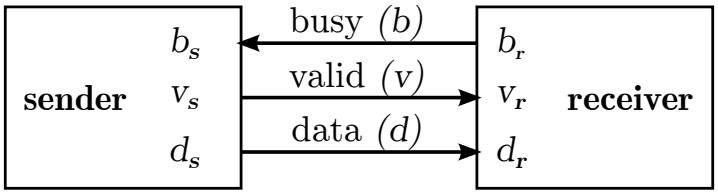


Figure 3.3: Signals of the Handshake Protocol

signal is not active, the receiver commits to sampling the data right away, that is in the same cycle.

Note that this simple protocol is closely related a handshaking protocol which is well-known and widely used: the valid/ready handshake. The only difference is that the receiver uses a positive acknowledgement signal instead of a negative one. The valid/ready handshake is, for example, used in ARM's more recent AXI protocols [ARM08].

We refer to the three signals with $b^t \in \mathbb{B}$, $v^t \in \mathbb{B}$, and $d^t \in \mathcal{D}$ where $\mathcal{D}$ is the set of data elements to be communicated. The interface is named according to the data flow direction: it is called *input interface* of machine M if the valid and data signals are inputs to M, and *output interface* of M if these two signals are outputs of M. Figure 3.3 illustrates the interconnection of two state machines.

The handshake protocol is formalised in terms of a *valid input interface* property, an assumption that specifies the behaviour of input signals to a state machine as valid and according to the protocol semantics.

Assumption 3.8 (valid input interface)

Let $b^t \in \mathbb{B}$, $v^t \in \mathbb{B}$, and $d^t \in \mathcal{D}$ be an input interface of machine M. The interface signals is called valid *iff they satisfy the following property for all times $t \in \mathbb{N}$:*

Let $x \in \mathcal{D}$ be an arbitrary but fixed data element, then

$$v^t \implies (d^t = x) \wedge \left(b^t \implies v^{t+1} \wedge (d^{t+1} = d^t)\right). \tag{3.20}$$

Note that v^t and d^t are (environment) inputs to machine M, whereas b^t is an output of M.

To specify the models in a compact and concise way, the standard interface abstracts from the valid signal, and the data signal is modelled using the option data type. The abstracted interface consists of two signals: a busy signal $b^t \in \mathbb{B}$ and a data signal $\tilde{d}^t \in \mathcal{D}$ *option*. The pair $(b^t, \tilde{d}^t)$ is called *abstract standard interface*. In the following, the $\tilde{d}^t$ is simplified to d^t. The abstraction mapping for the valid signal is given by:

$$v^t \equiv (\tilde{d}^t \neq \textbf{None}) \tag{3.21}$$

Next, this abstract standard interface in generalised to support multiple input and outputs. The idea is that a state machine with n inputs and m outputs has the same number of interfaces. Thus, the set of labelled tuples representing the state machine's input has to consist of n data signals and m busy signals. Analogously, the output has to consist of m data signals and n busy signals. Such a pair of input and output signals is called *abstract n-m standard interface*; it is defined formally in Definition 3.9 and illustrated in Figure 3.4.

Definition 3.9 (Abstract *n-m* Standard Interface)

Given a labelling function $\mathbf{l}_k$ with $\mathbf{l}_k(i) = l_i$ for $i \in [1, k]$, the input and output tuples of a state machine with n input interfaces and m output interfaces,

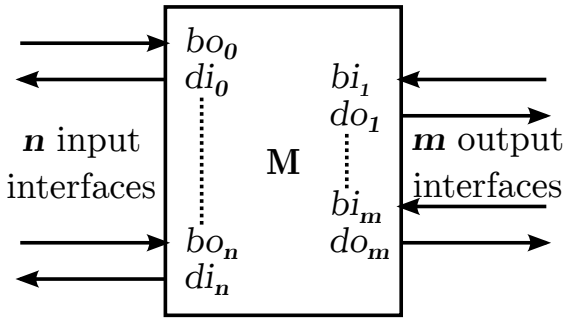


Figure 3.4: Abstract n-m Interface

complying to the standard interface, are given by:

$$I = \left(\widetilde{\prod}_{m,\mathbf{bi}_m} \mathbb{B}\right) \tilde{\odot} \left(\widetilde{\prod}_{n,\mathbf{di}_n} \mathcal{D}_i \ option\right) = \mathcal{BI}^m \tilde{\odot} \mathcal{DI}^n \tag{3.22}$$

$$O = \left(\widetilde{\prod}_{n,\mathbf{bo}_n} \mathbb{B}\right) \tilde{\odot} \left(\widetilde{\prod}_{m,\mathbf{do}_m} \mathcal{D}_j \ option\right) = \mathcal{BO}^n \tilde{\odot} \mathcal{DO}^m \tag{3.23}$$

The pair (I, O) is denoted (n, m) StdInterface.

To simplify notation, the following labelling convention is used: the k-th data input of a standard input interface is denoted di_k together with the k-th busy output bo_k, and n is the number of single input interfaces. Analogously, the k-th data output of a standard output interface is labelled do_k together with the k-th busy input bi_k, and m is the number of single output interfaces. If the abstract interface has only on standard interface, the subscript 0 is omitted. Moreover, as in Definition 3.9, $\mathcal{BI}^m(I)$ is used to refer to the m busy inputs and $\mathcal{DI}^n(I)$ to refer to the n data inputs of I. Similarly, $\mathcal{BO}^n(O)$ and $\mathcal{DO}^m(O)$ are used to refer to the output signals.

Using the model of communication from Definition 3.7, there is a possibility of creating combinational loops when combining Mealy machines. To prevent this, two symmetric interface properties are defined: one for an output interface

(Definition 3.10) and a symmetric one for an input interface (Definition 3.11). Intuitively, the latter states that the internal output of an interface behaves like a *Moore output*, that is the output signal is independent from the input signal and only depends on the current machine configuration, and the former says that the internal input signal of an interface is independent from the internal output (*Moore input*).

Definition 3.10 (Moore-like Output Interface)

Given a state machine M, an output interface $(bi_k \tilde{\in} I, do_k \tilde{\in} O)$ is called Moore-like*, or* busy-independent*, output iff the output function ω satisfies:*

$$\forall in,\, t.\ \omega\ (\tau^t_{M,i}, in^t).do_k = \omega\ (\tau^t_{M,in}, in^t(\!|\, bi_k := \mathsf{T}\,|\!)).do_k \tag{3.24}$$

In the following, this property is referred to as $Moore_{Out}(bi_k, do_k)$. M provides Moore-like, or busy-independent, outputs iff ω satisfies $Moore_{Out}(bi_k, do_k)$ for all $k \in [1, m]$.

Definition 3.11 (Moore-like Input Interface)

Given a state machine M, an input interface $(bo_k \tilde{\in} O, di_k \tilde{\in} I)$ is called Moore-like*, or* data-independent*, input iff the output function ω satisfies:*

$$\forall in,\, t.\ \omega\ (\tau^t_{M,in}, in^t).bo_k = \omega\ (\tau^t_{M,in}, in^t(\!|\, di_k := \mathbf{None}\,|\!)).bo_k \tag{3.25}$$

In the following, this property is referred to as $Moore_{Inp}(b0_k, di_k)$. M provides Moore-like, or busy-independent, inputs iff ω satisfies $Moore_{Inp}(bo_k, di_k)$ for all $k \in [1, n]$.

These two properties are used when two interfaces—an output and an input

interface—are being connected: composition operators require that either the output interface is Moore-like, or the input interface is. If this is satisfied, no loop will be created.

Note, that this is not necessarily the weakest possible assumption that prevents the creation of loops. In general, it is sufficient if at least one machine in any (potential) cycle provides Moore-like signals (with the signals that are part of the cycle). But the stronger assumption also ensures that the absence of a loop is a property that can be shown *locally* and does not introduce a global property which has to be checked. Reducing global properties to local ones is one of the major aspects of this framework.

Assumption 3.12 (Valid Global Communication Function)
A global communication function $\mathbf{com}_{\mathcal{M}}$ *for a set of state machines* M *is called* valid *iff for every internal interface intf, either* $Moore_{Inp}(intf)$ *or* $Moore_{Out}(intf)$ *holds.*

In order to argue about correctness in a reasonable way, an environment assumption has to be introduced: we assume in the following that the busy signal provided by the environment is fair in the sense that it is not constantly active. Thus, the environment allows progress in general. Assumption 3.13 formulates this by stating that a busy signal is at most constantly active for a finite period of time.

Assumption 3.13 (Fair *busy* Signals)
All external busy signals b *satisfy:* $b^t \implies \exists k. \forall k' < k. b^{t+k'} \wedge \neg b^{t+k}$

Note that this is a common assumption for inputs with a semantics similar to the one of the busy signal.

In the next chapter, the basic building blocks of the framework are introduced: abstract components and composition operators. When composition operators are applied to abstract components, Assumption 3.12 has to be satisfied to ensure that no combinatorial loops are created.

3.4 Related Work

Modelling systems using automata in general is a well studied field with a long history. A good introduction and overview can be found, for example, in the widely-cited book by Robert Kurchan [Kur94]. Modelling state machines in Isabelle/HOL goes back to at least Nipkow and Slind [NS95]. They formalized I/O automata and developed a meta-theory to represent them as objects in the logic. Our approach to state machines is similar to their formalization, but we restrict our state machine framework to a simpler formalization specialised to our requirements. Our model of execution is analogous to their infinite sequences. Also formal verification of protocols using I/O automata in theorem provers has a long history, e.g. [HSV94, LSGL95]. The aim of this thesis is not to provide yet another specific protocol verification using I/O automata and a theorem prover, but the formalization of a methodology. This work, including the Mealy machines and the handshaking protocol, has, nonetheless, been formalised in Isabelle/HOL.

Chapter 4

The Framework

Using the communicating Mealy machines from the previous chapter, this chapter details the generic framework. Section 4.1 defines the smallest, atomic building blocks: *abstract components*. *Composition operators*, detailed in Section 4.2, are used to combine basic building blocks or components in the framework to more complex blocks, thus building complex models incrementally. The chapter concludes with an introduction to *transformations* in Section 4.3. A transformation is a specification of a particular protocol feature that is modelled independently from other protocol features and encapsulates the complexity of the feature.

Note that all framework components as well as the lemmas and proofs have been formalised in Isabelle/HOL.

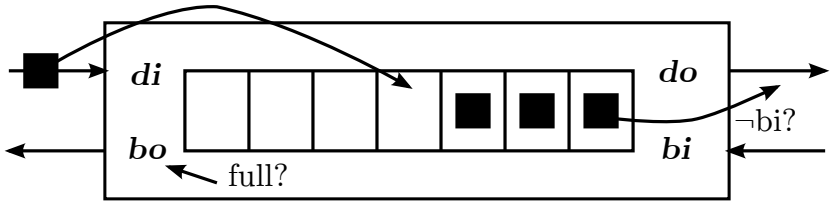


Figure 4.1: Simple Buffer of Fixed Size

4.1 Abstract Components

Abstract components are *minimal* Mealy machines implementing a specific purpose while obeying the standard interface. The framework only needs two abstract components to provide enough flexibility:

- A *bounded-size buffer*
- A *data modification* component

4.1.1 Unit-delay and Zero-delay Buffers

The buffer is the basic component of most models. There are two variants of it: a *unit-delay* and a *zero-delay* buffer. The difference is in the behaviour of an empty buffer: an empty zero-delay buffer outputs a valid input in the same time step, whereas an empty unit-delay buffer delays a valid input by at least one time step. The basic principle of a buffer with an abstract interface is depicted in Figure 4.1.

The storage part is modelled using a list. The datatype of a list containing data elements of type α is $\alpha\,\mathsf{list}$ with the constructors:

$$\mathsf{list} = \mathsf{nil} \mid x :: \mathsf{list} \tag{4.1}$$

To specify a buffer explicitly, some standard list operations are needed: the *head*

of a list, the *tail* of a list, the *concatenation* of two lists, and the *length* of a list:

$$\mathbf{hd}\ xs = \begin{cases} \mathbf{undef} & : xs = \mathsf{nil} \\ x & : xs = x :: xs' \end{cases} \tag{4.2}$$

$$\mathbf{tl}\ xs = \begin{cases} \mathbf{undef} & : xs = \mathsf{nil} \\ xs' & : xs = x :: xs' \end{cases} \tag{4.3}$$

$$xs@ys = \begin{cases} ys & : xs = \mathsf{nil} \\ x :: (xs'@ys) & : xs = x :: xs' \end{cases} \tag{4.4}$$

$$\mathbf{length}\ xs = \begin{cases} 0 & : xs = \mathsf{nil} \\ 1 + (\mathbf{length}\ xs') & : xs = x :: xs' \end{cases} \tag{4.5}$$

The state space of a bounded buffer has a component $buf \in \alpha\,\mathsf{list}$ to model the elements currently stored in the buffer, and a component $size \in \mathbb{N}$ to specify the size of the buffer. Note that for a zero-delay buffer any size greater or equal zero is supported, whereas the unit-delay buffer supports only sizes greater than zero: a unit-delay buffer of size zero is not 'logically consistent'. A zero-delay buffer of size zero behaves like a simple wire. A buffer implements a simple 1-1 standard interface, thus both input and output interface consist of a single busy and a single data component. Moreover, a buffer does not modify any data: the data domain of the input and output interface is α.

$$S_{buf} = (\!|\, buf : \alpha\,\mathsf{list},\ size : \mathbb{N} \,|\!) \tag{4.6}$$

$$I_{buf} = (\!|\, bi : \mathbb{B},\ di : \alpha\,option \,|\!) \tag{4.7}$$

$$O_{buf} = (\!|\, bo : \mathbb{B},\ do : \alpha\,option \,|\!) \tag{4.8}$$

To specify transition and output functions in a compact way, some auxiliary predicates and functions are used: $\mathbf{empty} : S \to \mathbb{B}$ indicates that a buffer is empty, $\mathbf{full} : S \to \mathbb{B}$ indicates a full buffer. The function $\mathbf{top} : S \to \alpha$ returns the oldest element in the buffer *without removing it from the list*, and $\mathbf{deq} : S \to \alpha\,\mathsf{list}$ returns the buffer content without the oldest element—in both cases, only if the buffer is not empty of course. Finally, $\mathbf{enq} : S \times \alpha \to \alpha\,\mathsf{list}$ appends an element as the newest and last element to the current buffer content.

$$\mathbf{empty}\ s = (s.buf = \mathsf{nil}) \tag{4.9}$$

$$\mathbf{full}\ s = (\mathbf{length}\ (s.buf) = s.size \wedge s.size > 0) \tag{4.10}$$

$$\mathbf{top}\ s = \mathbf{hd}\ (s.buf) \tag{4.11}$$

$$\mathbf{deq}\ s = \mathbf{tl}\ (s.buf) \tag{4.12}$$

$$\mathbf{enq}\ (s, x) = s.buf@[x] \tag{4.13}$$

Next, shorthands for the different *operating modes* of a buffer are defined to simplify the following definitions: First, $\mathbf{bypass} : S \times I \to \mathbb{B}$ indicates that input data has to bypass the list to provide zero-delay output: input data is forwarded to the buffer outputs in the same cycle if the buffer is empty.

$$\mathbf{bypass}\,(s, i) = \mathbf{empty}\ s \wedge i.di \neq \mathbf{None} \tag{4.14}$$

Second, $\mathbf{sample} : S \times I \to \mathbb{B}$ indicates that the buffer has to sample input data into the storage element, which happens in two cases: a unit-delay buffer samples data if it is not full and there is some data at the input ($\mathbf{sample_u}$); a zero-delay buffer additionally samples data in case input data should be bypassed, but it

cannot be output because of an active busy input ($\mathbf{sample_z}$). Note that in case of a zero-delay buffer, these two conditions are not disjoint.

$$\mathbf{sample_u}\,(s,i) = (i.di \neq \mathbf{None}) \wedge \neg\mathbf{full}\,s \tag{4.15}$$

$$\mathbf{sample_z}\,(s,i) = \mathbf{bypass}\,(s,i) \wedge i.bi \wedge (s.size \neq 0) \tag{4.16}$$

$$\mathbf{sample}\,(s,i) = (\neg\mathbf{bypass}\,(s,i) \wedge \mathbf{sample_u}\,(s,i)) \vee \mathbf{sample_z}\,(s,i) \tag{4.17}$$

Lemma 4.1 shows that $\mathbf{sample}\,(s,i)$ implies $i.di \neq \mathbf{None}$, which is a convenient property for formalising the models: whenever $\mathbf{sample}\,(s,i)$ holds, $i.di = \mathbf{Some}\,x$, thus **the** $i.di$ is well-defined.

Lemma 4.1

A buffer only samples data if there is valid input data.

$$\mathbf{sample}\,(s,i) \implies (i.di \neq \mathbf{None})$$

Proof The proof is straightforward by unfolding definitions:

$$\begin{aligned}
\mathbf{sample}\,(s,i) &\implies \mathbf{sample_u}\,(s,i)) \vee \mathbf{sample_z}\,(s,i) \\
\mathbf{sample_u}\,(s,i) &\implies i.di = \mathbf{Some}\,x \equiv i.di \neq \mathbf{None} \\
\mathbf{sample_z}\,(s,i) &\implies \mathbf{bypass}\,(s,i) \\
&\implies i.di = \mathbf{Some}\,x \implies i.di \neq \mathbf{None}
\end{aligned}$$

The proof is easily mechanised in Isabelle as well. ∎

Third, $\mathbf{succout} : S \times I \to \mathbb{B}$ indicates the successful output of a data element: a unit-delay buffer outputs data successfully if the busy input is not active and the

buffer is not empty ($\mathbf{succout_u}$). A zero-delay buffer also outputs data successfully if the busy input is not active and data is bypassed.

$$\mathbf{succout_u}\,(s,i) = \neg i.bi \wedge \neg \mathbf{empty}\; s \tag{4.18}$$

$$\mathbf{succout_z}\,(s,i) = \neg i.bi \wedge (\neg \mathbf{empty}\; s \vee \mathbf{bypass}\,(s,i)) \tag{4.19}$$

Using those auxiliary functions, the initial state, the transition function, and the output function of a unit-delay buffer are defined as (recall that $s.size > 0$):

$$s0_{\,ubuf}\; l = (\!|\, buf = \mathsf{nil}, size = l \,|\!) \tag{4.20}$$

$$\delta_{ubuf}\,(s,i) = \begin{cases} s & : \neg \mathbf{sample_u}\,(s,i) \wedge \neg \mathbf{succout_u}\,(s,i) \\ \mathbf{enq}\,(s, \mathbf{the}\, i.di) & : \mathbf{sample_u}\,(s,i) \wedge \neg \mathbf{succout_u}\,(s,i) \\ \mathbf{deq}\; s & : \neg \mathbf{sample_u}\,(s,i) \wedge \mathbf{succout_u}\,(s,i) \\ \mathbf{enq}\,(\mathbf{deq} \\ \quad s, \mathbf{the}\, i.di) & : \mathbf{sample_u}\,(s,i) \wedge \mathbf{succout_u}\,(s,i) \end{cases} \tag{4.21}$$

$$\omega_{ubuf}\,(s,i) = \begin{cases} (\!|\, bo_0 = \mathbf{full}\; s, do = \mathbf{None} \,|\!) & : \mathbf{empty}\; s \\ (\!|\, bo_0 = \mathbf{full}\; s, do = \mathbf{Some}\; (\mathbf{top}\; s) \,|\!) & : \neg \mathbf{empty}\; s \end{cases} \tag{4.22}$$

The zero-delay buffer is almost the same, but the output function has to take a buffer of size zero into account. Definitions 4.1 and 4.2 sum up the specifications

for both buffer models.

$$\delta_{zbuf}\,(s,i) = \begin{cases} s & : \neg\mathbf{sample}\,(s,i) \wedge \neg\mathbf{succout_z}\,(s,i) \\ \mathbf{enq}\,(s, \mathbf{the}\,i.di) & : \mathbf{sample}\,(s,i) \wedge \neg\mathbf{succout_z}\,(s,i) \\ \mathbf{deq}\,s & : \neg\mathbf{sample}\,(s,i) \wedge \mathbf{succout_z}\,(s,i) \\ \mathbf{enq}\,(\mathbf{deq}\,s, \\ \quad \mathbf{the}\,i.di) & : \mathbf{sample}\,(s,i) \wedge \mathbf{succout_z}\,(s,i) \end{cases} \tag{4.23}$$

$$\omega_{zbuf}\,(s,i) \;= \begin{cases} (\!|\, bo = i.bi, do = i.di \,|\!) & : (s.size = 0) \\ (\!|\, bo = \mathbf{full}\,s, do = i.di \,|\!) & : \mathbf{empty}\,s \wedge (s.size \neq 0) \\ (\!|\, bo = \mathbf{full}\,s, \\ do = \mathbf{Some}\;(\mathbf{top}\,s) \,|\!) & : \neg\mathbf{empty}\,s \wedge (s.size \neq 0) \end{cases} \tag{4.24}$$

Definition 4.1 (Parametrised Unit-delay Buffer)

A parametrised unit-delay *buffer of fixed size* $l > 0 \in \mathbb{N}$ *implements a* 1-1 *standard interface and is given by the Mealy machine*

$$M_{ubuf} = (S_{buf}, I_{buf}, O_{buf}, (s0_{\,buf}\;l), \delta_{ubuf}, \omega_{ubuf})$$

where the components are defined as in Equations 4.6, 4.7, 4.8, 4.20, 4.21, and 4.22.

Definition 4.2 (Parametrised Zero-delay Buffer)

A parametrised buffer of fixed size $l \in \mathbb{N}$ *with a* zero-delay *input bypassing and a*

1-1 *standard interface is given by the Mealy machine*

$$M_{zbuf} = (S_{buf}, I_{buf}, O_{buf}, (s0_{buf}\, l), \delta_{zbuf}, \omega_{zbuf})$$

where the components are defined as in Equations 4.6, 4.7, 4.8, 4.20, 4.23, and 4.24.

Buffer Correctness Results

Before introducing further abstract components, we detail the generic correctness of a buffer. As mentioned before, the execution semantics from Definition 3.6 is used and, if not stated otherwise, the following convention is used. For a state machine M,

- i^t refers to the input signal assignment at time t, thus $i : \mathbb{N} \to I$ is the environment input signal,
- s^t is the state configuration at time t, thus $s^t = \tau^t_{M,i}$, and
- o^t denotes the values of the outputs at time t, thus $o^t = out^t_{M,i}$.

The correctness properties introduced here are fully generic in the sense that they can be applied any time a buffer is used to construct more complex transformations, and any time a buffer is instantiated to model a concrete case study.

Lemma 4.2 (Zero-delay Bypassing for Zero-delay Buffers)
For a bounded-size buffer M_{zbuf}, an active **bypass** *predicate at time t implies that the buffer outputs the data element at its input (in the same time step), and that the buffer either stores the data element in case of an active busy input signal, or*

successfully outputs the element in case the busy input is not active. Formally, for a current state s^t and an input signal assignment i^t, the following holds:

$$\begin{aligned}\mathbf{bypass}\,(s^t, i^t) \implies & \left(\omega_{zbuf}\,(s^t, i^t).do_0 = i^t.di_0\right) \wedge \\ & \left(i^t.bi_0 \implies \mathbf{sample}\,(s^t, i^t) \wedge \neg\mathbf{succout}\,(s^t, i^t)\right) \wedge \\ & \left(\neg i^t.bi_0 \implies \neg\mathbf{sample}\,(s^t, i^t) \wedge \mathbf{succout}\,(s^t, i^t)\right)\end{aligned}$$

Proof Unfolding the definitions from Equations 4.14–4.18 and the definition of ω_{zbuf} (Equation 4.24), the lemma can easily be proven.

$$\mathbf{bypass}\,(s, i) \overset{D4.14}{\implies} \mathbf{empty}\,s \overset{D4.24}{\implies} \omega_{zbuf}\,(s, i).do_0 = i.di_0$$

$$\mathbf{bypass}\,(s, i) \overset{D4.17}{\implies} (i.bi_0 \equiv \mathbf{sample}\,(s, i))$$

$$\mathbf{bypass}\,(s, i) \overset{D4.18}{\implies} (\neg i.bi_0 \equiv \mathbf{succout}\,(s, i))$$

Unfolding definitions in the same way, the proof is easily mechanised in Isabelle.■

The following Lemma 4.3 argues about the output signals provided by any buffer, whether it is a zero-delay or a unit-delay one. The lemma states that if a buffer tries to output data, but the busy input is active, the data output of the buffer stays stable.

The lemma might seem marginal because it only argues about a single transition, but it is a significant property: first, it states that no data gets lost if the next component is not ready to receive it. Second, even though it argues only about one cycle, it can easily be expanded using induction, and third, the lemma states that a buffer satisfies Assumption 3.8; the assumption on interface compliant

signals. Thus, Lemma 4.3 shows that a buffer's output signal can be used as a valid input signal.

Lemma 4.3 (Stable Buffer Outputs)

Given a generic buffer, whether zero-delay or unit-delay, $B = (S, I, O, s0, \delta, \omega)$ *and an input signal* $i^t \in I$*, an active busy signal stalls the output of the buffer.*

$$\forall x \in \mathbf{dom}(do, O).\ bi^t \wedge (do^t = \mathbf{Some}\ x) \implies (do^{t+1} = \mathbf{Some}\ x)$$

Proof Although not particularly tricky, the proof is tedious because a few case splits are necessary to cover all corner cases, especially for the zero-delay buffer. For the unit-delay buffer, proofing the lemma is straightforward by unfolding definitions.

From $\omega_{ubuf}(s^t, i^t)$ and the definitions of **top** and **empty** follows:

$$(do^t = \mathbf{Some}\ x) \implies (x = \mathbf{hd}(s^t.buf)) \wedge (s^t.buf \neq \mathsf{nil})$$

Moreover, from bi^t and the definition of **succout** follows that:

$$bi^t \overset{D4.18}{\implies} \neg\mathbf{succout}_u(s, i) \overset{D4.22}{\implies} (\delta_{ubuf}(s, i) = s) \vee (\exists x.\delta_{ubuf}\ (s, i) = s.buf@[x])$$

But we also know that for all $(x :: xs) \in \alpha\, list$, $hd(x :: xs) = hd(x :: xs@[x])$ holds, and therefore $hd(s^{t+1}) = x$, thus $do^{t+1} = \mathbf{Some}\ x$.

For the zero-delay buffer, cases are split on $s^t.buf = \mathsf{nil}$. For $s^t.buf \neq \mathsf{nil}$, the proof is exactly the same as for the unit-delay buffer. So, assume $s^t.buf = \mathsf{nil}$.

1. $s^t.size > 0$: thus

$$\begin{aligned} sample_z(s^t, i^t) \wedge \neg succout(s^t, i^t) &\implies \delta_{zbuf}(s^t, i^t) = \mathbf{enq}(s^t, \mathbf{the}(i^t.di)) \\ &\implies s^{t+1} = \mathsf{nil}@[x] = [x] \\ &\implies do^{t+1} = \mathbf{Some}\ x \end{aligned}$$

2. $s^t.size = 0$: from the definition of ω_{zbuf} and from $i.bi^t = \mathbf{T}$ follows:

$$\begin{aligned} do^t &= i.di^t \text{ and } bo = i.bi^t = \mathbf{T} \\ &= i.di^{t+1} && \text{by Assumption 3.8} \\ &= do^{t+1} && \text{with } s^{t+1}.size = s^t.size \text{ and } \omega_{zbuf} \end{aligned}$$

The proof is formalised in Isabelle/HOL as it is detailed here. ∎

Finally, the main buffer correctness theorem states that a buffer itself ensures liveness if the environment is fair. It also states that a buffer does not modify or corrupt data, and that the ordering of elements in the buffer is preserved.

Theorem 4.4 (Buffer Liveness)

A generic buffer $(S, I, O, s0, \delta, \omega)$, whether zero-delay or unit-delay, satisfies the following liveness property under the assumption of a fair environment (Assumption 3.13):

$$\forall x \in \mathbf{dom}(i.di, I).\ \neg bo^t \wedge (i.di^t = \mathbf{Some}\ x) \implies \exists k.\ (do^{t+k} = \mathbf{Some}\ x)$$

Proof Assuming the busy input is not constantly active, it is easy to see that the hypothesis is true, but tedious to proof. The big picture is the following: first,

show that x becomes a list element, the last one actually, in the buffer. Second, show that x never moves backwards, but if x moves, x moves towards the output (the head of the list). Additionally, x only moves if and only if the busy signal is not active. Finally, there is only a finite number of elements in the list in front of x, but infinitely many inactive busy signals. Once the proof is split up like this, the proof is straightforward and has also been formalised in Isabelle. ■

4.1.2 Data Modification

Data modification provides orthogonal features to a buffer. A buffer does not provide any means of modifying data or of routing data through various interfaces. The data modification component provides all this: data modification and more flexible interfaces. The idea is to reason about data-independent properties, such as liveness, and data-dependent properties, such as end-to-end message correctness, separately and compositionally. Thus, the goal is to separate any data modification from the storage elements which is why this building block does not provide any data storage. It implements three core functions:

- An n-m standard interface.

$$I = \mathcal{BI}^m \,\tilde{\odot}\, \mathcal{DI}^n \tag{4.25}$$

$$O = \mathcal{BO}^n \,\tilde{\odot}\, \mathcal{DO}^m \tag{4.26}$$

- A parameter f for potential data modification

$$f : S \times \mathcal{DI}^n \to \mathcal{DO}^m \tag{4.27}$$

- A parameter b for potential busy signal *strengthening*

$$b : S \times I \to \mathcal{BO}^n \tag{4.28}$$

To ensure that the busy signal is only strengthened, b has to satisfy the following property P_b:

$$\forall s \in S. \forall\, i.bi_k \,\tilde{\in}\, \mathcal{BI}^m.\ i.bi_k \implies \bigwedge \mathit{aff}\,(b\,(s, i),\ bi_k)) \tag{4.29}$$

where $\mathit{aff}\,(bo, i.bi_k) \subseteq \mathcal{BO}^n$ is the set of busy output signals this is affected by $i.bi_k$.

To allow a data modification that is not only based on the current input data, but also on some *auxiliary data*, the state space of the building block can have an optional element $opt \in Opt$. A typical use of this component is the extension of an input data element with a sequence number or a header field. To abstract from the *opt* field, the specification is also parametrised using an initial state opt^0 and a transition function δ_{opt}. The data modification is sketched in Figure 4.2 to illustrate the basic concept.

Definition 4.3 (Data Modification)
The data modification $DM\,(f, g)$ *is parametrised in the functions given in Equations 4.27 and 4.28. The input and output domains are as given in Equations 4.25 and 4.26, and the remaining components are:*

$$S = (\!|\, opt : Opt \,|\!)$$

$$s0 = (\!|\, opt = opt^0 \,|\!)$$

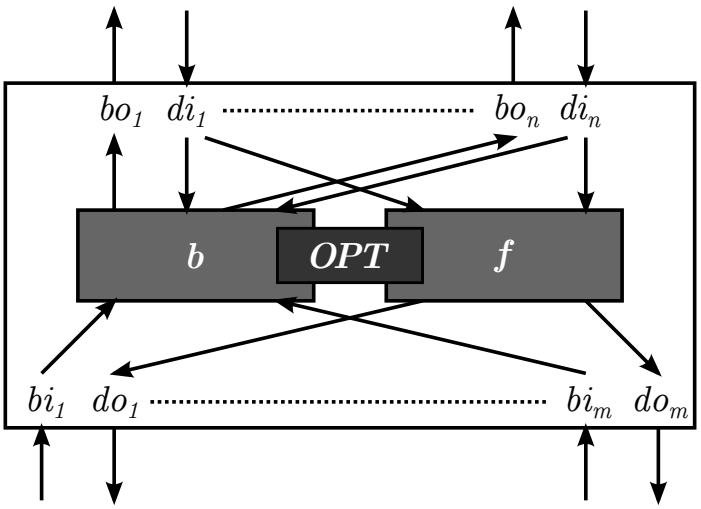


Figure 4.2: Schematics of the Data Modification

$$\delta = \lambda s.\, i.\, (\!|\, opt = \delta_{opt}\ (s, i)\,|\!)$$

$$\omega = \lambda s.\, i.\, g\,(s, i)\,\tilde{\odot}\, f\,(s, DI^n(i))$$

The data modification is a very flexible component because most of its components are to be instantiated and there are not many restrictions on how to instantiate them. On the one hand, this is favourable for composing new systems, on the other hand it makes generic verification hard. Lemma 4.5 states that the data modification provides Moore-like output interfaces.

Lemma 4.5

An instantiated data modification component provides Moore-like output interfaces.

$$DM\,(f, g) \models Moore_{out}$$

Proof Unfolding the output function ω of DM shows that $DO(o) = f(s, DI(i))$, for some i, s, which is independent from the busy signals. The proof is also

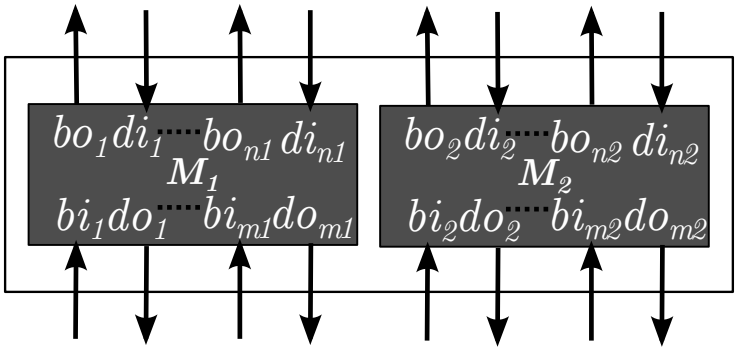


Figure 4.3: Schematics of the Parallel Composition

formalised in Isabelle by unfolding definitions. ■

4.2 Composition Operators

In order to use abstract components to build up more complex transformations, the framework specifies a set of composition operators. The first part of this section introduces two standard operations: *parallel composition* (Section 4.2.1) and *sequential composition* (Section 4.2.2). The second part specifies two special operators used to parallelize components in a controlled way: the multiplex/arbitrate composition (Section 4.2.3) and the replication operator (Section 4.2.4).

4.2.1 Parallel Composition

Parallel composition is the most simplistic composition operator: it is used to run two existing components *side-by-side*. The inputs and outputs of the composed system are a concatenation of the inputs and outputs of the *source components*, and the transition and output functions just apply the corresponding functions of the source components. This composition does not introduce any sharing

nor does it provide any means to control the execution of the sub-systems. The typical application of the parallel composition is to compose transmit and receive parts of a model such that the composed system is again a system within the framework. The basic principle is sketched in Figure 4.3.

Definition 4.4 (Parallel Composition)

The parallel composition of two components M_1 and M_2, denoted $M_1||M_2$, is given by:

$$
\begin{aligned}
S &= S_1 \tilde{\times}_{\mathcal{L}} S_2 = (\!| \, m_1 : S_1,\ m_2 : S_2 \, |\!) \\
s0 &= (\!| \, m_1 = s0_1,\ m_2 = s0_2 \, |\!) \\
I &= I_1 \tilde{\times}_{\mathcal{L}} I_2 = (\!| \, m_1 : I_1,\ m_2 : I_2 \, |\!) \\
O &= O_1 \tilde{\times}_{\mathcal{L}} O_2 = (\!| \, m_1 : O_1,\ m_2 : O_2 \, |\!) \\
\delta &= \lambda s,\ i.\, (\!| \, m_1 = \delta_1 \, (s.m_1, i.m_1),\ m_2 = \delta_2 \, (s.m_2, i.m_2) \, |\!) \\
\omega &= \lambda s,\ i.\, (\!| \, m_1 = \omega_1 \, (s.m_1, i.m_1),\ m_2 = \omega_2 \, (s.m_2, i.m_2) \, |\!)
\end{aligned}
$$

where $\mathcal{L} = \{(1, m_1), (2, m_2)\}$ is the labelling.

Given the correctness of the basic building blocks, the aim is to argue about the correctness of the composition operators. The main strategy for that is to show that the properties of the basic components are *preserved* by the compositions. Informally, the idea is that if a component satisfies a correctness property P, we aim at showing that the composed system satisfies a correctness property P' that can be derived from P only from the properties of the construction of the composition.

For the parallel composition, such a correctness property is straightforward: the composed system satisfies the conjunction of the individual correctness properties. Since parallel composition only executes the two state machines simultaneously without any control or data modification, one can easily see that this is the case.

Lemma 4.6 (Parallel Composition Correctness)
Given state machines M_1, M_2 with input signals $i_1^t \in I_1$ and $i_2^t \in I_2$, and let $i = \lambda t.\ (\!|\, m_1 = i_1^t,\ m_2 = i_2^t \,|\!)$ be the input signal to the parallel composition $M_1 \,||\, M_2$. Then, the following holds:

$$\langle M_1, i_1\rangle \models_{\mathcal{A}_1} P_1 \wedge \langle M_2, i_2\rangle \models_{\mathcal{A}_2} P_2 \implies \langle M_1 || M_2, i\rangle \models_{\mathcal{A}_1 \cup \mathcal{A}_2} P_1 \wedge P_2$$

Proof As the parallel composition does nothing else than providing a wrapper around two individual components, a simple unfolding of Definition 4.4 shows the hypothesis. The proof is also easily formalised in Isabelle. ∎

4.2.2 Sequential Composition

Sequential composition is the analogy of the sequential application of two functions $(f \circ g)(x) = f(g(x))$ with respect to the datapaths of the two components to be composed: the output interfaces of the first component are connected to the input interfaces of the second component. The principle is illustrated in Figure 4.4.

As mentioned before, composing input and output interfaces of two Mealy machines can result in unwanted loops: to guarantee an acyclic construction and to be able to define the inner signals of the composed system locally, at least one of the two participating Mealy machines has to provide Moore-like interfaces. This assumption is also captured in Definition 4.5 in order to define the internal

signals of the composition in a closed form.

Definition 4.5 (Internal Interface)

Given an output interface (bi, do) *with* $bi \,\tilde{\in}\, i_1 \in I_1$ *and* $do \,\tilde{\in}\, \omega_1\,(s_1, i_1) \in O_1$, *and an input interface* (bo, di) *with* $di \,\tilde{\in}\, i_2 \in I_2$ *and* $bo \,\tilde{\in}\, \omega_2\,(s_2, i_2) \in O_2$, *the corresponding internal interface* (b, d) *is defined as follows:*

- *If* $(bi, do) \models Moore_{out}$, *then*

$$b = (\omega_2\ (s_2, i_2(\!|\, di := d \,|\!))).bo$$
$$d = (\omega_1\ (s_1, i_1(\!|\, bi := \mathbf{T} \,|\!))).do$$

- *If* $(bo, di) \models Moore_{in}$, *then*

$$b = (\omega_2\ (s_2, i_2(\!|\, di := \mathbf{None} \,|\!))).bo$$
$$d = (\omega_1\ (s_1, i_1(\!|\, bi := b \,|\!))).do$$

A sequential composition of two Mealy machines M_1 and M_2 requires the following three assumptions to be satisfied:

- The composition is based on a total mapping from M_1's output interfaces to M_2's input interfaces, i. e. a communication function that map all the data input signals of M_2 to the data output signals of M_1 and vice versa for the corresponding busy signals is required.

- In order for the composed system to implement an *n-m* standard interface, the number of busy inputs of M_2 has to match the data inputs of M_1, and vice versa.

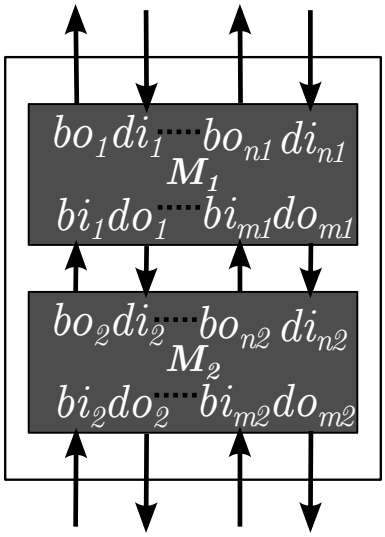


Figure 4.4: Schematics of the Sequential Composition

- For each input-output interface pair defined by the communication function, at least one of the two has to satisfy $Moore_{out}$ or $Moore_{in}$.

The first two assumptions are be satisfied if the input and output spaces have the following form that creates p internal interfaces:

$$(I_1, O_1) = (\mathcal{BI}^p \,\tilde{\odot}\, \mathcal{DI}^n,\ \mathcal{BO}^n \,\tilde{\odot}\, \mathcal{DO}^p) \tag{4.30}$$

$$(I_2, O_2) = (\mathcal{BI}^m \,\tilde{\odot}\, \mathcal{DI}^p,\ \mathcal{BO}^p \,\tilde{\odot}\, \mathcal{DO}^m) \tag{4.31}$$

The list of assumptions can be summarised in the following property:

$$SComp = \text{there exist } p \text{ acyclic internal interface pairs} \tag{4.32}$$

A pair of Mealy machines M_1, M_2 satisfying $SComp$, i. e. $M_1, M_2 \models SComp$, is called *sequentially composable*.

In the following, (i, i) with $i \in [1, p]$ is used as the mapping for the internal

interfaces to simplify notation. This means that output interface i of M_1 is connected to the input interface i of M_2. Then, the set of internal interfaces is given by the following busy and data signals:

$$\mathcal{IB} = (\!| \, b_1 : \mathbb{B}, \ldots, b_p : \mathbb{B} \, |\!) \tag{4.33}$$

$$\mathcal{ID} = (\!| \, d_1 : \alpha_0 \, option, \ldots, d_p : \alpha_p \, option \, |\!) \tag{4.34}$$

where each (b_i, d_i) interface is obtained according to Definition 4.5.

Definition 4.6 (Sequential Composition)

The sequential composition of a component pair $M_1, M_2 \models SComp$ is denoted $M_1;;M_2$, implements an n-m standard interface, and is defined as:

$$\begin{aligned}
S &= S_1 \tilde{\times}_{\mathcal{L}} S_2 \;=\; (\!| \, m_1 : S_1,\; m_2 : S_2 \, |\!) \\
s0 &= (\!| \, m_1 = s0_1,\; m_2 = s0_2 \, |\!) \\
I &= \mathcal{BI}^m \,\tilde{\odot}\, \mathcal{DI}^n \\
O &= \mathcal{BO}^n \,\tilde{\odot}\, \mathcal{DO}^m \\
\delta &= \lambda s, i. \, (\!| m_1 = \delta_1 \, \left(s.m_1,\, \mathcal{IB} \,\tilde{\odot}\, \mathcal{DI}^n(i)\right),\; m_2 = \delta_2 \, \left(s.m_2,\, \mathcal{BI}^m(i) \,\tilde{\odot}\, \mathcal{ID}\right) \, |\!) \\
\omega &= \lambda s, i. \; \mathcal{BO}^n \left(\omega_1 \left(s.m_1,\; \mathcal{IB} \,\tilde{\odot}\, \mathcal{DI}^n(i)\right)\right) \,\tilde{\odot}\, \mathcal{DO}^m \left(\omega_2 \left(s.m_2,\; \mathcal{BI}^m(i) \,\tilde{\odot}\, \mathcal{ID}\right)\right)
\end{aligned}$$

where the labelling $\mathcal{L}$ is $\{(1, m1), (2, m2)\}$.

The correctness statement for the sequential composition, the corresponding lemma to Lemma 4.6, is more complex then the one for the parallel composition. The correctness statement has to take into account that external inputs are turned into internal ones. These inputs need to become bound variables and substituted according to Definition 4.5.

Lemma 4.7 (Sequential Composition Correctness)

Given M_1, M_2 with input signals $i_1^t \in I_1$ and $i_2^t \in I_2$, and let $\mathcal{IB}$ and $\mathcal{ID}$ be the internal signals from Equation 4.33 and Equation 4.34. Moreover, let $M = M_1 \;;;\; M_2$ be the sequential composition of M_1 and M_2 with input signal $i = \lambda t.\ \mathcal{BI}^p(i_2^t) \mathbin{\tilde{\odot}} \mathcal{DI}^n(i_1^t)$. Then, if

$$\langle M_1, i_1 \rangle \models_{\mathcal{A}_1} P_1 \wedge \langle M_2, i_2 \rangle \models_{\mathcal{A}_2} P_2$$

holds, the sequential composition satisfies:

$$\langle M, i \rangle \models_{\mathcal{A}} P_1[\mathcal{BI}(i_1^t)/\mathcal{IB}] \wedge P_2[\mathcal{DI}(i_2^t)/\mathcal{ID}]$$

with $A = \mathcal{A}_1[\mathcal{BI}(i_1^t)/\mathcal{IB}] \cup \mathcal{A}_2[\mathcal{DI}(i_2^t)/\mathcal{ID}]$.

Proof Similar to the proof of Lemma 4.6, the proof is basically an application of the definition of sequential composition (Definition 4.6). The formalisation is more tedious then for parallel composition, but can also be mechanised in Isabelle. The mechanised proof makes heavy use of sledgehammer to automated large parts of it. ∎

4.2.3 Multiplex/Arbitrate Composition

In this section and the next one, two special operators are introduced: the multiplex/arbitrate composition and the replication operator. They are called special because they are not just binary operations, but parametrised in their components. The multiplex/arbitrate composition takes a set of Mealy machines as an operand, whereas the replication operator is a parametrised unary operator.

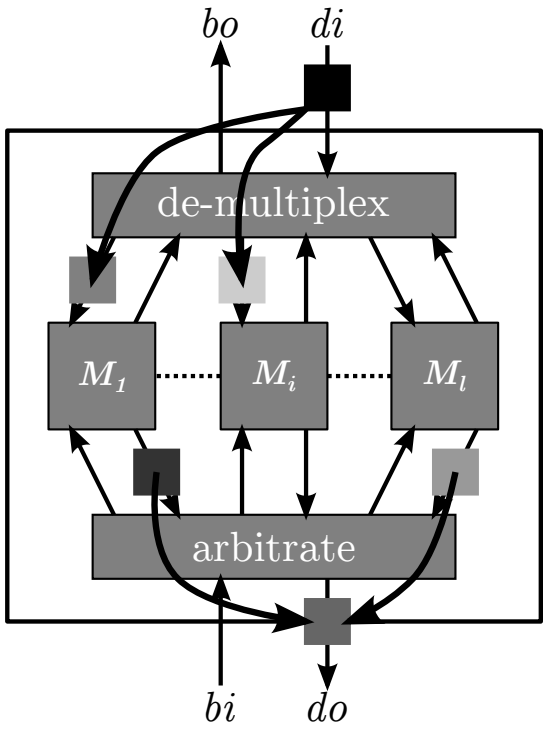


Figure 4.5: The Multiplex/Arbitrate Composition

The goal of the multiplex/arbitrate composition is to parallelize l (almost) arbitrary components in a structured way while maintaining the input and output interfaces. The only requirement for the l components is that they have to agree on their interfaces. Thus, for a set of l *inner components*

$$\mathcal{M} = \{M_1, \ldots, M_l\} \tag{4.35}$$

the individual interfaces have to match: for all $i \in [1, l]$

$$(I_i, O_i) = (\mathcal{BI}^m \tilde{\odot} \mathcal{DI}^n, \mathcal{BO}^n \tilde{\odot} \mathcal{DO}^m). \tag{4.36}$$

The construction principle is depicted in Figure 4.5. The composed system maintains the input and output interfaces of a single inner component; the input

and output domains of the newly created Mealy machine are then given by:

$$(I, O) = (\mathcal{BI}^m \tilde{\odot} \mathcal{DI}^n, \mathcal{BO}^n \tilde{\odot} \mathcal{DO}^m). \tag{4.37}$$

The operator is parametrised in two functions: a *multiplex function* and an *arbitration function*. Intuitively, the multiplex component *forwards* input signals at input interface i to input interfaces i of an arbitrary subset of internal components. The arbitration function provides the *inverse functionality* to the multiplex function: given the values provided by the i-th output interface of all l inner components, it generates the value of output interface i of the composed system.

$$\begin{aligned} mux : & Opt \times (\!| ibo_1 : \mathcal{BI}^n, \ldots, ibo_l : \mathcal{BI}^n |\!) \tilde{\odot} \mathcal{DI}^n \\ & \rightarrow \mathcal{BO}^n \tilde{\odot} (\!| idi_1 : \mathcal{DI}^n, \ldots, idi_l : \mathcal{DI}^n |\!) \end{aligned} \tag{4.38}$$

$$\begin{aligned} arb : & \ Opt \times \mathcal{BI}^m \tilde{\odot} (\!| ido_1 : \mathcal{DO}^m, \ldots, ido_l : \mathcal{DO}^m |\!) \\ & \rightarrow (\!| ibi_1 : \mathcal{BI}^m, \ldots, ibi_l : \mathcal{BI}^m |\!) \tilde{\odot} \mathcal{DO}^m \end{aligned} \tag{4.39}$$

Similarly to the data modification in Section 4.1.2, this operator also allows for an additional, optional component in the state space of the combined system. The optional component is only used by the multiplex and arbitration functions.

$$OPT = (Opt, \ opt^0 \in Opt, \ \delta_{opt} : Opt \times I \times O \rightarrow Opt) \tag{4.40}$$

Note, that in contrast to the step function of the data modification's optional component, the step function here also considers the output signals of the composed system, that is the signal produced by the arbitration component.

Taking all components into account, the state space and the initial state of the composed system are given by:

$$S = (\!| \, m_1 : S_1, \ldots, m_l : S_l, \mathit{opt} : \mathit{Opt} \, |\!) \tag{4.41}$$

$$s0 = (\!| \, m_1 = s0_1, \ldots, m_l = s0_l, \ \mathit{opt} = \mathit{opt}^0 \, |\!) \tag{4.42}$$

The remaining definitions are structured similarly to the definition of the sequential composition: first, requirements on the inner components are specified, then all inner signals are defined, and finally the overall composition is formulated in a compact way using the auxiliary definitions.

Looking at the construction in Figure 4.5, one can see that a "path" through the system in dataflow direction (data input—mux–inner component—arb—data output) consists of the sequential composition of three components. The multiplex function needs to provide Moore-like output interfaces. As it is basically a routing component, its data output signals should not depend on the busy signals from the inner components. Note that the multiplex component does not provide Moore-like input interfaces to the environment, as the busy output signals are defined as

$$\mathit{cmux.bo} = \bigvee \{ \mathit{ibos.ibo}_k \mid k \in [1, l] \wedge \mathit{cmux.do}_k \neq \mathbf{None} \} \tag{4.43}$$

with $\mathit{cmux} = \mathit{mux}\ (\mathit{opt}, \mathit{ibos} \tilde{\odot} \mathcal{DI}(i))$ for some input i, inner busy signals ibos, and current state of the optional component opt. The composed system is busy if one of the inner components is busy to which the current input data is passed on to. Hence, the composed system does not provide Moore-like input interfaces.

The assumptions on the multiplex function are summarised in the predicate P_{mux}.

$$P_{mux} = (\text{Eq. } 4.43) \wedge \forall k \in [1, l].\, (ibos.ibo_k.bo_j, cmux.idi_k) \models Moore_{out} \quad (4.44)$$

The arbitration function has to satisfy similar assumptions, and it has to provide Moore-like output interfaces. If $asel \subset [1, l]$ specifies the set of inner component indices that contribute to the current data output of the whole system, then the arbiter needs to forward the busy input to exactly those inner components. All other busy signals need to be active to prevent any other inner component from outputting data.

$$carb.ibo_k = \mathcal{BI}^m \vee (k \notin asel) \quad (4.45)$$

where $carb = arb\ (opt, \mathcal{BI}(i)\tilde{\odot}idos)$ for some input i, inner data outputs $idos$, and current state of the optional component opt. The arbitration function provides Moore-like output interfaces, but not Moore-like input interfaces, like the multiplex function.

$$P_{arb} = (\text{Eq. } 4.45) \wedge \forall k \in [0, l).\, (carb.ibi_k, idos_k) \models Moore_{out} \quad (4.46)$$

Thus, every inner component has to provide Moore-like outputs to avoid cyclic dependencies. Therefore, the set of inner components $\mathcal{M}$ has to ensure:

$$p_{\mathcal{M}} = \forall k \in [0, l).\, (M_i.bi, M_i.do) \models Moore_{out} \quad (4.47)$$

Using all these assumptions to eliminate dependencies in the definitions, the inner signals, ibi_k, idi_k, ibo_k, ido_k, can be defined because only the following dependencies remain:

- idi_k depends on the external input $i.di$ only.
- ido_k depends on idi_k only.
- ibi_k depends on ido_j for all $j \in [0, l)$ and the external busy input $i.bi$.
- ibo_k depends on ibi_k and idi_k.

These dependencies are free from circular, unresolvable ones and the inner signals can be defined in terms of the operator parameters (mux,arb) and the external signals. For inputs i and state s, the inner signals are defined as:

$$idi_k = (mux\,(opt,\ (\!|\, ibo_0 = \mathbf{T}^n, \ldots, ibo_{l-1} = \mathbf{T}^n, id = DIi \,|\!)).idi_k \tag{4.48}$$

$$ido_k = (\omega_k\,(s.m_k, (\!|\, bi = \mathbf{T}^m,\ di = idi_k \,|\!)).do \tag{4.49}$$

$$ibi_k = (arb\,(opt,\ \mathcal{BI}(i)\ \tilde{\odot}\ (\!|\, ido_0 = ido_0, \ldots, ido_{l-1} = ido_{l-1} \,|\!)).ibi_k \tag{4.50}$$

$$ibo_k = (\omega_k\,(s.m_k, (\!|\, bi = ibi_k,\ di = idi_k \,|\!)).bo \tag{4.51}$$

In order to simplify notation in this definition, the names of the inner signals are also used to refer to the respective inputs or outputs of a component, instead of introducing more origin-specific signal names. This way it is obvious which signals are connected to each other.

Definition 4.7 (Multiplex/Arbitrate Composition)

The multiplex/arbitrate composition is an operator parametrised in three components: mux, arb, and OPT. Its application to a set of Mealy machines $\mathcal{M} = \bigcup_{k\in[1,l]} M_k$ *with matching interfaces (Equation 4.36) is denoted* $\diamond\mathcal{M}$.

If $mux \models P_{mux}$, $arb \models P_{arb}$, *and for all* $k \in [1, l]$. $M_k \models Moore_{out}$, *the components of* $\diamond\mathcal{M}$ *are: I, O, S, and s0 according to Equations 4.37, 4.41,*

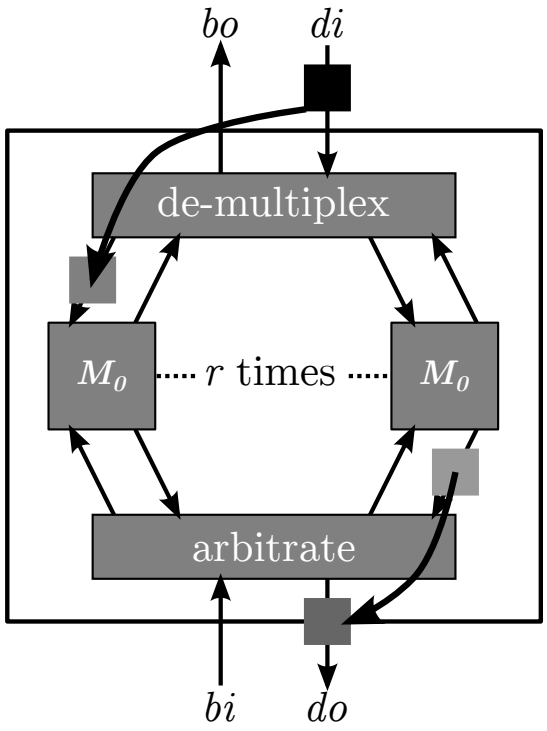


Figure 4.6: The Replication Operator

and 4.42, and

$$\delta = \lambda s.\, i. \tag{4.52}$$
$$(\!| m_1 = \delta_1\,(s.m_1, ibi_1 \mathbin{\tilde{\odot}} idi_1), \ldots, m_l = \delta_l\,(s.m_l, ibi_l \mathbin{\tilde{\odot}} idi_l),$$
$$opt = \delta_{opt}\,(s.opt, i)\,|\!)$$

$$\omega = \lambda s.\, i. \tag{4.53}$$
$$(\!|bo = (mux\,(opt,\ (\!|\, ibo_1 = ibo_1, \ldots, ibo_l = ibo_l\,|\!) \mathbin{\tilde{\odot}} \mathcal{DI}(i))).bo,$$
$$do = (arb\,(opt,\ \mathcal{BI}(i) \mathbin{\tilde{\odot}} (\!|\, ido_1 = ido_1, \ldots, ido_l = ido_l\,|\!))).do|\!)$$

4.2.4 Replication Operator

The replication operator is closely related to the multiplex/arbitrate composition, but it is less flexible and is has more requirements on the individual components. The goal is the controlled, parallel execution of r *copies* of a Mealy machine M while also maintaining the input and output interfaces of M. Besides requiring stronger assumptions on the inner components, the construction requires the multiplex component to select a unique inner component for every possible input, and the arbitration function to only select a single inner component to output data from at a given time. A schematic overview is shown in Figure 4.6.

Even though this construction is less flexible than the more generic multiplex/arbitration composition, the benefit is that a major generic correctness results about liveness properties can be shown (Theorem 4.8). However, the case studies in Chapters 5 and 6 show that in many cases the replication operator is expressive enough to construct a desired transformation.

Replication is also parametrised in a *mux* function, an *arb* function, and an OPT component. Additionally the number of replications, $r \in \mathbb{N}$, is a parameter as well. In contrast to the multiplex/arbitrate composition, replication is a unary operator on a single Mealy machine M_0. The state space and the input domains are:

$$(I, O) = (I_0, O_0) \tag{4.54}$$

$$S = (\!| \, m_1 : S_0, \ldots, m_r : S_0, opt : Opt \, |\!) \tag{4.55}$$

$$s0 = (\!| \, m_1 = s0_0, \ldots, m_r = s0_0, opt = opt^0 \, |\!) \tag{4.56}$$

with an OPT component according to Equation 4.40.

The assumptions on the multiplex and arbitration functions are formulated as a strengthening of the P_{mux} predicate from Equation 4.44 and the P_{arb} predicate from Equation 4.46, respectively.

$$P_{rmux} = P_{mux} \wedge |\{ibos.ibo_k \mid k \in [1,r] \wedge cmux.do_k \neq \mathbf{None}\}| = 1 \quad (4.57)$$

$$P_{rarb} = P_{arb} \wedge |arbsel| \leq 1 \quad (4.58)$$

In terms of generic correctness results, this strengthening of the multiplex and arbitration properties will leverage a much stronger liveness result. For the inner components, we require Moore-like output interfaces as before. Since the basic construction is analogous to the one of the multiplex/arbitrate operator, the internal signals are defined as in Equations 4.48 to 4.51, using ω_0 for all inner components, and parameter r instead of l.

Definition 4.8 (Replication Operator)

The replication operator applied to a Mealy machine M_0, parametrised with $r \in \mathbb{N}$, mux, arb, and OPT is denoted $\Re M_0$. If $mux \models P_{rmux}$, $arb \models P_{rarb}$, and $M_0 \models Moore_{out}$, the components of $\Re M_0$ are: I, O, S, and s0 as defined in Equations 4.54, 4.55, and 4.56, and

$$\begin{aligned}\delta \; &= \lambda s, i. \quad (4.59)\\ &\quad (\!| \, m_1 = \delta_0\,(s.m_1,\, ibi_1 \tilde{\odot} idi_1), \ldots, m_r = \delta_0\,(s.m_r,\, ibi_r \tilde{\odot} idi_r),\\ &\quad\; opt = \delta_{opt}\,(s.opt, i) \, |\!)\\ \omega \; &= \lambda s, i. \quad (4.60)\\ &\quad (\!| bo = (mux\,(opt,\, (\!| \, ibo_1 = ibo_1, \ldots, ibo_r = ibo_r \, |\!) \,\tilde{\odot}\, \mathcal{DI}(i))).bo,\\ &\quad\; do = (arb\,(opt,\, \mathcal{BI}(i) \,\tilde{\odot}\, (\!| \, ido_1 = ido_1, \ldots, ido_r = ido_r \, |\!))).do \, |\!)\end{aligned}$$

To instantiate the replication operator, the individual components need to satisfy the following assumptions.

1. the inner component component (M_0) is correct and ensures liveness,
2. the multiplex function is *correct*, i. e. $mux \models P_{rmux}$, and
3. the arbitration is satisfies P_{rarb}, i. e. $arb \models P_{rarb}$, and is *fair* with respect to an active data signal from an inner component.

Assumption 4.9 (Inner Component)
Let M_0 be the state machine to be replicated using the replication operator. Then, M_0 has to provide Moore-like outputs, i. e. $M_0 \models Moore_{out}$, and M_0 has to satisfy liveness:

$$\forall i \in [1,n].\forall x \in \mathbf{dom}(di_i, I).\ \neg b_o^t \wedge (di_i^t = \mathbf{Some}\ x) \\ \implies (\exists j \in [1,m], k \in \mathbb{N}.\ do_j^{t+k} = \mathbf{Some}\ f_{i,j}(x))$$

where $f_{i,j} : \mathbf{dom}(di_i, I) \to \mathbf{dom}(do_j, O)$ represents a possible data modification by M_0 from input di_i to output do_j.

The following theorem states that given Assumption 4.9 and the assumptions on the multiplex and arbitration functions, the derived system satisfies liveness

Theorem 4.8 (Correctness of Replication)
If the inner state machine satisfies Assumption 4.9, the system obtained using the replication operator satisfies this assumption again if the multiplex and arbitration functions ensure the previously mentioned assumptions.

PROOF The Isabelle proof of Theorem 4.8 is mainly obtained by unfolding definitions and assumptions. An induction is needed to conclude the stable input signals for the time interval from Assumption 4.9 and the fairness of the arbitration function. ■

4.3 Transformations

Transformations are used to introduce and add new protocol features. They are basically operators on Mealy machines: an existing model is extended with the feature that is modelled by the transformation. More abstractly, transformations are a feature-focused abstraction layer above the basic framework components. They are formalised as *parametrized, unary operators* on Mealy machines and each implements a specific protocol feature. Transformations are not protocol specific in general, although they may depend on certain assumptions about the target system. Naturally, many protocol features are topology-specific or have implicit dependencies on other features: for example packet reordering implicitly depends on some other feature that can conditionally delay packets.

Transformations are best introduced with the feature they implement. So, the details of the transformations covered in this dissertation are presented together with the case studies in Chapters 5 and 6. This section introduces the features briefly with a focus on a protocol independent characterisation.

Also, even though transformations are explained together with the case studies, they are specified in a parametrized form and as protocol-independently as possible, and only then instantiated for a specific protocol. The features covered by the transformations are inspired by the case studies from Chapters 5 and 6,

but many other widely-used communication protocols implement these features (or a subset) so that these transformation can also be used to model and verify other protocols.

This dissertation covers five features of which two are bus protocol specific and three are specific to packet-based, that is usually point-to-point, protocols.

Pipelining

Bus protocols often implement a transaction-based communication where each transaction consists of an *address phase* and a successive *data phase*: during the address phase, all the control data is provided by the initiator of a transaction (the sender); during the data phase the actual payload is exchanged. In the simplest case, each data phase immediately succeeds its corresponding address phase, and transactions happen strictly sequentially, one after the other. Pipelining is a transformation tailored to bus protocols that implement separate control and data buses, and many of the popular SoC protocol provide such an interconnect: for example, the Open Core Protocol [OCP05], IBM's CoreConnect [IBM], Altera's Avalon [Alt03], the asynchronous Marble protocol [BF98], the PI-Bus [Sei94], STMicroelectronics' STBus [STM], and the Virtual Component Interface standard [All01]. The first four of those, also have support for burst transfers; the next feature.

In this context, *pipelining* refers to pipelining such address-data-phase transactions: if the interconnect provides physically separate control and data buses, transactions can be parallelized, such that each data phase happens in parallel with the control phase of the next transaction. Chapter 5 details the pipelining transformations as an instantiation of the arbitrate/multiplex operator.

The basic idea is to execute two instances of the Mealy machine for sequential transfers in parallel and use the arbitration and multiplex components to break the symmetry, to prevent *illegal states* compared to a purely interleaved execution, and to combine the outputs properly.

Burst Transfers

Again, considering a bus protocol implementing the address-data-phase transaction style, a *burst transfer* is a way to combine sequential transactions of the same type, i. e. read or write, and to successive addresses into a single one. For example, if a device core attached to a bus controller wants to read data from addresses a to $a + b - 1$ (with b sufficiently small), the device core can initiate a burst read transaction from address a of size b.

The idea of the transformation is to *expand* a burst transaction from the device core to multiple standard transactions. Chapter 5 defines the burst transfer transformation as an instantiation of the data modification component.

Virtual Channels

Virtual channels is a feature usually found in packet-based point-to-point protocols. As the name suggests, it is a feature that *simulates* multiple communication channels on top of a single physical one that are (virtually) independent from each other. In practical terms, this means that virtual channels are used to categorise and prioritise packets sent over the same physical communication channel.

To achieve this, intuitively, the construction has to provide three basic features:

- Separate send or receive buffers for each virtual channel to handle the data elements for each channel separately.

- A total mapping from data elements to virtual channels, so that each data element can be mapped to a unique virtual channel. While the uniqueness is not strictly required for the overall construction, it seems more than reasonable considering current communication architectures. Mapping a data element to more than a single virtual channel would also result in duplication of the element.

- An arbitration function to arbitrate among the different virtual channels because in the end they have to be sent using the physical channel.

This intuition already shows that virtual channels seem to be a straightforward application of the replication operator. Chapter 6 details the instantiation within the PCI Express case study.

Flow Control

Similar to virtual channels, *flow control* is also a feature tailored to packet-based point-to-point communication protocols. Its purpose is to prevent packet loss because a recipient does not have enough buffer space to receive the packet from the interconnect. Thus, flow control is a mechanism for the sender of a data element to check whether the receiver has enough space in its local receive buffer before sending the data element. In addition to a packet-based communication network, flow control also requires bidirectional communication between sender and receiver: a *data channel*, to transport data elements, and a *control channel* in opposite direction. Sender and receiver are named according to the direction of the data channel.

The working principle of a flow control mechanism can be summarised as

follows: the receiver implements one or more flow control (receive) buffers to store data elements. The sender maintains a counter for each of those buffers. These counters are used to maintain a lower bound on the available space in the receiver's buffers. Before transmitting a data element, the sender checks that the receiver has enough available buffer space using these counters. The control channel is needed to update the sender's counter values: the sender can only track a reduction of available space when a data element is sent. Any increase of available space, i. e. data moved from a receive buffer to the host system, happens unnoticed by the sender. Thus, the receiver sends regular updates of the available space to the sender using the control channel.

Chapter 6 details both the transmit part as well as the receive part of the transformation. The receive part is realised using a multiplex/arbitrate component, a replication component, as well as a sequential composition. Thus, even though is sounds similar to virtual channels, it is a more complex construction. The transformation for the transmit part, however, can be specified using a simple data modification component and a sequential composition.

Reordering

Packet reordering is a feature that has an implicit dependency on a feature that allows for selective packet delaying, such as flow control. The basic idea is to improve overall performance in case a sender cannot transmit the current data packet. Then, reordering allows the sender to check whether the next element in its send buffer can be transmitted. Similar to flow control and virtual channels, reordering is a feature usually related to packet-based point-to-point architectures. Moreover, reordering is a transmit-part only transformation.

In order for this feature to work in a sensible way, the construction has to provide the following:

- A *selective packet transmit function* which specifies if a current data packet can be transmitted. Among possibly other aspects, this function has to depend on the current data packet, in order to be selective.
- A *set of overtaking rules* which specify if two packets may be reordered or not.

The transformation presented in Chapter 6 assumes the first function to be an input which is provided by some other feature, such as the flow control mechanism. The overtaking rules are a parameter of the transformation. This way, reordering can be specified independently from other features despite its implicit dependency. Note that the specification does not necessarily depend on the selective packet blocking: the transformation also works without it, but then there is no benefit from reordering as packets are never reordered in that case.

Chapter 5

ARM AMBA 2 Advanced High-Performance Bus

This chapter and the next one present the application of the framework to two case studies: the AMBA Advanced High-performance Bus protocol (AHB) from ARM [ARM99] and the PCI Express point-to-point communication protocol [PS06]. In both cases, crucial parts of the protocol specification are modelled using incremental modelling and key correctness properties are shown and maintained during the modelling process. The choice of a bus protocol and a point-to-point protocol shows the breadth of the methodology.

The Advanced High-Performance Bus protocol is a part of the Advanced Microcontroller Bus Architecture (AMBA) from ARM. The AMBA specification consists of three bus protocols:

- The Advanced High-performance Bus (AHB)
- The Advanced System Bus (ASB)

- The Advanced Peripheral Bus (APB)

The application area of the AHB protocol is high-performance, high clock frequency system modules where it is meant to act as a system backbone bus. It supports the efficient connection of processors and on-chip memories. The ASB protocol is similar to the AHB protocol but omits some of the latter's high-performance features. The APB protocol targets the interconnection of low-power system modules and is optimised for reduced interface complexity. As the goal of the case study is to show the application of incremental modelling to complex, high-performance protocols, AHB was chosen as a suitable case study.

The AHB protocol is an arbiter-based, master-slave communication protocol. A typical communication system consists (at least) of the following components:

- One or more *bus masters*: a bus master is able to initiate bus operations (reads or writes) by providing address and control information. Only one bus master can use the bus at any one time. Typical masters include processors, DSPs, or test interfaces.

- One or more *bus slaves*: a bus slave responds to bus operations (reads or writes) within a given address space. Typical slaves include internal memories, APB bridges, and external memory interfaces.

- One *bus arbiter*: the bus arbiter grants bus access to one of the bus masters and ensures that only one bus master at a time is allowed to initiate a bus operation. The only restriction on the arbitration algorithm is that it is fixed. The concrete realisation can be chosen depending on the application requirements.

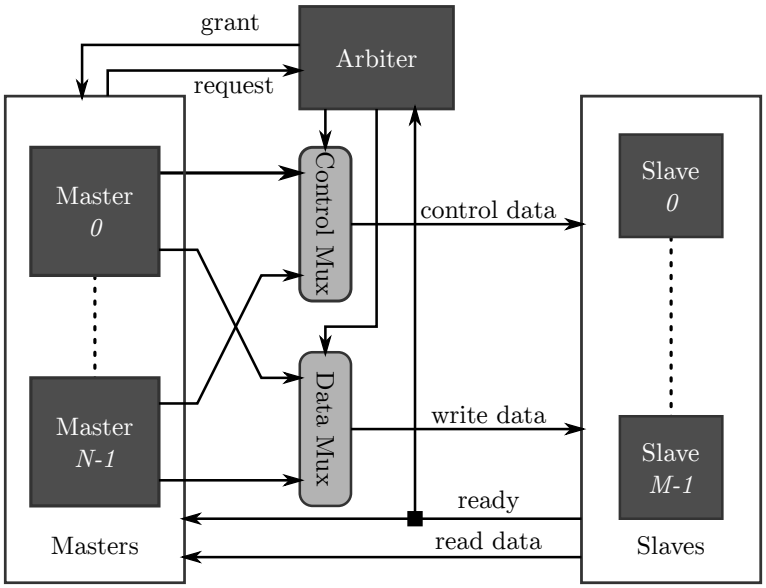


Figure 5.1: Sample AHB Topology

- One *bus decoder*: the bus decoder decodes a part of the address information provided by a master for a bus operation and provides a select signal for the addressed slave. Thus, the address information of an operation consists of a part that specifies the addressed slave, and a part that determines the address within the slave's address space.

As detailed in the next section, the model presented here abstracts from the decoder. A schematic overview of a typical AHB communication system without a decoder is depicted in Figure 5.1. AMBA AHB implements features required for high-performance systems which include *pipelined transactions*, *burst transfers*, *split transactions*, *single-cycle bus master handover*, and *non-tristate implementation*. The case study focuses on pipelined transactions and burst transfers, but the models also provide single-cycle bus handover and a non-tristate implementation.

5.1 Bus Signals and Transactions

In the AMBA 2 AHB protocol specification, a single unit of communication (*transaction*) between a master and a slave consists of two separate phases: an *address phase* and a successive *data phase*. Each transaction is either a *read transfer* or a *write transfer*. In the first case, an addressed slave delivers locally stored data to a master; in the second case, the master provides data to be stored in a slave's memory.

During the address phase, the master that is granted bus access provides *control data* for the transaction. This control data includes an address, which specifies both the slave and a memory location within the slave, a flag indicating a read or write transaction, and a flag indicating an *idle transfer*. An idle transfer is a special, 'empty' transaction: in case the bus is granted to a master that does not have any data to transmit, the master has to initiate such an idle transfer. This case can occur because the protocol specification requires that the bus is granted to some master at all times. An idle transfer is defined formally in Definition 5.3 once all the bus signals have been introduced. In the following, $\mathcal{C}_{bus}$ is used to refer to the domain of the control bus which is given by:

$$\mathcal{C}_{bus} = (\!|\, trans : \mathbb{B},\ wr : \mathbb{B},\ addr : \mathbb{N} \times \mathbb{N} \,|\!) \tag{5.1}$$

Given an element $ctrl \mathrel{\tilde{\in}} \mathcal{C}_{bus}$, the following semantics applies:

- $ctrl.trans = \mathbf{F}$ indicates an idle transfer, whereas $ctrl.trans = \mathbf{T}$ signals a standard transaction.
- $ctrl.wr = \mathbf{F}$ specifies a read transfer, and $ctrl.wr = \mathbf{T}$ indicates a write

transfer.

- $ctrl.addr = (addr_{sl}, addr_{mem})$ addresses slave $addr_{sl}$ and the location $addr_{mem}$ in that slaves local memory.

During the data phase, either the addressed slave delivers the requested data at the end of the phase in case of a read transaction, or the master provides the data to be written during the phase in case of a write transaction. To implement this bidirectional dataflow between masters and slaves, two bus signals are used: *wdata* for write data provided by a master, and *rdata* for read data provided by the slaves. In the following, $\mathcal{D}$ is the set of possible data elements to be transferred and stored in the slaves' memories. Then, the domain of the data bus $\mathcal{D}_{bus}$ is given by the following labelled tuple:

$$\mathcal{D}_{bus} = (\!| \, wdata \in \mathcal{D}, rdata \in \mathcal{D} \, |\!) \tag{5.2}$$

The end of each phase, both address and data phase, is signalled using a special ready signal *rdy* which semantically corresponds to the inverse of a busy signal: the end of a phase is signalled by an active *rdy* signal. The *rdy* signal is generated by the addressed slave. The complete definition of the AMBA communication bus signals is given in Definition 5.1.

Definition 5.1 (AMBA Communication Bus Signals)

The communication bus at time t is defined as a signal triple

$$bus^t = (rdy^t, cbus^t, dbus^t) \in \mathbb{B} \times \mathcal{C}_{bus} \times \mathcal{D}_{bus}$$

where the components are:

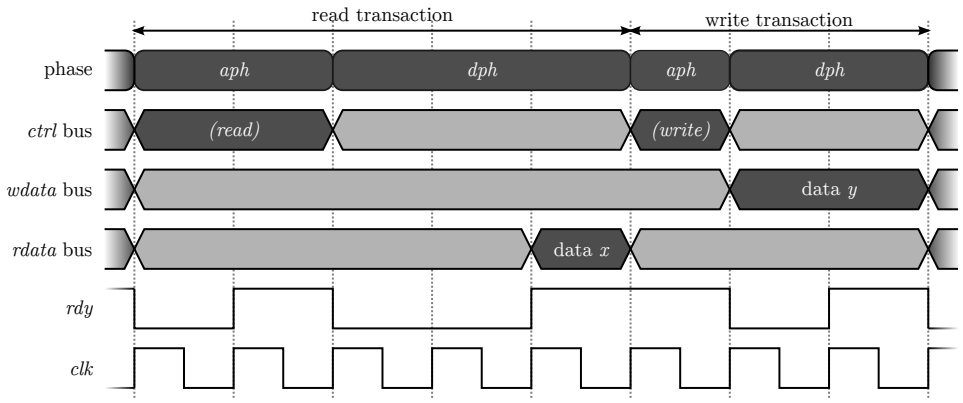


Figure 5.2: Sample Read and Write Transactions

- rdy^t *is the bus ready signal. A signal* $rdy^t = \mathbf{T}$ *indicates the end of an address or data phase at time* t*.*
- $cbus^t$ *is the value of the control bus at time* t *as given by Equation 5.1.*
- $dbus^t$ *is the value of the data bus at time* t *as given by Equation 5.2.*

Figure 5.2 depicts the bus signals for a sample sequence of a read and a write transaction with corresponding address and data phases. Note that every phase ends with an active *rdy* signal. For the read transaction, the *cbus* bus provides the control signals to indicate a data transfer ($trans = \mathbf{T}$), a read transaction ($wr = \mathbf{F}$), and an address ($addr = a1$). Thus the value of the *cbus* bus is $(\mathbf{T}, \mathbf{F}, a1)$ (cf. Equation 5.1). At the end of the data phase, the slave provides the requested data element, say $x \in \mathcal{D}$, on the read part of the data bus (*rdata*). Similarly for the write transfer: the master provides control signals, say $(\mathbf{T}, \mathbf{T}, a2)$, during the address phase, but during the data phase, the master also provides the data to be written on the write data part of the data bus (*wdata*). Note that, in contrast to read data, write data has to be stable for the length of the data phase.

In order to model the components of the communication system using the framework from Chapter 4, the interface between the components and the bus has to obey the standard interface convention, i. e. each interface has to consist of a busy and a data signal. Slaves have to receive the control data and the write data from the bus. Read data has to be provided by the slaves. Thus, to attach a slave S_j for $j \in [1, N_S]$ and N_S is the number of slaves in the communication system, the bus has to provide two output interfaces—one for control data, one for write data—and one input interface for the read data.

$$I_{bus,S}[j] = (\!| \, bi_{ctrl} : \mathbb{B}, bi_{wdata} : \mathbb{B}, di_{rdata} : \mathcal{D}\, option \, |\!) \tag{5.3}$$

$$O_{bus,S}[j] = (\!| \, bo_{rdata} : \mathbb{B}, do_{ctrl} : \mathcal{C}_{bus}\, option, do_{wdata} : \mathcal{D}\, option \, |\!) \tag{5.4}$$

Analogously, masters produce control and write data, and read data from the bus. Thus, to attach a master M_k for $k \in [1, N_M]$ and N_M is the number of masters in the communication system, the bus has to provide two input interfaces and one output interface.

$$I_{bus,M}[k] = (\!| \, bi_{rdata} : \mathbb{B}, di_{ctrl} : \mathcal{C}_{bus}\, option, di_{wdata} : \mathcal{D}\, option \, |\!) \tag{5.5}$$

$$O_{bus,M}[k] = (\!| \, bo_{ctrl} : \mathbb{B}, bo_{wdata} : \mathbb{B}, do_{rdata} : \mathcal{D}\, option \, |\!) \tag{5.6}$$

The interface components have to be mapped to the AMBA bus signals from Definition 5.1. According to the specification, bus signals produced by the masters are obtained using multiplex circuits, whereas bus signals provided by slaves are generated using or-trees. The *rdy* and the *dbus.rdata* signals are generated by the slaves. The *rdy* signal is obtained from the busy inputs produced by the slaves and the read data output. The semantics is that lowering the busy signal

or producing read data corresponds to an active ready bus signal. Read data is simply obtained from a slave that has data to output. Let $i = i^t_{bus,S}$, then

$$rdy^t = \bigvee_{j=0}^{N_S-1} (\neg i[j].bi_{ctrl} \vee \neg i[j].bi_{wdata} \vee (i[j].di_{rdata} \neq \textbf{None})) \tag{5.7}$$

$$dbus^t.rdata = \bigvee_{j=0}^{N_S-1} i[j].di_{rdata} \tag{5.8}$$

Bus access for masters is granted by the arbiter. Thus an additional input interface is needed to select the signals from the master that has access to a bus. The arbiter grants access to the control and data buses at time t using two signals: $cgrant^t \in [1, N_M]$ and $dgrant^t \in [1, N_M]$. New grant signals are generated when the bus ready signal indicates the end of a transaction phase.

$$bo^t_A = \neg rdy^t \in \mathbb{B} \tag{5.9}$$

$$di^t_A = (cgrant^t, dgrant^t) \in (\mathbb{N} \times \mathbb{N})\, option \tag{5.10}$$

$$cbus^t = i^t_{bus,M}[cgrant^t].di_{ctrl} \tag{5.11}$$

$$dbus^t = i^t_{bus,M}[dgrand^t].di_{wdata} \tag{5.12}$$

Definition 5.2 (AHB Communication Bus)

The AMBA AHB communication bus that connects N_M masters and N_S slaves is given by the state machine

$$(S_{bus} = \emptyset,\ I_{bus},\ O_{bus},\ s0_{bus} = \emptyset,\ \delta_{bus} = (\lambda\ s, i.s),\ \omega_{bus})$$

where the input and output domains are defined as follows: let $j \in [1, N_M]$ and

$k \in [1, N_S]$, *then*

$$\begin{aligned} I_{bus} = & \widetilde{\prod}_j (\!| \, bi_{rdata,M_j} \in \mathbb{B}, di_{ctrl,M_j} \in \mathcal{C}_{bus}\, option, di_{wdata,M_j} \in \mathcal{D}\, option \, |\!) \, \tilde{\circ} \\ & \widetilde{\prod}_k (\!| \, bi_{ctrl,S_k} \in \mathbb{B}, bi_{wdata,S_k} \in \mathbb{B}, di_{rdata,S_k} \in \mathcal{D}\, option \, |\!) \, \tilde{\circ} \\ & di_A \\ O_{bus} = & \widetilde{\prod}_j (\!| \, bo_{ctrl,M_j} \in \mathbb{B}, bo_{wdata,M_j} \in \mathbb{B}, do_{rdata,M_j} \in \mathcal{D}\, option \, |\!) \, \tilde{\circ} \\ & \widetilde{\prod}_k (\!| \, bo_{rdata,S_k} \in \mathbb{B}, do_{ctrl,S_k} \in \mathcal{C}_{bus}\, option, do_{wdata,S_k} \in \mathcal{D}\, option \, |\!) \, \tilde{\circ} \\ & bo_A \end{aligned}$$

The output function is given by the labelled tuple composed of the following signals:

$$\begin{aligned} o^t_{bus,M}[j].bo_{ctrl} &= o^t_{bus,M}[j].bo_{wdata} = bo^t_A = \neg rdy^t \\ o^t_{bus,s}[k].bo_{rdata} &= \mathbf{F} \\ o^t_{bus,M}[j].do_{rdata} &= dbus^t.rdata \\ o^t_{bus,s}[k].do_{ctrl} &= cbus^t \\ o^t_{bus,s}[k].do_{wdata} &= dbus^t.wdata \end{aligned}$$

The bus signals are defined according to Equations 5.7, 5.8, 5.11, and 5.12.

To illustrate Definition 5.2, imagine a simple system with one master and two slaves. Then, the input and output domains are given by the following labelled tuples:

$$\begin{aligned} I_{bus} = (\!| \, & bi_{rdata,M_0} \in \mathbb{B}, di_{ctrl,M_0} \in \mathcal{C}_{bus}\, option, di_{wdata,M_0} \in \mathcal{D}\, option, \\ & bi_{ctrl,S_0} \in \mathbb{B}, bi_{wdata,S_0} \in \mathbb{B}, di_{rdata,S_0} \in \mathcal{D}\, option, \end{aligned}$$

$$
\begin{aligned}
& bi_{ctrl,S_1} \in \mathbb{B}, bi_{wdata,S_1} \in \mathbb{B}, di_{rdata,S_1} \in \mathcal{D}\, option, \\
& di_A \,\}\!) \\
O_{bus} = (\!\{\, & bo_{ctrl,M_0} \in \mathbb{B}, bo_{wdata,M_0} \in \mathbb{B}, do_{rdata,M_0} \in \mathcal{D}\, option, \\
& bo_{rdata,S_0} \in \mathbb{B}, do_{ctrl,S_0} \in \mathcal{C}_{bus}\, option, do_{wdata,S_0} \in \mathcal{D}\, option, \\
& bo_{rdata,S_1} \in \mathbb{B}, do_{ctrl,S_1} \in \mathcal{C}_{bus}\, option, do_{wdata,S_1} \in \mathcal{D}\, option, \\
& bo_A \,\}\!)
\end{aligned}
$$

To conclude the specification of the bus signals, the previously mentioned concept of an *idle transaction* is detailed and formally defined. The AHB specification requires that bus access is granted to some master at all times. Also, in order to make progress, the system depends on an active bus ready signal which is generated by slaves. Thus, the model has to consider the case in which no master requests bus access and no master has data to be transmitted. While the specification leaves the arbitration function unspecified, and so does the model presented here, it is assumed that in case no master requests the bus, the bus is granted to a default master $defM \in [1, N_M)$. This master initiates an idle transaction that is send to a default slave $defS \in [1, N_S)$ addressing memory location $defM \in \mathbb{N}$. The slave only acknowledges the transaction by raising its ready output. The control and data bus signals for an idle transaction are given by the following definition.

Definition 5.3 (Idle Transaction)

The values of the bus signals for an idle transaction are:

$$ictrl = (\!\{\, trans = \mathbf{F}, wr = \mathbf{F}, addr = (defS, defM) \,\}\!)$$

$$iwdata = \textbf{None}$$

$$irdata = \textbf{None}$$

5.2 Arbiter

The focus of this case study is on the bus master, but to specify a complete communication system, arbiter and slave are introduced briefly as well.

The arbiter grants bus ownership to one of the masters. The interface between the arbiter and the communication bus has already been specified in Equations 5.9 and 5.10. The arbiter generates a pair of grant signals

$$do_{bus} = (cgrant^t, dgrant^t) \in (\mathbb{N} \times \mathbb{N})option$$

which specify control ($cgrant$) and data ($dgrant$) bus ownership at time t.

A master can request bus ownership from the arbiter by raising a request signal. This signal is modelled using the busy signal of the simple output interface from the arbiter to each master. For each master $i \in [1, N_M]$, the arbiter provides an output interface

$$(bi_{M,i}, do_{M,i}) \in (\mathbb{B},\ \mathbb{B}\ option) \tag{5.13}$$

with the following semantics:

$$req_i = \neg bi_{M,i} \tag{5.14}$$

$$do_{M,i} = \begin{cases} \textbf{Some T} & : cgrant = i \wedge i \in [1, N_M] \\ \textbf{None} & : \text{otherwise} \end{cases} \tag{5.15}$$

$$grant_i = (do_{M,i} = \textbf{Some T}) \tag{5.16}$$

The arbiter is modelled as a simple Mealy machine in the framework. The arbiter state stores the (current) grant values, the last grant value, a vector of pending requests from masters, and a flag that indicates which phase the sequential communication bus is currently in. Thus, the arbiter configuration, the input record, and the output record are given by:

$$S_{arb} = (\!| \, cgr : \mathbb{N}, dgr : \mathbb{N}, preqs : \mathbb{B}^{N_M}, aph : \mathbb{B} \, |\!) \tag{5.17}$$

$$I_{arb} = (\!| \, bi_{bus} : \mathbb{B}, bi_{M,1} : \mathbb{B}, \ldots, bi_{M,N_M} \, |\!) \tag{5.18}$$

$$O_{arb} = (\!| \, do_{bus} : (\mathbb{N} \times \mathbb{N}) \; option, do_{M,1} : \mathbb{B} \; option, \ldots, do_{M,N_M} \; option \, |\!) \tag{5.19}$$

$$s0_{arb} = (\!| \, cgr = defM, dgr = defM, preqs = \mathbf{F}^{N_M}, aph = \mathbf{F} \, |\!) \tag{5.20}$$

The output signals of the arbiter are obtained from the current state in a straightforward way.

$$\begin{aligned} \omega_{arb} = \lambda \; s, i. \; (\!| \, do_{bus} = \textbf{Some} \, (s.cgr, s.dgr), \; do_{M,1} = \textbf{Some} \, unary(cgr)[0], \\ \ldots, do_{M,N_M} = \textbf{Some} \, unary(cgr)[N_M - 1] \, |\!) \end{aligned} \tag{5.21}$$

The operator *unary* defines the *unary*, or *1-hot*, bit vector encoding for a natural number:

$$unary(n)[i] = \mathbf{T} \Leftrightarrow n = i \text{ for some } n \in \mathbb{N} \tag{5.22}$$

The actual arbitration algorithm is modelled using an uninterpreted *arbitration function* **af** that returns a new grant value depending on the current state of the master and the request signals from the masters.

$$\mathbf{af} : S_{arb} \times \mathbb{B}^{N_M} \to \mathbb{N} \tag{5.23}$$

In order to argue about correctness and liveness of the communication system, this arbitration function is assumed to be fair with respect to the masters' request signals, and, of course, to provide only grant values in the interval $[1, N_M]$.

Assumption 5.4 (Valid Arbitration Function)
The arbitration function **af** *has to satisfy the following two properties: for all* $s^t \in S_{arb}$ *and for all* $req^t \in \mathbb{B}^{N_M}$

$$\mathbf{af}(s^t, req^t) \in [1, N_M] \text{ and}$$
$$req_i^t \implies \exists\, k.\ \mathbf{af}(s^{t+k}, req^{t+k}) = i$$

Ownership of the bus changes at the end of an address or data phase, thus the ready signal is used to trigger an update of the grant signals using the **af** function.

The sequential arbiter updates the grant values at the end of the data phase and updates both, *cgrant* and *dgrant*, at the same time.

$$\begin{aligned}\delta_{arb,seq} =& \lambda\ s, i.\ \text{let}\\ & cgr' = \begin{cases} \mathbf{af}(s, req_1 \ldots req_{N_M}) & \neg s.aph \wedge \neg i.bi_{bus} \\ s.cgr & : \text{otherwise} \end{cases}\\ & dgr' = cgr'\end{aligned}$$

$$
\begin{aligned}
& preqs' = s.preqs \vee (req_1 \ldots req_{N_M}) \\
& aph' = \begin{cases} \neg s.aph & : \neg i.bi_{bus} \\ s.aph & : \text{otherwise} \end{cases} \\
\text{in } & (\!| \, cgr = cgr', dgr = dgr', preqs = preqs', aph = aph' \, |\!)
\end{aligned} \tag{5.24}
$$

The step function of the pipelined arbiter is similar, but the two grant values are updated at the end of each phase to support pipelined masters. This also means that the *aph* flag is not used in this variant.

$$
\begin{aligned}
\delta_{arb,pip} = & \lambda \; s, i. \text{ let} \\
& cgr' = \begin{cases} \mathbf{af}(s, req_1 \ldots req_{N_M}) & : \neg i.bi_{bus} \\ s.cgr & : \text{otherwise} \end{cases} \\
& dgr' = \begin{cases} s.cgr & : \neg i.bi_{bus} \\ s.dgr & : \text{otherwise} \end{cases} \\
& preqs' = s.preqs \vee (req_1 \ldots req_{N_M}) \\
\text{in } & s(\!| \, cgr := cgr', dgr := dgr', preqs := preqs' \, |\!)
\end{aligned} \tag{5.25}
$$

Also note the different update of the *dgr* field: in the case of pipelining, the master owning the bus during the data phase is the master that owned the bus for the preceding address phase.

5.3 Bus Slaves

The task of a slave is to react to read or write transactions by accessing a local memory system. A slave provides two input interfaces and an output interface to connect to the communication bus: one input interface for the control bus, one input interface for the write part of the data bus, and an output interface to provide data for the read part of the data bus.

$$I_S = (\!| \, bi_{rdata} : \mathbb{B}, di_{ctrl} : ((\!| \, trans : \mathbb{B} \, |\!)) \, \tilde{\cup} \, \mathcal{C}_{bus}) \, option, di_{wdata} : \mathcal{D} \, option \, |\!) \tag{5.26}$$

$$O_S = (\!| \, bo_{ctrl} : \mathbb{B}, bo_{wdata} : \mathbb{B}, do_{rdata} : \mathcal{D} \, option \, |\!) \tag{5.27}$$

Given an address $addr \in \mathbb{N} \times \mathbb{N} \tilde{\in} \mathcal{C}_{bus}$, the first element of the pair is used to address the slave in the communication system and the second element is used to access the local memory. For simplicity it is assumed that each slave has a globally unique address, and that every slave checks locally whether a given address refers to a location private to that slave. Thus, for each slave $s \in [1, N_S]$, an *active* flag, $act_s \in \mathbb{B}$, indicates whether slave s is the currently addressed slave.

$$act_s(i) = (\mathbf{fst} \ (\mathbf{the} \ i.di_{ctrl}.addr) = s) \tag{5.28}$$

Note that this definition relies on $di_{ctrl}.addr$ always being defined, that is not being equal to **None**. This is merely for readability, as the definition can easily being extended to cover the **None** case.

In case a slave is addressed at the beginning of an address phase with a non-idle transfer, it has to sample the control data and access the memory system using this control data during the successive data phase. During the memory access,

the model accounts for possible memory delay: an (internal) busy signal from the memory to the slave is used to stall the slave in this case. The delay is given by an uninterpreted *memory-delay function* $\mathbf{md} : \mathbb{N} \to \mathbb{N}$ where $\mathbf{md}(t)$ is the delay of a memory request started in cycle t. This model does not take into account a possible dependency of the delay on the access type or varying memory access times on different slaves, but it is straightforward to extend the model. For liveness verification purposes, this function has to be bound for all accesses.

Assumption 5.5 (Slave Memory Access Time Bound)
For all times $t \in \mathbb{N}$, any access to a slave's memory system has to be completed in at most md_{max} cycles.

$$\forall\ t.\ \mathbf{md}(t) \leq md_{max}$$

The local memory of a slave is simply a mapping from addresses to data elements: $mem : \mathbb{N} \to \mathcal{D}$. The above prose description is formalised using a simple Mealy machine which can be specified ad-hoc. The control automaton without state holding elements is depicted in Figure 5.3. The state space and initial state of a slave is given by:

$$\begin{aligned} S_S = (\!|\, & state : \{idle, mreq\}, swr : \mathbb{B}, saddr : \mathbb{N}, sdata : \mathcal{D}\ option, \\ & mem : \mathbb{N} \to \mathcal{D}, mbusy : \mathbb{B}\ |\!) \end{aligned} \tag{5.29}$$

$$\begin{aligned} s0_S = (\!|\, & state = idle, swr = \mathbf{F}, saddr = 0, sdata = \mathbf{None}, \\ & mem = mem_0, mbusy = \mathbf{F}\ |\!) \end{aligned} \tag{5.30}$$

Where mem_0 is an arbitrary but fixed initial memory state.

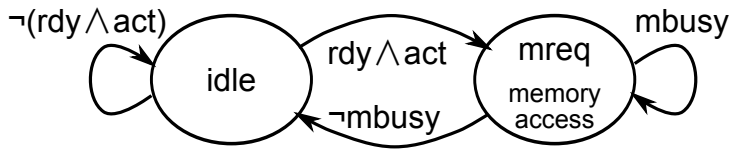


Figure 5.3: Simple Control Automaton of a Bus Slave

A slave for the pipelined communication system differs only marginally from a slave for the sequential communication system. There is no specific model for communication system with burst support because the slave does not need to change when a master is extended with burst transfers.

The difference between the pipelined and the sequential slave is only the generation of the bus ready signal: whereas the sequential slave needs to generate a bus ready at the end of both phases, a pipelined slave only outputs a ready signal at the end of the data phase. Thus, for both the step function is given by:

$$\delta_S = \lambda\ s, i.\ \text{let}$$

$$state' = \begin{cases} mreq & : s.idle \wedge \neg i.bi_{rdata} \wedge act_s(i) \\ idle & : s.mreq \wedge \neg mbusy \\ s.state & : \text{otherwise} \end{cases}$$

$$swr' = \begin{cases} (\mathbf{the} i.di_{ctrl}).wr & : \neg i.bo_{rdata} \wedge act_s(i) \\ s.swr & : \text{otherwise} \end{cases}$$

$$saddr' = \begin{cases} snd(\mathbf{the} i.di_{ctrl}).addr & : \neg i.bo_{rdata} \wedge act_s(i) \\ s.saddr & : \text{otherwise} \end{cases}$$

$$sdata' = \begin{cases} i.di_{wdata} & : s.mreq \wedge s.swr \\ s.sdata & : \text{otherwise} \end{cases}$$

$$mem' = \begin{cases} s.mem(s.saddr := \mathbf{the} s.sdata) & : s.mreq \wedge \neg mbusy \\ s.mem & : \text{otherwise} \end{cases}$$

$$\text{in } (\!| state = state', swr = swr', saddr = saddr', mem = mem' |\!) \tag{5.31}$$

where $f(x := y)$ denotes the update of a function or mapping $f : \mathcal{X} \to \mathcal{Y}$ at position $x \in \mathcal{X}$ with $y \in \mathcal{Y}$. The semantics of the busy input is the standard bus ready interpretation:

$$rdy = \neg i.bi_{rdata} \tag{5.32}$$

As mentioned before, the output function depends on pipelining support. A sequential slave generates a ready signal at the end of the address phase (via bo_{ctrl}) and at the end of the data phase (via bo_{wdata}). Both slaves output data delivered from the local memory system for read transactions.

$$\begin{aligned} \omega_{S,seq} = & \lambda\ s, i. \\ & ardy = s.idle \wedge (\delta_S(s,i).state = mreq) \\ & drdy = s.mreq \wedge \neg mbusy \\ & rdata = \begin{cases} \mathbf{Some}\ mem(s.saddr) & : \neg s.swr \wedge s.mreq \wedge \neg mbusy \\ \mathbf{None} & : \text{otherwise} \end{cases} \\ & \text{in } (\!| bo_{ctrl} = ardy, bo_{wdata} = drdy, do_{rdata} = rdata |\!) \end{aligned} \tag{5.33}$$

With the definition of the bus ready signal from Equation 5.7, this generates bus ready signals at the end of each phase. In the sequential system, the address

phase last therefore exactly one cycle.

To obtain a slave for a pipelined communication system, only the bo_{ctrl} output has to be strengthened such that a slave never generates a ready signal at the end of the address phase. This can easily be achieved by forcing bo_{ctrl} to $\mathbf{T}$:

$$\omega_{S,pip} = \lambda\ s, i.\ (\omega_{S,seq}(s, i))(\!|\, bo_{ctrl} := \mathbf{T} \,|\!) \tag{5.34}$$

Note that this is also a trivial application of the data modification component of the framework.

5.4 Basic Sequential Master

The sequential communication system represents the basic system which is then extended with pipelining and burst transfers. The term sequential refers to the way transactions are put or executed on the communication bus: a sequential master executes one, single, full transaction after the other, as already depicted in Figure 5.2. The following sections focus on the incremental construction of the master needed for the different communications systems as the master is the most complex component.

Abstract Sequential Transfers

Before detailing the components of the basic, sequential communication system, the concept of abstract transfers is introduced: *abstract transfers* abstract an actual transaction consisting of an address and data phase to a set of time points—grant time, address phase time, and data phase time—together with the index of the master to which the transfer belongs and a flag indicating an idle transfer.

Abstract transfers simplify the following formalisation in two ways: they are used to formulate key protocol characteristics concisely, and they ease the statement of the main correctness theorems of the modelling steps as simulation relations. The convenience of abstract transfers will become evident in the rest of this chapter.

Using the fact that each phase of a transfer is completed by an active bus ready signal ($rdy = \mathbf{T}$), and abstract sequential transfer is defined as in Definition 5.6.

Definition 5.6 (Abstract Sequential Transfer)

The i-th abstract sequential transfer $str(i)$ is defined in terms of a grant value $gnt \in [1 : N_M]$, a single bit $isdata \in \mathbb{B}$ indicating a idle or data transfer, and three cycle-accurate time points:

- $tg \in \mathbb{N}$ *is the time when the bus is granted to the master gnt,*
- $ta \in \mathbb{N}$ *is the time when the address phase ends, and*
- $td \in \mathbb{N}$ *denotes the time when the data phase of transfer i ends.*

Thus the i-th abstract sequential transfer is the quintuple

$$str(i) = (gnt, isdata, tg, ta, td) \in [1 : N_M] \times \mathbb{B} \times \mathbb{N}^3$$

where the components are defined as

$$gnt = arb.grant^{tg}$$

$$isdata = \begin{cases} 0 & : idle\ transfer \\ 1 & : otherwise \end{cases}$$

$$tg = \begin{cases} 0 & : i = 0 \\ str(i-1).td & : otherwise \end{cases}$$

$$ta = \min\{t > tg \mid rdy^t\}$$

$$td = \min\{t > ta \mid rdy^t\}$$

arb.grant denotes the arbiter configuration component specifying the currently granted master.

From the abstract transfer definition, we can summarise three key protocol characteristics for this core design step:

1. every transfer consists of an address and a data phase,
2. the end of each phase is defined by the bus signal rdy, and
3. the bus is granted to some master at every time instant.

These characteristics and the definition of an abstract sequential transfer are illustrated in Fig. 5.4. To ease notation, we define a function $i(u,t)$. It denotes the next transfer such that tg is greater or equal to t and the bus is granted to master u.

$$i(u,t) = \min\{j \mid t \leq str(j).tg \wedge u = str(j).gnt\} \tag{5.35}$$

In case no such minimum is defined, we say $i(u,t) = -\infty$.

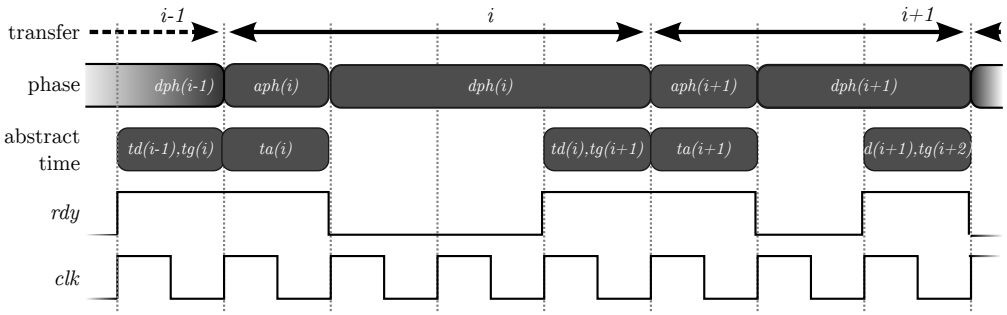


Figure 5.4: Abstract Sequential Transfers

Sequential Master

The master provides the interface between the communication system and an attached host system. It handles host requests to transfer data. Intuitively, a AHB master works as follows: if the master is not busy with a transaction, the host can request a transaction by providing the control data and possibly the write data to the master. The master starts the transfer by checking if it is already granted the bus and, in case it is not, by requesting bus access from the arbiter. Once the bus is granted and a new bus transaction is about to start, the master outputs the control data of the transfer to the bus. During the data phase of the transaction, the master either provides the write data on the data bus, or it reads the read data from the data bus at the end of the phase. Note that a host system is not allowed to be busy, so received data can be directly forwarded to the host system. The sequential master is the main component in the basic communication system.

The send part of the sequential master is constructed by a sequential composition of a simple send buffer and a data modification component to implement the control. The receive part is only depicted in Figure 5.5 since it only forwards bus data right away to the host, i. e. it is just a wire, while providing

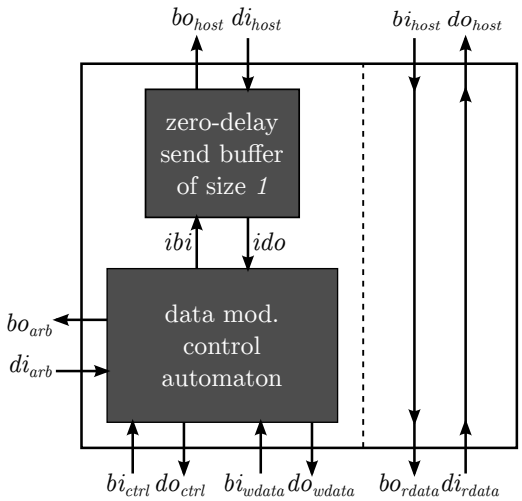


Figure 5.5: Schematics of the Sequential Master

the following two interfaces:

$$I_{MSeq,r} = (\!| \, bi_{host} : \mathbb{B}, di_{rdata} : \mathcal{D}\ option \, |\!) \tag{5.36}$$

$$O_{MSeq,r} = (\!| \, bo_{rdata} : \mathbb{B}, do_{host} : \mathcal{D}\ option \, |\!) \tag{5.37}$$

In the following, the construction of the sender part is detailed. The sender provides two output interfaces to the communication bus: for control data ($ctrl$) and for write data ($wdata$). It also provides two input interfaces: one form the host ($host$) and one from the arbiter (arb). Let $\mathcal{C} = (\!| \, wr \in \mathbb{B}, addr \in \mathbb{N} \times \mathbb{N} \, |\!)$, then

$$\begin{aligned} I_{MSeq,s} = (\!| \, & bi_{ctrl} : \mathbb{B}, bi_{wdata} : \mathbb{B}, di_{arb} : \mathbb{B}\ option, \\ & di_{host} : (\!| \, hctrl \in \mathcal{C}, hwdata \in \mathcal{D}\ option \, |\!)\ option \, |\!) \end{aligned} \tag{5.38}$$

$$\begin{aligned} O_{MSeq,s} = (\!| \, & bo_{arb} : \mathbb{B}, bo_{host} : \mathbb{B}, \\ & do_{ctrl} : ((\!| \, trans \in \mathbb{B} \, |\!) \, \tilde{\cup} \, \mathcal{C})\ option,\ do_{wdata} : \mathcal{D}\ option \, |\!) \end{aligned} \tag{5.39}$$

The first part of the sender is the send buffer. As mentioned above, the bus controller serves one host request at the time, i. e. while processing a read or write transaction the bus controller signals the host that it is busy by activating the host busy output bo_{host}. Additionally, the buffer has to 'notify' the control automaton when the host requests a data transmission. A simple zero-delay buffer of size 1 provides all these properties. Also recall that the buffer provides Moore-like output signals.

Definition 5.7 (Send Buffer of the Sequential Master)

The sequential master uses a zero-delay buffer of size 1 *as a send buffer.*

$$SB_{Mseq} = (\mathcal{C} \times \mathcal{D}\ option)\ zbuf \tag{5.40}$$

For the second part of the sender, an instantiation of the data modification component is used to model a control automaton for the bus access, denoted M_c. The instantiation has to provide the inputs and outputs from Figure 5.5:

$$\begin{aligned} I_c = (\!|\, & bi_{ctrl} : \mathbb{B}, bi_{wdata} : \mathbb{B}, && (5.41)\\ & di_{arb} : \mathbb{B}\ option, ido : (\!|\, hctrl : \mathcal{C}, hwdata : \mathcal{D}\ option \,|\!)\ option \,|\!) \\ O_c = (\!|\, & bo_{arb} : \mathbb{B}, ibi : \mathbb{B}, && (5.42)\\ & do_{ctrl} : ((\!|\, trans : \mathbb{B} \,|\!) \tilde{\cup} \mathcal{C})\ option, do_{wdata} : \mathcal{D}\ option \,|\!) \end{aligned}$$

In case the host requests a data transfer, the control automaton has to perform the following steps: in case the bus is not already granted to the master, it has to request bus access from the arbiter. Once the bus is granted to the master and the next address phase starts—indicated by an active *rdy* signal—it has to

output the control data during the address phase. During the data phase, it has to provide the data to be written or has to sample read data at the end of the data phase. Additionally, the master has to perform an idle transaction in case it is granted the bus but there is no data to transmit. The automaton to be implemented using the OPT part of the data modification component is depicted in Figure 5.6. The following auxiliary signals are used to define the automaton:

$$startreq^t = (i^t.ido = \mathbf{Some}\, x) \quad \text{for some } x \in (\!|\, hctrl : \mathcal{C}, hwdata : \mathcal{D}\, option \,|\!) \tag{5.43}$$

$$grant^t = (i^t.di_{arb} = \mathbf{Some}\, \mathbf{T}) \tag{5.44}$$

$$rdy^t = \neg i.bi_{ctrl} \tag{5.45}$$

The automaton has three states: $idle$, address phase (aph), and data phase (dph). Additionally, a flag $itrans$ indicates that an idle transfer is needed. Thus, the state space and initial state of OPT_c are given by:

$$Opt_c = (\!|\, state : \{idle, aph, dph\}, itrans : \mathbb{B} \,|\!) \tag{5.46}$$

$$opt_c^0 = (\!|\, state = idle, itrans = \mathbf{F} \,|\!) \tag{5.47}$$

Given an input assignment $i \in I_c$, the automaton state $opt.state$ is updated

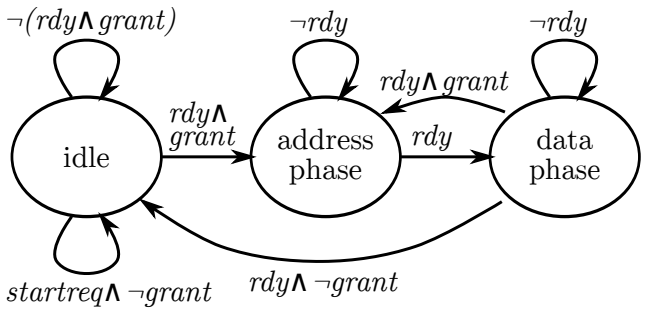


Figure 5.6: Control Automaton for the Sequential Master

according to the following definition:

$$state' = \begin{cases} idle & : (opt.state = dph) \wedge i.rdy \wedge \neg i.grant \\ aph & : (opt.state = idle) \wedge i.rdy \wedge i.grant \vee \\ & \quad (opt.state = dph) \wedge i.rdy \wedge \neg i.grant \\ dph & : (opt.state = aph) \wedge i.rdy \\ opt.state & : \text{otherwise} \end{cases} \tag{5.48}$$

The *itrans* flag indicates that an idle transfer is in progress. Therefore, it has to be set to ⊤ in case a master has to start a transaction, but there is no data to be send. A master has to start transmitting if the *rdy* and the *grant* inputs are active while the control automaton is in the idle state. Once the flag is set, it has to be unset at the end of the idle transfer in case there is no consecutive idle transfer afterwards. A transaction finishes with an active ready signal in the *dph* state. The flag is unset if the bus is not granted any more, or the bus is granted

but there is now data to be transmit.

$$itrans' = \begin{cases} \mathbf{T} & : (opt.state = idle) \wedge i.rdy \wedge i.grant \wedge \\ & \quad (ido = \mathbf{None}) \\ \mathbf{F} & : (opt.state = dph) \wedge itrans \wedge \\ & \quad (\neg i.grant \vee i.grant \wedge (ido \neq \mathbf{None})) \\ opt.itrans & : \text{otherwise} \end{cases} \tag{5.49}$$

Using Equations 5.48 and 5.49, $\delta_{opt,c}$ is defined by:

$$\delta_{opt,c} = \lambda\ opt.\ i.\ (\!|\, state = state', itrans = itrans' \,|\!) \tag{5.50}$$

To complete the specification of M_c, all the outputs have to be defined and the data modification needs to be instantiated. M_c has to output control data when the state machine is in the address phase state and possibly write data while in the data phase. The data output on the control bus is extended with the information about whether the transfer is an idle transfer or an actual data transfer; the *trans* component of the control bus (cf. Definition 5.1).

$$dctrl\ (opt, ido) = \begin{cases} \mathbf{Some}\ (\!|\, trans = \mathbf{T} \,|\!) \\ \quad \tilde{\cup}\ (\mathbf{the}\, ido).ctrl) & : (opt.state = aph) \\ & \quad \wedge \neg opt.itrans \\ \mathbf{Some}\ idlectrl & : (opt.state = aph) \\ & \quad \wedge\ opt.itrans \\ \mathbf{None} & : \text{otherwise} \end{cases} \tag{5.51}$$

$$dwdata\ (opt, ido) = \begin{cases} \textbf{Some}\ (\textbf{the}\ ido).wdata & : (opt.state = dph) \\ & \quad \wedge \neg opt.itrans \\ \textbf{Some}\ idlewdata & : (opt.state = dph) \\ & \quad \wedge\ opt.itrans \\ \textbf{None} & : \text{otherwise} \end{cases} \tag{5.52}$$

The busy signal *ibi* going from M_c to the send buffer is used to keep data in the send buffer until it has been send completely. Therefore, the busy signal is deactivated only at the end of the data phase. The remaining output is the busy output to the arbiter: it is used to request the bus from the arbiter.

$$sbb\ opt = \neg(opt.rdy \wedge (opt.state = dph) \wedge \neg opt.itrans) \tag{5.53}$$

$$arb\ i = startreq \wedge \neg grant \tag{5.54}$$

Definition 5.8 (Control Automaton for Sequential Master)
The control automaton for the sequential master M_c is defined as the data modification $DM(mathitOPT_c, f_c, g_c)$ with the following components:

- $OPT_c = (Opt_c, opt_c^0, \delta_{opt,c})$ *as defined in Equations 5.46, 5.47, and 5.50.*
- $f = \lambda\, s.\, di.\, (\!|\, do_{ctrl} = dctrl\ (s.opt, di.ido), do_{wdata} = dwdata\ (s.opt, di.ido)\,|\!)$
- $g = \lambda\, s.\, i.\, (\!|\, ibi = (sbb\ s.opt), bo_{arb} = (arb\ i)\,|\!)$

The complete send part of the sequential master is given by the sequential composition of M_{MSeq} and M_c. The full sequential master is the parallel composition of the send and receive part.

Definition 5.9 (Sequential Master)

The sequential master is defined as

$$M_{seq} = (SB_{MSeq};;M_c)||M_{rcv} \tag{5.55}$$

5.5 Pipelined Master

Previous chapters already discussed that pipelining refers to parallelizing address and data phases on the communication bus, taking advantage of physically separate buses for control information and data. The idea is to execute the address phase of a transfer in parallel with the data phase of the previous transfer, relative to the interconnect. The length of each phase is then only defined by the length of the data phases of each transfer.

In the following, the notion of an abstract transfer is applied to the pipelined system. The pipelining transformation is detailed afterwards and, at the end of this section, applied to the sequential master from the previous section.

Abstract Pipelined Transfer

The definition of an abstract pipelined transfer is almost equivalent to the definition for the sequential system. The key property of pipelining, the parallelized phases, is represented by *moving* the grant time of a transfer i from the end of the data phase of transfer $i-1$ to the end of the address phase of that transfer. In the model of abstract transfers, this is the only chance required to represent pipelining. Fig. 5.7 shows a sequence of pipelined transfers as an example.

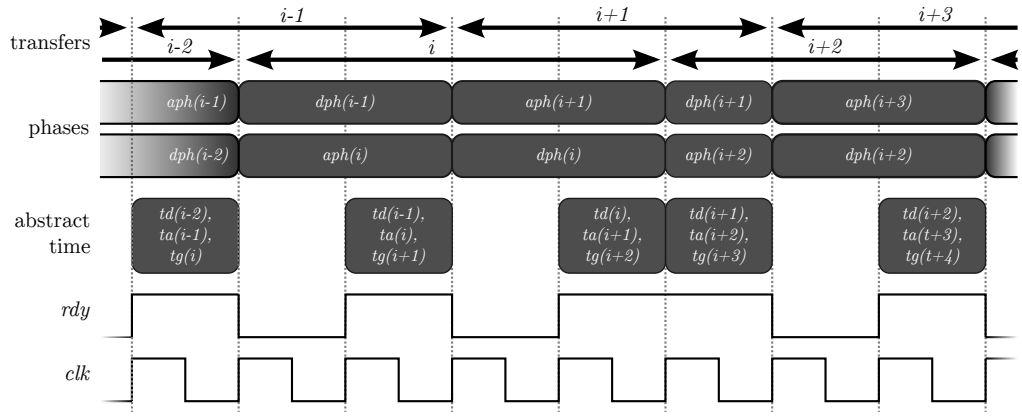


Figure 5.7: Sequence of Pipelined Transfers

Definition 5.10 (Abstract Pipelined Transfer)

The i-th abstract pipelined transfer $ptr(i)$ *is given by the tuple* $(gnt, isdata, tg, ta, td)$ *where the components are defined as:*

$$tg = \begin{cases} 0 & : i = 0 \\ ptr.ta & : otherwise \end{cases}$$

$$x = tr_{seq}(i).x \quad for\ x \in \{gnt, isdata, ta, td\}$$

5.5.1 Pipelining Transformation

Pipelining of transfers on the bus means parallelizing the execution of address and data phases of two consecutive transfers. Naturally, the transformation is therefore based on the duplication of sequential masters using the multiplex/arbitrate composition. Two observations are worth pointing out at this point:

- The replication operator, although preferable over the multiplex/arbitrate composition in general, cannot be applied here because the arbitration

relation has to output data from more than a single inner component at the same time.

- If bus ownership changed after every transfer, for example if the arbiter implemented a strict round-robin policy, there would not be any need to transform a master. The master only has to be transformed to support the *local* execution of a address and data phase simultaneously, which is only the case if a master owns the bus for at least two consecutive transfers.

The core idea behind the transformation is to execute two copies of the sequential master in parallel, and use the multiplex and arbitrate components to constrain allowed executions with respect to a purely interleaved one. The key question is which are the necessary constraints and how to realise them. Considering the Mealy machine for the sequential master from Figure 5.6 and Definition 5.9, the following intuitive constraints can easily be seen:

- There must not be any *phase contention*, thus the two control automatons cannot both be in the *aphase* state or the *dphase* state.

- If both automata are in a non-idle state, then either both or none of them have to change state. So both automatons must end phases at the same time.

- An (artificial) rule has to resolve the symmetry in the construction and to to specify the master that initiates the first transfer is a sequence of consecutive pipelined transfers. Examples for such rules are: the *first* master, i. e. the master with the lowest index, starts, or transfers are partitioned into even and odd transfers between the two masters (with respect to the abstract transfer index).

Recall from Definition 4.7 that the multiplex/arbitrate operator is parametrised in three parameters: a multiplex relation mux, an arbitration function arb, and an optional component OPT, which includes state space Opt, initial state opt^0, and update function δ_{opt}. The instantiated operator is then applied to a set of two sequential masters, the send parts to be precise, to obtain the pipelined master.

$$M_{pipe,snd} = \diamond\{M_{1,seq,snd}, M_{2,seq,snd}\} \tag{5.56}$$

In the following, the instantiation is specified using the signal names according to the AHB specification, i. e. $startreq$, $grant$, rdy, instead of the framework signals only based on busy and data signals. This improves readability significantly and the formal mapping from busy-data-interfaces to AHB signals has been established in the previous sections. Also, let M_i be a shorthand for $M_{i,seq,snd}$ ($i \in \{1, 2\}$).

To resolve the symmetry issue, this transformation always uses M_2 to initiate the first transfer in a series of transfers, which means that M_2 is exclusively used for pipelining. To achieve this, the mux and arb components have to ensure that

- any data from the device core is only passed to the second master if at the end of a bus address phase, the first master is locally in an address phase and the master has still ownership of the bus, and
- the second master cannot request the bus from the arbiter as it might result in lost requests.

First, we detail the instantiation of the arbitration functions. It has to ensure that an active ready signal is always passed to M_1, but to M_1 only if M_0 is in the

address phase. M_0 is in the address phase if, and only if, it outputs control data.

$$rdy_1^t = i^t.rdy \tag{5.57}$$

$$rdy_2^t = i^t.rdy \wedge (ido_1^t.do_{ctrl} \neq \textbf{None}) \tag{5.58}$$

For the data outputs, the arbitrate functions simply forwards any data the sequential master output and relies on the fact that there is no phase contention.

$$do_{ctrl}^t = \begin{cases} ido_2^t.do_{ctrl} & : (ido_2^t.do_{ctrl} \neq \textbf{None}) \\ ido_1^t.do_{ctrl} & : \text{otherwise} \end{cases} \tag{5.59}$$

$$do_{wdata}^t = \begin{cases} ido_2^t.do_{wdata} & : (ido_2^t.do_{wdata} \neq \textbf{None}) \\ ido_1^t.do_{wdata} & : \text{otherwise} \end{cases} \tag{5.60}$$

Equations 5.57–5.60 specify the arbitration component arb_{pipe} since the arbitration does not rely on any OPT component.

The multiplex function has to ensure that only M_1 requests the bus from the arbiter. This is achieved by passing any data from the device core to the M_1 unless M_1 is in the address phase at time ta (end point of address phase) and the grant signal is still active. This is also the only time, the M_2 busy signal is forwarded to the host. In order to realise this with the multiplex component, the current control automaton state of M_0 is required. As the multiplex component has to access to the local configuration of M_0, the OPT component, OPT_{pipe} is used to provide this information.

$$Opt = (\!| \, m0state : \{idle, aph, dph\} \, |\!) \tag{5.61}$$

$$opt^0 = (\!| \, state = idle \, |\!) \tag{5.62}$$

$$\delta_{opt} = \lambda s, i, o.\ \delta_0(s, i).M_c.state \tag{5.63}$$

Using this OPT component, the multiplex function mux_{pipe} can be specified according to the above prose.

$$idi_1^t.di_{host} = i^t.di_{host} \tag{5.64}$$

$$idi_1^t.di_{arb} = i^t.di_{arb} \tag{5.65}$$

$$idi_2^t.di_{host} = \begin{cases} i^t.di_{host} & : (opt^t.m0state = aph) \wedge grant^t \\ \textbf{None} & : \text{otherwise} \end{cases} \tag{5.66}$$

$$idi_2^t.di_{arb} = \begin{cases} i^t.di_{arb} & : (opt^t.m0state = aph) \wedge grant^t \\ \textbf{None} & : \text{otherwise} \end{cases} \tag{5.67}$$

Definition 5.11 (Pipelining Transformation)

The pipelining transformation for a AHB sequential master M_{seq} is given by an instance of the arbitrate/multiplex composition operator.

$$PipeTr = \diamond[arb_{pipe}, mux_{pipe}, OPT_{pipe}] : Mealy\ set \to Mealy$$

The transformation is applicable to a set of two copies of the sequential Master's send part.

Definition 5.12 (Pipelined Master)

Using the transformation from Definition 5.11 and let $M_{snd} = (SB_{MSeq};;Mc)$,

then a pipelined master for AMBA AHB is given by:

$$M_{pipe} = PipeTr\{M_{snd}, M_{snd}\}||M_{rcv}$$

5.6 Master with Burst Transfer Support

Burst transfers is a feature for the host system, which is connected to a bus controller, to initiate a sequence of transfers with consecutive addresses that is not interrupted. If the host system initiates such a burst transfer, it has to provide a *burst size* together an address as well. For example, a burst transfer of size s to address a accesses s-many data elements from address a to $a - s - 1$. This model supports burst transfers of arbitrary, but fixed length.

The transformation for burst transfers is general enough to be independent from pipelining. Thus the transformation is applicable to both masters, sequential and pipelined. In the very few instances in which the transformation has to be specific about the master, the flags $isseq$ and $ispipe$ are used to indicate the respective masters.

Before detailing the transformation, the definition of an abstract transfer is adapted to burst transfers.

Abstract Burst Transfer

The necessary changes to the abstract transfer definitions for the previous masters are more significant than before. To keep track of burst transfers, a new component $bs \in \mathbb{N}$ specifies the burst size of a transfer. In case the transfer is not a burst transfer, bs is set to 0.

In case of a burst transfer, the end of the address phase is not a unique time point anymore, but a partial function mapping address phase end times to *sub-transfer*, a single transfer with a burst transfer. Figure 5.8 depicts an illustrating example for the sequential case.

Definition 5.13 (Abstract Burst Transfer)

The i-th abstract burst transfer $btr(i)$ is defined as the tuple

$$(gnt, isdata, tg, ta, td, bs) \in [1 : N_M] \times \mathbb{B} \times \mathbb{N} \times (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N}) \times \mathbb{N}$$

where the individual components are:

$$gnt = \{s, p\}tr(i).gnt$$

$$isdata = \{s, p\}tr(i).isdata$$

$$tg = \begin{cases} 0 & : i = 0 \\ btr(i-1).ta(btr(i-1).bs) & : i > 0 \wedge ispipe \\ btr(i-1).td & : i > 0 \wedge isseq \end{cases}$$

$$ta(n) = \begin{cases} \min\{t > tg \mid rdy^t\} & : (bs = 0) \vee n = 0 \\ \min\{t > ta(n-1) \mid rdy^t\} & : 0 < n < bs \wedge ispipe \\ \min\{t > td(n-1) \mid rdy^t\} & : 0 < n < bs \wedge isseq \\ undefined & : otherwise \end{cases}$$

$$td(n) = \begin{cases} \min\{t > ta(0) \mid rdy^t\} & : (bs = 0) \\ \min\{t > ta(n) \mid rdy^t\} & : 0 < n < bs \\ undefined & : otherwise \end{cases}$$

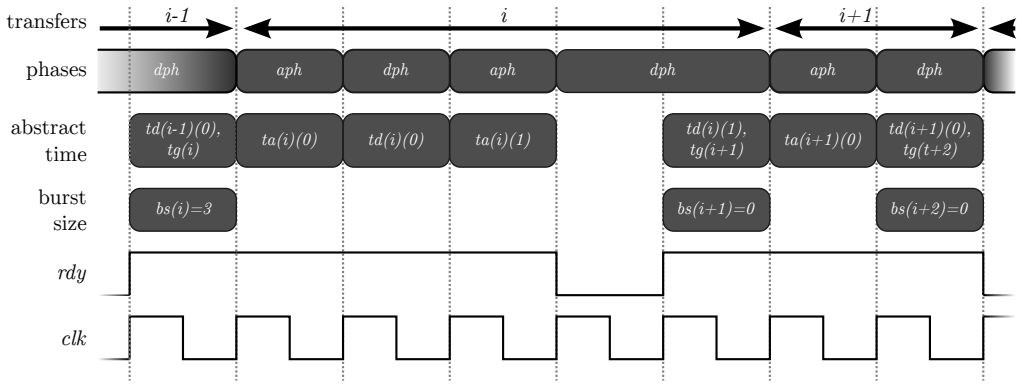


Figure 5.8: Sample Burst Transfers

$$bs = bm_{gnt}^{ta(0)}.bs$$

The main difference between this definition and the previous one is the case split on actual burst transfers. For a non-burst transfer the above definition resolves to one of the previous transfer definitions depending on the system we are extending.

5.6.1 Transformation for Burst Transfers

Similar to the pipelining transformation, the burst transformation is a transformation for the send part of the master only. It instantiates the data modification component from the framework, which is then combined with an existing send part using sequential composition. The intuition behind this transformation is the following: the transformation has to adapt the interfaces between master and host, and master and arbiter such that

- if the host requests a burst transfer, the *old* master is used to generate a sequence of non-burst transfers on the bus which are observationally equivalent to the burst transfer (from the host),

- during a burst transfer, the data modification component stalls the host by strengthening the busy signal except for new write data,
- to maintain bus ownership for the duration of the burst transfer, the arbiter has to now about the burst transfer and its size.

The host and arbiter interfaces of the sender part of a sequential or pipelined master are (cf. Equations 5.38 and 5.39):

$$req_M = (bi_{arb} \in \mathbb{B}, do_{arb} \in \mathbb{B}\ option) \tag{5.68}$$

$$gnt_M = (bo_{arb} \in \mathbb{B}, di_{arb} \in \mathbb{B}\ option) \tag{5.69}$$

$$host_M = (bo_{host} \in \mathbb{B}, di_{host} \in (\!|\, hctrl : \mathcal{C}, hwdata : \mathcal{D}\ option \,|\!)\ option) \tag{5.70}$$

where the host control data is given by:

$$\mathcal{C} = (\!|\, wr : \mathbb{B}, addr : \mathbb{N} \times \mathbb{N} \,|\!) \tag{5.71}$$

The interface to the host is modified by extending the control data with a field for the burst size.

$$\mathcal{C}_{bs} = \mathcal{C} \tilde{\circ} (\!|\, bs : \mathbb{N} \,|\!) \tag{5.72}$$

$$host_{M,bs} = (bo_{host} \in \mathbb{B}, di_{host} \tilde{\in} (\!|\, hctrl : \mathcal{C}_{bs}, hwdata : \mathcal{D}\ option \,|\!)\ option) \tag{5.73}$$

Similarly, the interface to the arbiter is extended with the size of the bust access. The arbiter uses this information to count the number of individual transactions

for which the bus ownership cannot change.

$$req_{M,bs} = (bi_{arb} \in \mathbb{B}, do_{arb} \in \mathbb{N}\ option) \tag{5.74}$$

Recall from Definition 4.3 that the data modification need to be instantiated by providing two functions: a data modification function

$$f : S \times \mathcal{DI}^n \to \mathcal{DO}^m$$

and a busy signal strengthening function

$$g : S \times I \to \mathcal{BO}^m.$$

The component also allows for an optional component

$$OPT = (Opt, opt^0, delta_{opt})$$

which is used to store control data and the address counter needed to generate a series of individual transfers.

First the OPT component needs to be instantiates so that it can store control data for a transfer and a counter values for address computation.

$$Opt_{bs} = (\!|\, bwr : \mathbb{B}, baddr : \mathbb{N} \times \mathbb{N}, bs : \mathbb{N}, caddr : \mathbb{N} \,|\!) \tag{5.75}$$

$$opt^0_{bs} = (\!|\, bwr = \mathbf{F}, baddr = (defS, 0), bs = 0, naddr = 0 \,|\!) \tag{5.76}$$

The update function δ_{opt} is also straightforward, but tedious because there are many different case to cover: if the bus controller is idle and the host initiates a

non-burst transfer, nothing changes as these requests are simply forwarded to the old sender part; however if that transfer is a burst transfer, then the first transfer is immediately forwarded, but all the control data is also stored, and the next address field *naddr* is set to the *base address*, the first address, plus one. Then, each time the old send part indicates the start of a new transfer (inactive busy), the data modification composes control data using *naddr* as address, increments *naddr*, and in case of a write transfer new data from the host. The burst transfer ends if *naddr* is equal to $baddr + bs$.

$$\delta_{opt,bs} = \lambda\ opt, i.\ \text{let}$$

$$(bwr', baddr', bs') = \begin{cases} \mathbf{the} i.di_{host}.hctrl & : v(i.di_{host}) \wedge \\ & \quad \mathbf{the}\ i.di_{host}.hctrl.bs > 0 \\ (\mathbf{F}, (defS, 0), 0) & : bdone(opt) \\ (opt.bwr, opt.baddr, opt.bs) & : \text{otherwise} \end{cases}$$

$$naddr' = \begin{cases} baddr' + 1 & : v(i.di_{host}) \\ naddr' + 1 & : \neg i.bi_M \wedge (bs' \neq 0) \\ 0 & : bs' = 0 \\ naddr' & : \text{otherwise} \end{cases}$$

$$\text{in } (\!| bwr = bwr', baddr = baddr', bs = bs', naddr = naddr' |\!) \tag{5.77}$$

where the predicate $bdone(opt)$ indicates that the last address phase of the burst transfer has just pasted ($ta(bs-1)$).

$$bdone(opt) = (opt.baddr + opt.bs = opt.naddr)$$

The busy strengthening function has to ensure to keep the busy signal active during burst read transfers. During burst write transfers, the host provides a new chunk of write data every time a new sub-transfer is generated, so the busy signal has to be lowered and can simply be forwarded from the old master. The busy signal to the arbiter is unmodified.

$$g_{bs} = \lambda\ s, i.\ (\!|bo_{host} = (i.bo_{host} \vee (s.opt.bs > 0 \wedge \neg s.opt.bwr)),$$
$$bo_{arb} = i.bo_{M,arb}\,|\!) \tag{5.78}$$

Lastly, the actual data modification function f has to be instantiated. The generation of sub-transfers has already been elaborated above. The missing modification to complete the instantiation of f is the modification of the arbiter output interface: in case the sequential or pipelined master requests the bus from the arbiter, the Boolean value of the former do_{arb} signal is replace with a corresponding natural number: 0 in case of a standard transfer, and bs is case of a burst transfer.

$$f_{bs} = \lambda\ s, i.\ \text{let}$$
$$arb_o = \begin{cases} \textbf{Some} s.opt.bs & : i.do_{arb} \\ \textbf{None} & : \text{otherwise} \end{cases}$$
$$hctrl_o = \begin{cases} (\!|\, wr = (\textbf{the}\ i.di_{host}).wr, & \\ \quad addr = (\textbf{the}\ i.di_{host}).addr\,|\!) & : v(i.di_{host}) \wedge s.opt.bs = 0 \\ (\!|\, wr = s.opt.bwr, addr = s.opt.naddr\,|\!) & : s.opt.bs > 0 \\ (\!|\, wr = \mathbf{F}, addr = 0\,|\!) & : \text{otherwise} \end{cases}$$

$$hwdata_o = \begin{cases} (\textbf{the}\ i.di_{host}).hwdata & : v(i.di_{host}) \\ \textbf{None} & : \text{otherwise} \end{cases}$$

$$host_o = \begin{cases} \textbf{Some}((\!|\, hctrl = hctrl_o \,|\!) \tilde{\circ} hwdata_o) & : v(i.di_{host}) \vee s.opt.bs > 0 \\ \textbf{None} & : \text{otherwise} \end{cases}$$

$$\text{in } (\!|\, do_{arb} = arb_o, do_{host,M} = host_o \,|\!) \tag{5.79}$$

Definition 5.14 summarized the construction of the transformation and Definition 5.15 specifies a master with burst support.

Definition 5.14 (Burst Support Transformation)

The transformation that adds support for burst transfers to either a sequential or a pipelined master, consists of the sequential composition of the source *Mealy machine and the data modification component as defined above.*

$$BurstTr = \lambda M.\ DM[g_{bs}, f_{bs}, OPT_{bs}];; M : Mealy \rightarrow Mealy$$

where M has to provide matching interfaces for req_M and hin_M.

Definition 5.15 (Master with Burst Support)

Given a sequential master $M_{seq} = (SB_{MSeq};; M_C)||M_{rcv}$, a sequential master with support for burst transfers is given by

$$M_{sbst} = (BurstTr(SB_{MSeq};; M_C))||M_{rcv}$$

A master with support for pipelined and burst transfers is given by

$$M_{pbst} = (BurstTr\ \ PipeTr\ \{SB_{MSeq};;M_C, SB_{MSeq};;M_C\})\,||M_{rcv}$$

5.7 Related Work

A lot of work as been done on the ARM AMBA 2 protocol, as it is a widely used, high-performance, and especially open, bus protocol protocol.

Amjad [Amj04, Amj06] has done a lot of work on AMBA and has verified latency, arbitration, coherence, and deadlock freedom properties of a simplified AMBA model. In his more recent contribution, he combines theorem proving in HOL with model checking to reduce the verification, or theorem proving, effort of showing various control and datapath properties. Amjad provides a comprehensive verification effort of the AMBA protocol, however his modelling and verification approach relies, like most existing work, on monolithic modelling and post-hoc verification.

Roychoudhury *et al.* [RMK03] also tackle the verification of the AMBA protocol, but in contrast to Amjad's work this work builds on model checking only. The authors formalise an academic version of the AMBA specification and use the SMV model checker to verify certain design invariants. As the author's verification approach is solely based on the SMV model checker, the work is based on monolithic modelling and post-hoc verification, similar to most existing work on specific protocols.

Finally, D'silva *et al.* [DRS04] use the AMBA protocol as a case study to illustrate the application of their framework for modelling on-chip bus

architectures. Their framework is also based on synchronous finite state machines, thus the basic modelling approach is comparable with the one presented in this dissertation. However, the authors focus in their work on compatibility verification, interface synthesis, and model checking with automated specifications, which distinguishes their work from the work here: our focus is on generality and support for incremental modelling and verification.

Overall, most existing work on AMBA protocol verification relies on monolithic modelling and post-hoc verification. It is also important to note that the aim of this work is not to provide another verification of a specific protocol, but a generic framework. Therefore, the presented work does also not attempt to "match" the comprehensive verification effort of previous work on the AMBA protocol, such as Amjad's or Roychoudhury's contributions.

Chapter 6

PCI Express 2.0

As a second case study, the PCI Express protocol [PS06], in particular its transaction layer, is modelled using the framework. PCI Express is a serial, high-performance, point-to-point interconnect, implementing communication between two devices using dual uni-directional paths. In contrast to a bus protocol, such as the AHB protocol from the previous chapter, a point-to-point protocol does not need any arbitration among the devices for access to the interconnect; every PCI Express controller has exactly one neighbour which is the receiver for the controller's transmit channel and the sender for the controller's receive channel.

The *PCI Express Base Specification* defines four types of basic PCI Express elements in an interconnect:

- *Root complex*: a root complex is the *head* or *root* of an interconnect hierarchy which can provide several PCI Express interfaces to attach other PCI Express elements. Every interface off the root complex constitutes an independent hierarchy domain and communication across these domains is not required by the standard.

- *Switches*: a switch is used to fan out a PCI Express hierarchy, so it is similar to a root complex in the sense that it provides more than a single PCI Express interface to attach other PCI Express elements. However, *downstream* devices attached to a switch belong to the same hierarchy domain and a switch has to manage communication among them. The sole purpose of a switch is to direct and forward transactions to the right interface.

- *Endpoints*: an endpoint is a device implementing a single PCI Express interface that can request and/or complete PCI Express transactions.

- *PCI Express to PCI bridges*: a PCI Express to PCI bridge is similar to an endpoint in the sense that it implements a single PCI Express port, but provides one or more PCI bus interfaces instead of acting as a "proper" endpoint in the system.

A schematic of a typical PCI Express interconnect network is shown in Figure 6.1. Note that this work focuses on modelling a single PCI Express port with transmit and receive part, which can be considered as focusing on modelling an endpoint. This part of the PCI Express specification implements the actual communication protocol features and is needed for all the elements listed above.

Similar to the AHB protocol, PCI Express communication is also transaction-based, but the single unit of communication that constitutes a transaction is fundamentally different: PCI Express is a packet-based protocol which means every item of communication contains a proper header field that provides all required control data for the transaction. A PCI Express transaction is initiated by a sender by transmitting a *request packet* to a receiver. Since not every request

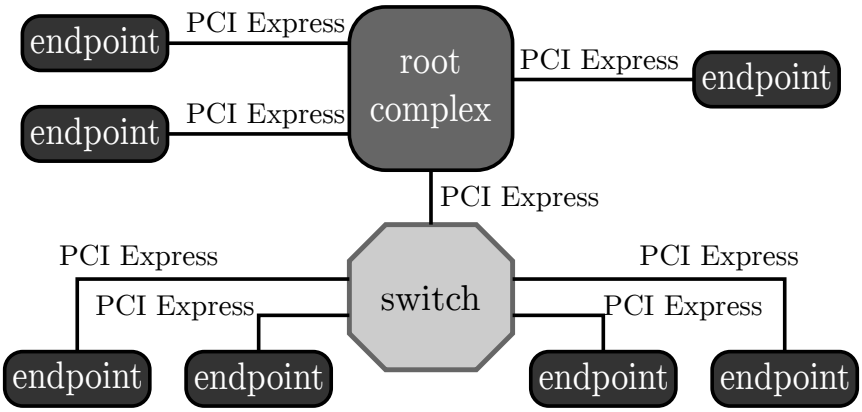


Figure 6.1: Sample PCI Express Topology

packet requires a response, a transaction can consist of either a single packet from a sender, or of a request-completion packet pair, in which case the receiver has to send a *response packet* back to the sender.

The PCI Express specification defines four different transaction types: *memory transactions*, *I/O transactions*, *configuration transactions*, and *message transactions*. Memory transactions are used to transfer data to or from memory or a memory-mapped location, and in order to implement the data transfer, there are three different types of memory transactions: memory read request, memory read completion, memory write request. Note that there is no memory write completion to acknowledge the write request. I/O transactions are similar to memory transactions but for I/O-mapped locations. In contrast to memory transactions, there is a I/O write completion transaction in addition to the other three transactions.

The remaining two transaction types are PCI Express device specific and not "general-purpose" transactions like the previous two: configuration transactions are used for device configuration and setup, and message transactions are used for inter-device communication such as interrupt signalling or power management.

Transaction Type	Subtypes
Memory	Read Request (MRd), Read Completion (CplD), Write Request (MWr)
I/O	Read Request (IORd), Read Completion (CplD), Write Request (IOWr), Write Completion (Cpl)
Configuration	Read Request (CfgRd), Read Completion (CplD), Write Request (CfgWr), Write Completion (Cpl)
Message	Request w/o Data (Msg), Request with Data (MsgD)

Table 6.1: Transaction Types

As this case study is considering abstract packets as transactions, the specifics about the different transaction types are not detailed here, but simply summarised in Table 6.1. Note that the completions for the different transaction types are not transaction specific but generic, and only differ in having payload (Completion Data; *CplD*) or not having any payload (Completion; *Cpl*).

Finally, a PCI Express protocol interface is defined in terms of three abstract protocol layers, similar to the TCP protocol specification: the *transaction layer*, the *data-link layer*, and the *physical layer*. The layer hierarchy is depicted in Figure 6.2. On the lowest abstraction level, the physical layer is responsible for the actual transmission of a transaction (or the bits of a transaction) across a PCI Express link. The data-link layer's responsibility is to establish a fault-tolerant communication to the direct link neighbour. The transaction layer provides the abstraction of a logical, endpoint-to-endpoint, transaction-based communication. This thesis focuses on the transaction layer as it implements complex features that are known to be hard. Also, the higher abstraction level compared to the AMBA AHB protocol strengthens the flexibility of the framework.

The interested reader may consult Wilen *et.al.*'s *Introduction to PCI Express* [WST03] for a more comprehensive general introduction, or Budruk *et.al.*'s in-depth description and reference, *PCI Express System Architecture* [BAS03],

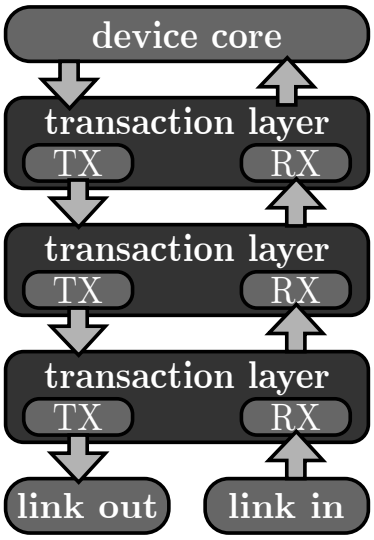


Figure 6.2: The PCI Express Protocol Stack Layers

which provides a detailed, feature-focused discussion of the protocol.

6.1 Transaction Layer

The transaction layer is the uppermost layer of the protocol stack. It provides logical endpoint-to-endpoint communication between sender and receiver of a PCI Express transaction. The unit of communication on transaction-layer level is a *transaction layer packet (TLP)*. As already depicted in Figure 6.2, the transaction layer only communicates with the device core and the data-link layer: it receives requests and completions from the device core and relies on the data-link layer for correct communication with the rest of the PCI Express subsystem. The key features implemented by the transaction layer are:

- virtual channels for prioritisation and categorisation of packets,
- flow control to avoid packet loss because a recipient has no buffer space, and

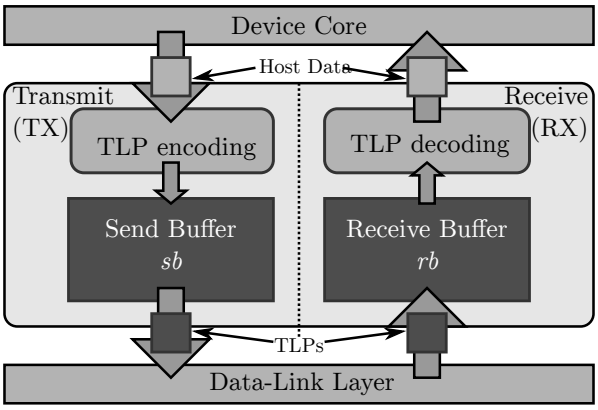


Figure 6.3: The Basic Transaction Layer

- TLP reordering to increase performance under high loads.

This case study details the incremental modelling of a transaction layer that supports all those features: first, a simple, basic transaction layer, implementing none of these features, is specified. Then, this model is extended using transformations and framework components to support the features.

The Basic Model

The basic transaction layer only encodes and decodes TLPs, and provides simple send and receive buffers. If there is a transmission request from the device core, the transmit part (TX) encodes a TLP and adds it to the send buffer. Packets waiting in the send buffer are forwarded to the data-link layer if it is not busy. Similarly, the receive part (RX) receives packets which are passed upwards from the data-link layer. These TLPs are added to the receive buffer, and packets waiting in the receive buffer are handed to the device core after decoding. Figure 6.3 depicts the overall schematic of the basic model.

Most of the encoding and decoding details for TLPs are not relevant in this

context. Thus TLP composition and decomposition is abstracted using two functions

$$\mathbf{enc_{TLP}} : \mathit{HData} \rightarrow \mathit{TLP}$$

$$\mathbf{dec_{TLP}} : \mathit{TLP} \rightarrow \mathit{HData}$$

that generate and decode packets immediately without delay.

There are five different TLP types and Table 6.1 already lists four of them: *memory requests (M)*, *I/O requests (IO)*, *configuration requests (Cfg)*, and *messages (Msg)*—a TLP type for each transaction type. Additionally, *completions (Cpl)* are a distinct TLP type. Moreover, each TLP can be classed into one of three categories:

- *Posted (P)* TLPs do not require a completion to be send back from the receiver to the sender.
- *Non-Posted (NP)* TLPs do require a completion to be send by the receiver back to the sender.
- *Completions (CPL)* are accordingly TLPs that are used to *reply* to non-posted TLPs

Packets of each category can either carry payload or not. These categories are important for the flow control mechanism in PCI Express, which is why they are introduced here.

Transmit Part

To model the transmit part with the framework components, the TLP encoding has to provide a proper interface to the device core as well as to the send buffer. Therefore, a data modification component is used as a wrapper for the encoding function.

$$OPT_{btx} = eOPT \tag{6.1}$$

$$f_{btx} = \lambda\ s \in S,\ i \in (\!|\, di : HData\ option \,|\!).\ \text{let} \tag{6.2}$$

$$tlp = \begin{cases} \mathbf{Some}(\mathbf{enc_{TLP}}(\mathbf{the}\ i.di)) & : v(i) \\ \mathbf{None} & : \text{otherwise} \end{cases}$$

$$\text{in } (\!|\, do = tlp \,|\!)$$

$$b_{btx} = \lambda\, s, i \in (\!|\, bi : \mathbb{B}, di : HData\ option \,|\!).\ (\!|\, bo = i.bi \,|\!) \tag{6.3}$$

$$TLPEnc = DM[OPT_{btx}, b_{btx}, f_{btx}] \tag{6.4}$$

where $eOPT$ is the empty OPT component as defined in Definition 6.1.

Definition 6.1 (Empty OPT Component)

To simplify the instantiation of components when no additional OPT component is needed, the empty OPT component*, $eOPT$, is defined as:*

$$Opt = \emptyset \qquad opt^0 = \emptyset \qquad \delta_{opt} = \lambda s, i.\, s$$

It is trivial to see that the empty OPT component does not affect anything, and $eOPT$ is always used if no optional component is needed.

The transmit part is constructed by composing the *TLPEnc* component from Equation 6.4 with a zero-delay buffer as a send buffer.

Definition 6.2 (Basic Transmit Part)

Let SB_{tl} be a zero-delay buffer M_{zbuf} of arbitrary but fixed size $c_{sb} \in \mathbb{N}$ with $c_{sb} > 1$. The transmit part, TX_{basic}, is defined as the sequential composition of the TLPEnc *component and the send buffer.*

$$TX_{tl} = TLPEnc \;;; SB_{tl}$$

It is easy to see that the basic transmit part *inherits* all buffer correctness properties, except for not changing the data: the busy signal is just forwarded, data is modified without delay, and there is no *OPT* component. The proof is easily derived from the buffer correctness property and can be mechanised using Isabelle/HOL.

Receive Part

The receive part (RX) is symmetric to the transmit part. Again, the TLP decoding function has to be wrapped in a data modification component to provide a standard interface.

$$OPT_{brx} = eOPT \tag{6.5}$$

$$f_{brx} = \lambda\, s \in \S,\ i \in (\!|\, di\colon TLP\ option \,|\!).\ \text{let}$$

$$data = \begin{cases} \mathbf{Some}(\mathbf{dec_{TLP}}(\mathbf{the}\ i.di)) & : v(i) \\ \mathbf{None} & : \text{otherwise} \end{cases}$$

$$\text{in } (\!|\, do = data \,|\!) \tag{6.6}$$

$$b_{brx} = \lambda\ s, i \in (\!|\, bi : \mathbb{B}, di : TLP\ option \,|\!).\ (\!|\, bo = i.bi \,|\!) \tag{6.7}$$

$$TLPDec = DM[OPT_{brx}, b_{brx}, f_{brx}] \tag{6.8}$$

The receive buffer RB_{tl} is also a zero-delay buffer of arbitrary but fixed size c_{rb}. Definition 6.3 summarises the construction.

Definition 6.3 (Basic Receive Part)

Using sequential composition with the output interface of RB_{tl} and the input interface of TLPDec as inner signals, the receive part is given by:

$$RX_{tl} = RB_{tl}\ ;;\ TLPEnc$$

Analogously to the transmit part and because of the same reasoning, the receive part also satisfies all correctness properties of the buffer except for no date modification.

6.2 Virtual Channels and Traffic Classes

In PCI Express and other communication architectures, virtual channels and traffic classes provide a means to prioritize and categorize communication data. Virtual channels create (virtually) independent communication channels between link neighbours. In PCI Express, traffic classes are used to assign TLPs to virtual channels.

In a more general setting without the specifics of traffic classes, a construction that implements a virtual channel mechanism, has to provide three basic features:

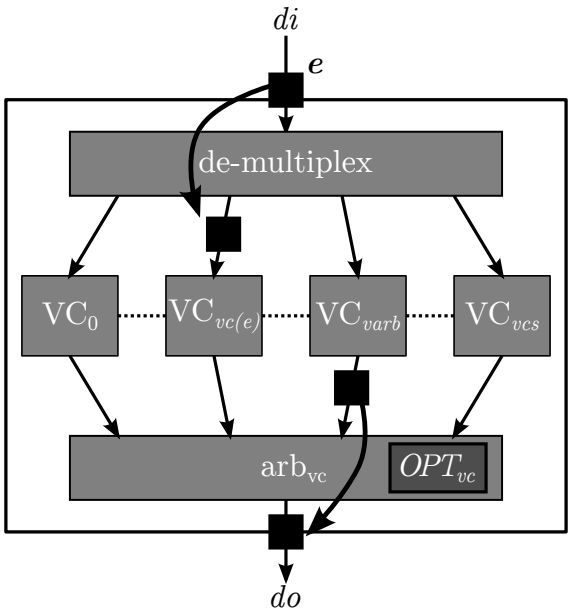


Figure 6.4: The Virtual Channels Transformation

- Separate send or receive buffers for each virtual channel to handle the data elements of each one separately.
- A total mapping from data elements to virtual channels, so that each data element can be mapped to a unique virtual channel. Note that the uniqueness is not strictly a necessity here, but seems *sensible* as mapping a packet to multiple channels would result in data duplication and an increase in data traffic.
- An arbitration function to arbitrate among the different virtual channels: in the end all data has to be send through a single physical channel.

This intuition already indicates that virtual channels can be realised using the replication operator. Figure 6.4 sketches the schematics of the transformation. The transformation is specified in a generic way using uninterpreted functions for all the components. This eases the application of the transformation to other

communication architectures and results in a construction that is symmetric for sender and receiver.

6.2.1 A Generic Virtual Channel Transformation

A transmit or receive part of a communication controller modelled within the framework can be extended with virtual channels by instantiating the replication operator. The resulting virtual channel transformation, *VCTrans*, is an operator on a Mealy machine with input data type α. It is parametrised in:

- $vcs \in \mathbb{N}$: the number of virtual channels,
- $vc : \alpha \to [0, vcs)$: the mapping function, and
- $varb : (Opt \times \tilde{\bigodot}_k ido_k) \to [0, vcs)$: the arbitration function

In order to instantiate the replication operator, the parameters r, OPT, mux, and arb have to be provided. The number of replications is given by $r = vcs$.

Equation 4.38 in Section 4.2.3 specifies that the multiplex function has to be of the following type:

$$\begin{aligned} mux : Opt \times (\!| ibo_1 : \mathcal{BI}^n, \ldots, ibo_r : \mathcal{BI}^n |\!) \,\tilde{\odot}\, \mathcal{DI}^n \\ \to \mathcal{BO}^n \,\tilde{\odot}\, (\!| idi_1 : \mathcal{DI}^n, \ldots, idi_r : \mathcal{DI}^n |\!) \end{aligned} \tag{6.9}$$

Input data is forwarded to the inner component that is specified by the vc mapping. The busy output signal of the composed system is obtained from the busy output of exactly that inner component. The OPT component is only used in the arbitration part to allow the flexibility of using different arbitration schemes,

e. g. a round robin scheme could use the *opt* field to memorise the next channel to be selected. There is no need for an optional component in the multiplex part.

$$
\begin{aligned}
mux_{vc}\ (opt, ibos \mathbin{\tilde{\odot}} i.di) = \text{let} \\
mbo = \text{if } (i.di = \mathbf{Some}\, x) \text{ then } ibos.bo_{vc(x)} \text{ else } \mathsf{F} \\
midi_k = \begin{cases} \mathbf{Some}\, x & : i.di = \mathbf{Some}\, x \wedge vc(x) = k \\ \mathbf{None} & : \text{otherwise} \end{cases} \\
\text{in } (\!|\, bo = mbo, idi_1 = midi_1, \ldots, idi_{vcs} = midi_{vcs} \,|\!)
\end{aligned}
\tag{6.10}
$$

It is easy to see that this definition satisfies $Moore_{out}$ as the data output only depends on the current data input and the vc function, which itself only depends on the current data element. Since vc is a total function over the set of data elements, the instantiation also satisfies the uniqueness property of the multiplex function. Thus,

$$mux_{vc} \models P_{rmux} \tag{6.11}$$

The instantiation of the arbitration function is constructed similarly. From Equation 4.39 in Section 4.2.3, the arbitration function has the form:

$$
\begin{aligned}
arb : Opt \times \mathcal{BI}^m \mathbin{\tilde{\odot}} (\!|\, ido_0 : \mathcal{DO}^m, \ldots, ido_{l-1} : \mathcal{DO}^m \,|\!) \\
\rightarrow (\!|\, ibi_0 : \mathcal{BI}^m, \ldots, ibi_{l-1} : \mathcal{BI}^m \,|\!) \mathbin{\tilde{\odot}} \mathcal{DO}^m
\end{aligned}
\tag{6.12}
$$

Using an uninterpreted arbitration function $varb$, the arbitration function arb can be instantiated to get arb_{vc}:

$$arb_{vc}\ (opt, i.bi \mathbin{\tilde{\odot}} idos) = \text{let} \tag{6.13}$$

$$
\begin{aligned}
& abi_k = i.bi \vee (varb(opt, idos) \neq k) \\
& ado = idos.ido_{varb(opt,idos)} \\
\text{in } & (\!| \, ibi_0 = abi_0, \ldots, ibi_{vcs-1} = abi_{vcs-1}, do = ado \, |\!)
\end{aligned}
$$

Again, it is easy to see that arb_{vc} satisfies P_{rarb} if $varb$ selects the output of exactly one inner component as global output. This property is formulated as:

$$P_{varb} \equiv \forall t.\ \exists! k \in [0, vcs).\ varb(opt, idos).do^t = idos.ido_k^t \tag{6.14}$$

Definition 6.4 (Virtual Channel Transformation)

The unary operator VCTrans *extends a given Mealy machine M_b, which implements a message buffer for data elements of type α, with the virtual channel feature.* VCTrans *for $vcs \in \mathbb{N}$ channels, mapping function $vc : \alpha \to [0, vcs)$, arbitration $varb : (Opt \times \tilde{\bigodot}_k ido_k) \to [0, vcs)$, and optional arbitration component OPT_{vc}. $M = VCTrans\ (M_b)$ is defined as $\Re\, M_b$ with*

$$l = vs \quad OPT = OPT_{vc} \quad mux = mux_{vc} \quad arb = arb_{vc}$$

In the following, the two main correctness results for the transformation are detailed: Lemma 6.1 argues about the liveness property of the composed system, and Theorem 6.2 states that functional correctness is preserved by the transformation.

Lemma 6.1 (Liveness of Virtual Channels)

Given $M = VCTrans\,(M_b)$ and $M_b \models Moore_{out}$. Then M satisfies liveness if M_b satisfies liveness and the arbitration function varb provides weak fairness.

PROOF The lemma follows directly from the replication operator correctness result (Theorem 4.8). The proof is formalised in Isabelle/HOL by instantiating this Theorem and discharging its assumptions using the assumptions stated in the Lemma. ■

Definition 6.5 (Filtered Signal)

Given an input signal $i^t = \mathcal{BI}^m \tilde{\circ} \mathcal{DI}^n$. *The input signal* $i^t[P]$ *for a filter predicate* $P : \mathcal{DI}^n \to B$ *is called* P-filtered input signal *and defined as*

$$i^t[P] = \mathcal{BI}^m \circ (\!| di_1 = i^t.di_1[P_1], \ldots, di_n = i^t.di_n[P_n] |\!)$$

where

$$i^t.di_k[P_k] = \begin{cases} i^t.di_k & : P(i^t.DI) \\ \textbf{None} & : \neg P(i^t.DI) \end{cases}$$

A P-filtered output signal *is defined analogously.*

Theorem 6.2 states the correctness of the transformation in terms of trace equivalence: given an input to the transformed system, the output filtered for a specific virtual channel is trace equivalent to the output generated by a signal inner component with the input trace filtered for that particular channel, modulo the delay caused by the arbitration—weak trace equivalence if a **None** in the output trace is associated with a τ step in standard LTL systems—assuming that the overall system provides liveness.

Theorem 6.2 (Correctness of Virtual Channels)

Given a Mealy machine with virtual channels, $M = VCTrans\ (M_b)$, *input signal* $i^t \in I$, *and output signal* $o^t \in O$. *If* M_b *satisfies a liveness property* P_{live}, *then* M

satisfies the following input-output correctness property based on M_b*:*

$$\forall\ v \in [1, vcs].\ (M.\omega\ (s^t, i^t))[P_v] \equiv_d M_b.\omega\ (s^t.m_v, i^t[P_v])$$

where $\equiv_d$ *denotes trace equivalence modulo the delay caused by the arbitration, i. e. weak trace equivalence considering* **None** *as a* τ *event, and the filter predicate* P_v *is given by:*

$$P_v\ i^t = \big(\mathit{if}\ (i^t.di = \mathbf{Some}\, x)\ \mathit{then}\ (vc(x) = v)\ \mathit{else}\ \mathsf{F}\big)$$

Proof It is easy to see that neither mux_{vc} nor arb_{vc} implement any data modifications. So, if there is a data modification, then it is caused by M_b.

Let idi_v^t be the input to the v-th inner component VC_v, which is an instance of M_b, and let ido_v^t be the corresponding output trace. Using the definition of mux_{vc} and the type definition of $vc(x)$, it can be shown that

$$idi_v^t = i^t.di[P_v] \tag{6.15}$$

which then implies that

$$\omega_b\ (s_b^t, i^t.bi\tilde{\circ}idi_v^t) = \omega_b\ (s_b^t, i^t.bi\tilde{\circ}i^t.di[P_v]) \tag{6.16}$$

since ω_b is deterministic. The left hand side of Equation 6.16 is the definition of the internal output of inner component VC_v, thus ido_v^t. Because the fairness of the arbitration function and the fact that it does not modify the data, i. e. exactly

one virtual channel produces the output of the composed system, we obtain that

$$(M.\omega\,(s^t, i^t))[P_v] \equiv_w ido_v^t \tag{6.17}$$

The key property to the correctness of this step, apart from the assumptions on the arbitration function, is that only data elements in virtual channel v satisfy P_v.

Finally, we obtain that

$$\omega_b\,(s_b^t,\ i^t.bi \mathbin{\tilde{\circ}} i^t.di\ [P_v]) = M.\omega\,(s^t, i^t[P_v]) \tag{6.18}$$

using Definition 6.5 and the fact that VC_v is the only non-empty channel given data inputs $i^t.di[P_v]$ which, together with Equation 6.17, concludes the proof. The proof is tedious but has been mechanised in Isabelle/HOL. ∎

6.2.2 Virtual Channels in PCI Express

As mentioned before, the PCI Express protocol implements the mapping from data packets (TLPs) to virtual channels by introducing traffic classes. Traffic classes relate to different priority levels and each TLP is assigned one of eight different TCs: $\mathsf{TC0}, \ldots, \mathsf{TC7}$ with $\mathsf{TC0}$ being the default class; the others are optional. Additionally, up to eight VCs are supported: $\mathsf{VC0}, \ldots, \mathsf{VC7}$. Again, $\mathsf{VC0}$ is the default channel and required to be enabled. Also, $\mathsf{TC0}$ is always mapped to $\mathsf{VC0}$; all other mappings are configurable. However, the mapping must be a function (exactly one VC for a TC, if TC is used), and the mappings of two link neighbours need to be identical. This configurable mapping of traffic classes to virtual channels is PCI Express' realisation of the multiplex function mux_{vc}. It is

worth noting, that this function does not have to be be surjective and therefore not every VC has to be *active*.

The default priority interpretation is that VC0 corresponds to the lowest and VC7 is assigned the highest priority, and by default PCI Express implements strict priority arbitration among virtual channels. However, this default setting can be adjusted by splitting the virtual channels in two *disjoint* groups:

- a *low priority group* $\mathcal{LP}_{vc}$

$$\mathcal{LP}_{vc} = [\mathsf{VC0} : \mathsf{VC}j] \tag{6.19}$$

- and a *high priority group* $\mathcal{HP}_{vc}$

$$\mathcal{HP}_{vc} = (\mathsf{VC}j : \mathsf{VC7}] \tag{6.20}$$

  for $j \in [0 : 7]$.

The lower priority group can be configured to use an alternative arbitration scheme instead of strict priority: round robin or weighted round robin.

In order to apply the virtual channel transformation *VCTrans* to a send or receive part model of the PCI Express transaction layer, the following parameters and functions need to be instantiated (cf. Definition 6.4):

$$\begin{aligned} vcs &\in \mathbb{N} \\ vc &: TLP \to [0, vcs) \\ varb &: (Opt \times \tilde{\bigodot}_k ido_k) \to [0, vcs) \end{aligned}$$

Assuming the traffic class of a TLP can be determined from a TLP itself, i. e. there exists a function $\mathbf{tc} : \mathit{TLP} \to \{\mathsf{TC0}, \ldots, \mathsf{TC7}\}$, the first two instantiations are straightforward:

$$vcs = 8 \tag{6.21}$$

$$vc = \lambda\ tlp.\ \#(vc_{tc}(\mathbf{tc}\ tlp)) \tag{6.22}$$

where $\# : \{\mathsf{VC0}, \ldots, \mathsf{VC7}\} \to [0, 7]$ denotes an *index operator* that simply returns the index of a virtual channel, i. e. $\#\ VCi = i$, and vc_{tc} is the user-defined TC to VC mapping.

The PCI Express VC Arbitration.

Unfortunately, the PCI Express arbitration scheme complicates the instantiation: without further restrictions on the arbitration scheme or assumptions on the input stream of TLPs, the PCI Express arbitration scheme does not satisfy fairness: imagine the arbitrations scheme is configured to use VC0 and VC1 only, but with strict priority arbitration. Now imagine at some point in time, both virtual channels have packets waiting in their buffers, but starting from this point, a continuous stream of VC1-channel packets arrives. Because of the strict priority arbitration, the packets in VC0 are never going to be send.

The PCI Express specification is aware of this issue and provides additional methods of regulation in this case, which are based on injecting the waiting, low priority packets into the stream of high priority packets at a rate not slower than a fixed minimum. This *seems*—as these methods have neither been modelled nor verified—to prevent starvation of low priority packets, but obviously breaks the

specification of the arbitration scheme, which is an undesirable result as well.

Therefore, this dissertation assumes the system to be configured to use one of the alternative arbitration schemes, round robin or weighted round robin, only. For both of these schemes, it is straightforward to proof fairness; in case of weighted round robin of course only if the weights allow it. Under this assumption, the arbitration function *varb* can easily be instantiated and *VCTrans* can be applied to a PCI Express transaction layer model.

Note, that there are also less restrictive assumptions which lead to fair scheduling even under a strict priority scheme, especially making assumptions on the nature of the stream of packets and excluding streams which can lead to a deadlock for a packet. Such assumptions, however, have to be extremely conservative because such restrictions have to consider potential global effects and not just local ones: for example, low priority packet throughput may also be affected by a flow control mechanism which can affect the frequency with which high priority packets are allowed to arrive.

Commutativity of Transformations.

Virtual channels are a particularly suitable example for pointing out *intrinsic, logical dependencies* between features: even though transformations specify features independently, encapsulate their complexity, and allow features to be added in a structured, controlled way, there are dependencies among features which are unrelated to complexity. Some features simply do not make sense, if they are applied in the wrong order or if another feature is missing. For example, the core idea of a virtual channel is to be virtually identical to the physical channel, modulo timing properties and reduced to the packets from the same

virtual channel (cf. Theorem 6.2). Therefore the virtual channel transformation should be applied after all other transaction layer features have been added, so that all these features are available in each virtual channel.

Another example of a logical dependency between features is packet reordering: it relies on a feature which blocks packets selectively. Of course, as detailed in Section 6.4, the transformation is technically independent and can also be applied to a system which does not support a selective blocking of packets, the reordering feature will just never find a pair of packets that can be reordered.

6.3 Flow Control

Flow control is a mechanism that allows a sender to check whether a receiver has enough space in its local receive buffer before sending a data element. In point-to-point protocols like PCI Express, flow control is therefore a mechanism that operates between two link neighbours, even though it is a transaction layer feature. Given two link neighbours and the *communication path* from one neighbour's transmit part to the other's receive part, the working principle can be summarised as follows: the receiver has one or more flow control (receive) buffers to store data elements and the sender maintains a counter for each of them. These counters maintain a lower bound on the currently available space in the receiver's flow control buffers. Then, before transmitting a data element, the sender checks that the receiver has enough available buffer space using these counters. Assuming the counter values are indeed always a lower bound on the available space on the receiver's side, the sender knows that the receiver has enough buffer space to receive the message.

In contrast to virtual channels, this feature requires asymmetric sender and receiver transformations. Moreover, flow control requires bidirectional communication between sender and receiver: a *data channel*, the "normal" channel, to transport data elements, and a *control channel* in opposite direction, which is used by the receiver to provide the sender with updates on the available space. In the following, sender and receiver are named according to the direction of the data channel. The control channel is required because the sender can only keep track of a reduction in available space. Any increase of available space, i. e. when the receiver moves received data to the device core, happens unnoticed by the sender. Therefore, the receiver sends regular updates of the available space to the sender using the control channel. An overview of the flow control transformations is depicted in Figure 6.5. Note that the control channel does not necessarily have to be a physically dedicated channel, but can also simply be the receiver's data channel to the sender in a system with bi-directional links.

6.3.1 A Generic Flow Control Transformation

Similar to the virtual channel transformation, the flow control transformation is first specified in a generic way without PCI Express specifics. As an asymmetric transformation, the flow control transformation consists of separate transformations for sender, $FCTransTX$, and receiver, $FCTransRX$. The sender part has to extend an existing system with counters, a check mechanism for a data element which is about to be sent, and a handler for counter updates from the control channel. The receive part has to provide one or more flow control buffers, counters for space that was freed, and a mechanism to initiate the transmission of control data. Naturally, sender and receiver need to have common knowledge

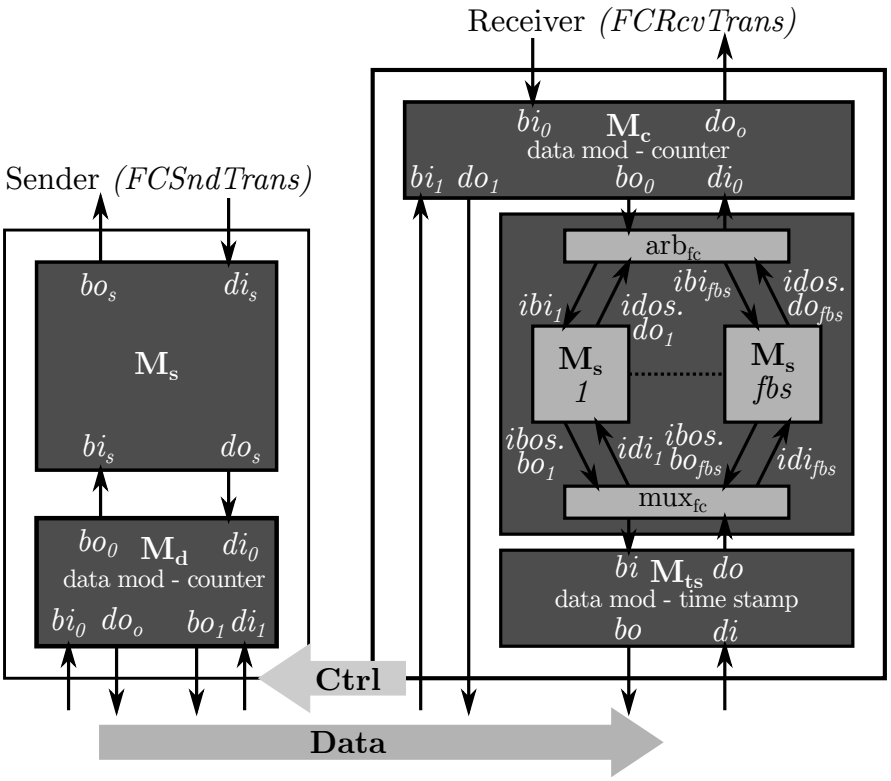


Figure 6.5: Overview of the Flow Control Transformation

and therefore both transformations are parametrised in a data type α and the following values and functions:

- $fbs \in \mathbb{N}$: the number of flow control buffers.
- $fb : \alpha \to [1, fbs]$: a total mapping from data elements to one of the flow control buffers.
- $size : \alpha \to \mathbb{N}$: a measurement function that assigns a size to each data element.
- $s_1, \ldots, s_{fbs}$: the sizes of the flow control buffers.
- $upd : Opt_{r,fc} \to [1, fbs]\, option$: a function to trigger the transmission of an update on the control channel.

The rest of this section uses the following notation: M_s refers to the sender, i. e. the Mealy machine that models the TX part, without flow control, $M_{s,fc}$ denotes the sender with flow control. M_r and $M_{r,fc}$ refer to the receiver Mealy machines, respectively. Moreover, M_s and M_r implement abstract 1-1 interfaces.

Sender

The sender is constructed using a data modification component and a sequential composition. Even though there is no actual data modification happening, the busy signal strengthening is used to implement the check for available space. To account for control channel input, the data modification has to provide a 2-1 abstract interface: one additional input interface to connect to the control channel. The sequential composition operator is used to 'connect' the first input interfaces to the output interface of M_s. The first input interface is referred to as *data interface*, the second one is called *control interface*. The components of the data modification component are indexed with a d in the following.

$$I_d = (\!| \, bi_0 : \mathbb{B},\ di_0 : \alpha\, option,\ di_1 : ([1, fbs] \times \mathbb{N})\, option \, |\!) \tag{6.23}$$

$$O_d = (\!| \, bo_0 : \mathbb{B}.\, bo_1 : \mathbb{B},\ do_0 : \alpha\, option \, |\!) \tag{6.24}$$

Data elements received on the control interface are of type $[1, fbs] \times \mathbb{N}$: an element (x, y) indicates that y units of space have been freed in buffer x, thus the local counter of available space for buffer x needs to be increased by y.

As there is no actual data modification happening, the function f is simply the identity function between the two data interfaces. The control input interface is

only used internally.

$$f_d : S \to DI(I_d) \to DO(O_d)$$
$$f_d = \lambda\, s.\, i.\, (\!|\, do_0 = i.di_0 \,|\!) \tag{6.25}$$

The *OPT* component is used to implement the counters, one for each flow control buffer. The values are updated if there is an input on the control interface.

$$Opt_d = (\!|\, bcnt_1 : \mathbb{N}, \ldots, bcnt_{fbs} : \mathbb{N} \,|\!) \tag{6.26}$$
$$opt_d^0 = (\!|\, bcnt_1 = s_0, \ldots, bcnt_{fbs} = s_{fbs} \,|\!) \tag{6.27}$$
$$\delta_{opt,d} = \lambda opt.\, i.\, \text{let} \tag{6.28}$$
$$opt' = \text{if } (i.di_1 = \textbf{Some } (x, y)) \text{ then } opt(\!|\, bcnt_x += y \,|\!) \text{ else } opt$$
$$opt'' = \text{if } (i.di_0 = \textbf{Some}\, x) \text{ then } opt(\!|\, bcnt_{fcb(x)} -= size(x) \,|\!)$$
$$\text{else } opt'$$
$$\text{in } opt''$$

The counter check is modelled using the busy strengthening function b: if the data cannot be send because there is not enough space, the busy signal to M_s is activated.

$$b_d : S \to I_d \to BO(O_d)$$
$$b_d = \lambda\, s.\, i.\, \text{let} \tag{6.29}$$
$$chk = \text{if } (i.di_0 = \textbf{Some}\, x) \text{ then } (size\, x \leq (s.opt.bcnt_{fc(x)}))$$
$$\text{else } \mathbf{F}$$
$$\text{in } (\!|\, bo_0 = (i.bi \vee \neg chk),\ bo_1 = i.bi \,|\!)$$

This completes the instantiation of the data modification component. Note, that this instantiation does not provide Moore-like input interfaces (as bo_0 depends on di_0). So, to sequentially compose it with M_s, M_s has to provide a Moore-like output interface.

Definition 6.6 (FCTransTX)

The sender part of the flow control transformation is given by the sequential composition of the operand with the data modification instance M_d given by Equations 6.25–6.3.1. For the sequential composition to be applicable, the operand has to provide Moore-like outputs.

$$M_s \models Moore_{out} \implies FCTransTX\ M_s = M_s;;M_d$$

Receiver

The transformation for the receiver is similar to the virtual channel transformation: the flow control buffers are parallelized and received data elements are mapped according to fc. Additionally, however, the receiver with flow control support has to keep track of the space available in the flow control buffers, and needs to initiate the transmission of space updates to the sender. Similar to the sender transformation, the flow control buffers are sequentially composed with a data modification unit that implements these control tasks.

Data elements are passed to an upper layer in the order they have been received. To maintain packet ordering, a time stamp is added to every incoming data element before it is placed in one of the flow control buffers. To add the time stamp to received data elements, another data modification component has

to instantiated:

$$I_{ts} = (\!|\, bi\!:\!\mathbb{B},\ di\!:\!\alpha\ option \,|\!) \tag{6.30}$$

$$O_{ts} = (\!|\, bo\!:\!\mathbb{B},\ do\!:\!(\alpha \times \mathbb{N})\ option \,|\!) \tag{6.31}$$

The *OPT* component is used to maintain the time stamp counter. Time stamps are modelled using naturals, mainly for convenience reason. However, using the finite-size property of the buffers, an upper bound on the number of (simultaneously) needed time stamps can be calculated and natural numbers can be replaced by a finite domain together with proper modulo arithmetic. This also holds for the available space counters.

$$Opt_{ts} = (\!|\, ts\!:\!\mathbb{N} \,|\!) \tag{6.32}$$

$$opt^0_{ts} = (\!|\, ts = 0 \,|\!) \tag{6.33}$$

$$\delta_{opt,ts} = \lambda opt, i.\ \text{if } (i.di = \mathbf{Some}\, x) \text{ then } opt(\!|\, ts := opt.ts{+}1 \,|\!) \text{ else } opt \tag{6.34}$$

No busy signal strengthening is needed for this instantiation, so the function b simply propagates the busy input signal. The data modification function f needs to add the time stamp to each received data element.

$$f = \lambda s, di.\ \text{if } (di = \mathbf{Some}\, x) \text{ then } \mathbf{Some}\,(x, s.opt.ts) \text{ else } \mathbf{None} \tag{6.35}$$

$$b = \lambda s, i.\ (\!|\, bo = i.bi \,|\!) \tag{6.36}$$

Since $b(s, i)$ is independent from $i.di$, M_{ts} provides Moore-like input and output

interfaces.

$$M_{ts} \models Moore_{in} \wedge M_{ts} \models Moore_{out} \tag{6.37}$$

The parallelized flow control buffers are constructed using the replication operator. The abstract data type of M_r is instantiated with $(\alpha \times \mathbb{N})$ to handle the newly added time stamp. The multiplex function mux_{fc} is constructed analogously to mux_{vc} (Equation 6.10) using fcb instead of vc and fbs instead of vcs.

$$\begin{aligned} mux_{fc}\ (opt, ibos \,\tilde{\odot}\, i.di) = \text{let} \\ mbo = \text{if } (i.di = \mathbf{Some}\,(x,t)) \text{ then } ibos.bo_{fcb(x)} \text{ else } \mathsf{F} \\ midi_k = \begin{cases} \mathbf{Some}\,(x,t) & : i.di = \mathbf{Some}\,(x,t) \wedge fcb(x) = k \\ \mathbf{None} & : \text{otherwise} \end{cases} \\ \text{in } (\!|\, bo = mbo, idi_1 ! = midi_1, \ldots, idi_{fbs} = midi_{fbs} \,|\!) \end{aligned} \tag{6.38}$$

The arbitration function selects the oldest element among the flow control buffer outputs:

$$\begin{aligned} arb_{fc}\ (opt, i.bi \,\tilde{\odot}\, idos) = \text{let} \\ sel = k \text{ s.t. } idos.do_k = \mathbf{Some}\,(x,ts) \wedge \\ \qquad (\forall j \neq k . idos.do_j = (x', ts') \implies ts' > ts) \\ abi_k = i.bi \vee (sel \neq k) \\ ado = idos.ido_{sel} \\ \text{in } (\!|\, ibi_1 = abi_1, \ldots, ibi_{fbs} = abi_{fbs}, do = ado \,|\!) \end{aligned} \tag{6.39}$$

The definition of the arbitration function provides a Moore-like output interface, which propagates to the outputs of the flow control buffers.

$$M_{fcb} \models Moore_{out} \tag{6.40}$$

The last part needed to complete *FCTransRX* is the data modification instance M_c which maintains the space counters and removes the time stamp from the data elements. The data modification also provides the additional output interface to sent regular updates to the sender.

$$I_c = (\!|\, bi_0 : \mathbb{B},\ bi_1 : \mathbb{B},\ di_0 : (\alpha \times \mathbb{N})\, option \,|\!) \tag{6.41}$$

$$O_c = (\!|\, bo_0 : \mathbb{B},\ do_0 : \alpha\, option,\ do_1 : ([1, fcbs] \times \mathbb{N})\, option \,|\!) \tag{6.42}$$

The *OPT* component is used to implement the counters. They are updated in two cases: first, when an update via the control channel is sent the corresponding value is reset to zero; second, when a data element is removed from a flow control buffer and moved to the device core the corresponding counter is increased.

$$Opt_c = (\!|\, fcnt_1 : \mathbb{N}, \ldots, fcnt_{fbs} : \mathbb{N} \,|\!) \tag{6.43}$$

$$opt_c^0 = (\!|\, fcnt_1 = 0, \ldots, fcnt_{fbs} = 0 \,|\!) \tag{6.44}$$

$$\delta_{opt,c} = \lambda opt, i.\, \text{let} \tag{6.45}$$

$$opt' = \text{if } (upd\, opt = \textbf{Some}\ n) \text{ then } opt(\!|\, fcnt_n = 0 \,|\!) \text{ else } opt$$

$$opt'' = \text{if } (i.di = \textbf{Some}\, x) \text{ then } opt'(\!|\, fcnt_{fcb(x)} += size(x) \,|\!)$$

$$\text{else } opt'$$

$$\text{in } opt''$$

To finish the M_c instantiation, the functions f and b have to be defined. Once more, the busy strengthening of the data modification is not needed, and the busy input signal is just forwarded to the output. The data modification function f is used for two things: first, the time stamp is removed from each data element. Second, if the trigger functions *upd* fires, a *control packet* with updated space values is send.

$$b_c = \lambda s.\, i.\, (\!|\, bo_0 = i.bi_0 \,|\!) \tag{6.46}$$

$$\begin{aligned} f_c = &\lambda s.\, di.\, \text{let} \\ &co = \text{if } (upd\ s.opt = \mathbf{Some}\, n) \\ &\qquad \text{then } \mathbf{Some}\, (n, s.opt.fcnt_n) \text{ else } \mathbf{None} \\ &\text{in } (\!|\, do_0 = di,\ do_1 = co \,|\!) \end{aligned} \tag{6.47}$$

Definition 6.7 (FCTransRX)

The receiver part of the flow control transformation is given by the following series of sequential compositions:

$$M_r \models Moore_{out} \implies FCTransRX\ M_r = (M_{ts};;M_{fcb}(M_r));;M_c$$

6.3.2 Instantiation for PCI Express

The PCI Express flow control algorithm distinguishes between the different TLP categories: posted (P), non-posted (NP), and completion (Cpl). The mechanism also handles TLP headers (H) and payloads (D) separately as not every TLP contains payload. Thus, PCI Express uses six flow control buffers: PH, PD, NPH, NPD, CplH, CplD. TLPs which only consist of a header are stored in the

TLP	MRd	MWr	IORd	IOWr	CfgRd	CfgWr	Msg	MsgD	Cpl	CplD
Category	NP	P	NP	NP	NP	NP	P	P	CPL	CPL
Data	no	yes	no	yes	no	yes	no	yes	no	yes

Table 6.2: TLP Types and Categories

corresponding header buffer. TLPs with payload are split into header and payload: the former is stored in the header buffer and the latter in the corresponding data buffer. Table 6.2 lists TLP Types with their categories and indicates if a TLP type has data payload or not.

The transformation for the receiver part cannot be applied to PCI Express right away because of the header-payload separation: the flow control transformation uses the replication operator which means that the multiplex component can only select a single inner component to send input data to. But in case of a TLP with payload, both parts have to be stored in different buffers. Therefore, a smaller PCI Express specific transformation is used transform a receiver's send buffer into a proper flow control buffer. Each flow control buffer holds packets of one TLP category, but both header and payload. Given such a *double-buffer* as receive buffer, the flow control transformation for the receiver can be instantiated for three double-buffered flow control buffers.

The mapping from data elements to flow control buffers, fb, is given by Table 6.2. The measurement unit for buffer space, and hence for TLPs, is called *flow control credits (FCCs)*. In FCCs, the size of a TLP without any payload is 1, and the size of a TLP with payload is relative to the number of payload bytes; the unit value is 16 bytes. The models in this dissertation abstract from any byte counts, but TLPs can "carry" their size in FCCs as a natural number in the payload. Thus, instantiating the *size* function is also straightforward.

What remains is the transformation to construct a flow control buffer from

the receiver without flow control M_r. Since a packet with payload is split into header and data, and put into different buffers, the replication operator cannot be applied. Thus, the multiplex/arbitrate composition is instantiated to construct a flow control buffer from two copies of M_r: M_0 and M_1.

Given the flexibility of the multiplex/arbitrate composition, the instantiation of the multiplex component is straightforward: if an arriving TLP has no payload, the TLP is passed on to M_0, in case the TLP has payload, the TLP without the payload is put into M_0 and the payload in M_1. However, in order to apply the multiplex/arbitrate operator, the interfaces of the inner components have to be consistent; which can be easily solved by instantiating the abstract types of M_0 and M_1 with a type for the union of payload and headers (plus time stamp).

6.4 Transaction Reordering

The goal of reordering is to improve performance if a sender cannot send a data packet because it is blocked, for example by the flow control mechanism. In this case, the sender may be able to send the next element in its send buffer. Using the reordering feature, a sender can check whether the next element in the send buffer is allowed to overtake the blocked one and, if reordering is allowed, the sender may be able to send that package. Thus, reordering only makes sense in combination with a flow-control-like feature that can selectively block data elements. Sticking to the interface standard used in this thesis, blocking is assumed to be implemented by raising the busy input signal.

6.4.1 A Generic Packet Reordering Transformation

The generic specification of the reordering transformation is parametrised in a reordering function $pass : \alpha \times \alpha \to \mathbb{B}$ that defines if an element is allowed to overtake (*pass*) another element: $pass\,(e_1, e_2) = \mathbf{T} \equiv e1$ may overtake $e2$.

The reordering transformation extends a given sender M_s with a message buffer and an abstract 1-1 interface. The transformation is specified by instantiating the replication operator in a slightly specific way, namely without any replication, i. e. $r = 1$ and the *identity mux function* that simply maps the inputs to the corresponding outputs:

$$mux_{ro} = \lambda\,(opt, i).\,(\!|\, bo = ibo, idi = i.di \,|\!) \tag{6.48}$$

The idea is to make use of the arbitration component to arbitrate between the two oldest elements in the message buffer by placing the oldest element (the blocked one) in the OPT component.

Note that even though using the data modification component seems more obvious for implementing reordering, it cannot be used because of the busy signal strengthening requirement: in case the oldest element in the buffer is blocked, the busy input signal is active. But in order to access the second oldest element in the buffer, the busy signal has to be set to false to output the oldest element. This however, is weakening the busy signal which is not allowed using the data modification.

If the current data element in not blocked and there no pending data, the arbitration function does not do anything except for forwarding and nothing is stored in the *Opt* field. If the current data element is blocked and there is no

packet already pending in the *Opt* field, the blocked packet is moved to the *Opt* field and the next step, the arbitration tries to send the next data element in the next step in case *pass* indicates that the next packet is allowed to overtake. Thus, the *Opt* field has to provide a single storage space for a pending data element (*pdata*). Additionally, a flag (rr) is used to arbitrate between the two data packets, in case there is a pending one.

$$Opt_{ro} = (\!| \, pdata : \alpha \, option, rr : \mathbb{B} \, |\!) \tag{6.49}$$

$$opt^0_{ro} = (\!| \, pdata = \mathbf{None}, rr = \mathbf{F} \, |\!) \tag{6.50}$$

Recall that δopt for the replication operator takes as arguments the current state, the input signals of the extended machine, as well as the outputs of the system (cf. Equation 4.40). Given a current input to δopt of (opt, i, o), the *pdata* field is only updated if blocked data has to be saved ($i.bi \wedge pdata = \mathbf{None}$) or if pending data can be output ($\neg i.bi \wedge opt.rr$):

$$pdata' = \begin{cases} \mathbf{None} & : \neg i.bi \wedge opt.rr \\ o.do & : i.bi \wedge (opt.pdata = \mathbf{None}) \\ opt.pdata & : \text{otherwise} \end{cases} \tag{6.51}$$

The rr flag is more complex since it represents the control variable and because has to satisfy some important invariants as the arbiter relies on it: for example, the rr flag has to ensure that a packet does not overtake another packet if it is not allowed to. Given an initial state $rr^0 = \mathbf{F}$, the rr flag has to be updated in the following cases: it has to be set to to $\mathbf{T}$ if:

- there is data pending and the next packet is not allowed to overtake.

$$i.bi \wedge (pdata \neq \mathbf{None}) \wedge \neg pass(i.di, pdata) \tag{6.52}$$

- there is no next data element at the data imput.

$$i.bi \wedge (pdata \neq \mathbf{None}) \wedge (i.di = \mathbf{None}) \tag{6.53}$$

Finally, the flag as to be alternated in the following cases:

- if there is pending data and the next packet is allowed to overtake, but the busy input signal is still active

$$i.bi \wedge (pdata \neq \mathbf{None}) \wedge pass(i.di, pdata) \tag{6.54}$$

- if one of the waiting data elements can be output

$$\neg i.bi \wedge (pdata \neq \mathbf{None}) \wedge (rr \vee pass(i.di, pdata)) \tag{6.55}$$

It is easy to see, that with these update rules, rr cannot become true if there is no data pending; this is one of the invariants which are needed to argue about correctness. Summarising the update rules for $pdata$ and rr, the next state function δopt_{ro} is given by:

$$\delta opt_{ro} = \lambda\, opt, i, o.\ \text{let} \tag{6.56}$$

$$pdata' = \text{as in } \textit{Equation } 6.51$$

$$rr' = \begin{cases} \mathbf{T} & : (i.bi \wedge opt.pdata \neq \mathbf{None} \wedge \\ & \quad pass(i.di, opt.pdata)) \vee \\ & \quad (i.bi \wedge (pdata \neq \mathbf{None}) \wedge (i.di = \mathbf{None})) \\ \neg opt.rr & : (i.bi \wedge (opt.pdata \neq \mathbf{None}) \wedge \\ & \quad pass(i.di, opt.pdata)) \vee \\ & \quad (i.bi \wedge (opt.pdata \neq \mathbf{None}) \wedge \\ & \quad (opt.rr \vee pass(i.di, opt.pdata)) \\ opt.rr & : \text{otherwise} \end{cases}$$

$$\text{in } (\!| pdata = pdata', rr = rr' |\!)$$

Then the actual arbitration function works as follows: if the busy input signal is not active and there is no pending data element, it just outputs the data output of M_s. If the busy signal is active, it selects according to the *opt.rr* field which data element to output: if rr is false, the data output of M_s is selected (if it is not equal to **None**), if rr is true, the *opt.pdata* field is selected.

$$arb_{ro} = \lambda\, opt, (\!| i.bi\ \mathbb{B},\ ido : \alpha\ option |\!).\ \text{let} \tag{6.57}$$

$$out = \begin{cases} opt.pdata & : opt.rr \\ ido & : \text{otherwise} \end{cases}$$

$$bout = i.bi$$

$$\text{in } (\!| ibi = bout,\ i.do = out |\!)$$

Row pass Column?	MWr, Msg, MsgD	MRd, IORd, CfgRd	IOWr, CfgWr	CplD	Cpl
MWr, Msg, MsgD	No	Yes	Yes	Y/N	Y/N
MRd, IORd, CfgRd	No	Y/N	Y/N	Y/N	Y/N
IOWr, CfgWr	No	Y/N	Y/N	Y/N	Y/N
CplD	No	Yes	Yes	a) Y/N b) No	Y/N
Cpl	No	Yes	Yes	Y/N	Y/N

Table 6.3: TLP Reordering Rules

Definition 6.8 (ROrdTrans)

Given a passing function pass : $\alpha \times \alpha \to \mathbb{B}$*, the reordering transformation of a Mealy machine* M_b*,* ROrdTrans M_b*, is given by* $\Re\, M_b$ *with* $r = 1$*,* $mux = \lambda\,(opt, inp).\,(\!|\, bo = inp.ibo, idi = inp.di \,|\!)$*,* $arb = arb_{ro}$*, and* $OPT = OPT_{ro}$*.*

6.4.2 Reordering in PCI Express

The PCI Express specification defines a set of reordering rules for TLPs to be transmitted. These rules are supposed to increase the performance: they avoid that a single TLP blocks all communication if only one flow control buffer on the receiver side is full, for example. The rules ensure that the Producer/Consumer programming model, implemented by PCI Express without reordering, is not violated.

The reordering rules are given in Table 6.3. The table specifies whether a *row TLP* may pass a *column TLP*. A *No* indicates that it must not, a *Yes* that is has to pass to avoid deadlock, and a *Y/N* that it may pass to increase performance. Note the duplicated entry for two CplD TLPs: read completions corresponding to different request may overtake each other (case a)) but read completions corresponding to the same request—completions from burst requests—must not

overtake each other. This can be determined from the TLP headers.

Using Table 6.3 we can create a transformation that extends a send buffer with reordering features using the multiplex/arbitrate transformation. The inner components are instantiated with the send buffer which is replicated five times. Each of them stores messages of the types listed in one column. Thus, the multiplex function is straightforward. Additionally, a time stamp is added to each message, similar to the flow control buffers.

The arbitration function is again a bit more tricky. In case the oldest TLP of the five top elements from the buffers is not blocked, it is transmitted. In case that the TLP is blocked, potential reordering takes place. The blocked TLP is placed in a component of the *opt* part and the oldest of the five top elements is chosen again. Interpreting Table 6.3 as a function from two TLP types to Boolean, we can check whether the next oldest can pass the currently blocked TLP.

Note that this approach provides a *one stage* reordering in terms that if a TLP is blocked, only the direct successor may pass the packet. However, the method can be extended to a *k-stage* reordering by introducing a list of k elements in the *opt* part to store blocked TLPs. The selection method is then executed recursively until a non-blocked, younger TLP is found (up to k times). Thus, a TLP may overtake more than one TLP if all TLPs older than that one are blocked.

6.5 Related Work

There is rich literature and plenty of existing work on PCI Express and, especially, on PCI protocol verification or validation. Most existing work on PCI Express,

however, addresses protocol compliance checking for specific, new designs. For example in [HS05], Hyun and Seong present a PCI Express endpoint covering full functionality of the data link and transaction layer. In order to verify protocol compliance and corner case testing, the authors propose a verification environment based on random testing and test bench generation. Thus the verification method is fundamentally different from the proposed methodology. Moreover, the models are written on a lower abstraction level using Verilog HDL [TM98].

Song [Son07] proposes an assertion-based verification environment for PCI, PCI-X, and PCI Express. Besides neither stating how the PCI Express components are modelled nor detailing the checked properties, the verification is also based on testing.

In [MHJG00] Mokkedem *et al.* present a formalization of the PCI 2.1 protocol using the PVS theorem prover. Their model incorporates a solution to the transaction reordering problem in PCI 2.1 and they prove that the Producer/Consumer property is satisfied. Although the authors state that a reusable theory hierarchy was developed, the presented work is tailored to a specific protocol and a monolithic modelling approach is used.

Cansell *et al.* [CGJ+02] also verify the Producer/Consumer Property of the PCI 2.1 protocol. The authors provide an incremental proof and highlight the benefits of replacing the traditional monolithic verification approach with an incremental one. Thus, the basic verification approach is in the same spirit as the one presented here, but the aim of this work is not another specific protocol verification, but a framework to work in.

More recently, Moinudeen *et al.*'s work on PCI-X [MHT06] presents a design approach to support verification. Although the basic idea is similar, the authors'

aim is to integrate the verification process into an existing design process using model checking and testing.

Summarising, the key difference to most existing work is that the focus of this work is not to provide another *ad-hoc* verification of an existing communication protocol, but to provide a framework for a structured, incremental modelling and verification process. Cansell *et al.* verified the PCI Consumer/Producer in a similar spirit, but their verification approach is tailored to a specific verification effort.

Chapter 7

Conclusion

This dissertation has presented a framework for the incremental modelling of on-chip communication architectures with verification in mind. The approach has been applied to two case studies to illustrate the application and breadth of the framework: the ARM AMBA Advanced High-performance Bus (AHB) protocol and the PCI Express high-performance point-to-point protocol. By choosing industrially relevant case studies, this dissertations shows the technical feasibility of an incremental model derivation process. Complete models in this framework provide significant merits against ad-hoc models: they are independent from the actual implementation or design architecture and they are functionally verified. Such models can act as longer-term reference models for new architectures.

The core idea behind the framework is to replace the traditional monolithic modelling and post-hoc verification methodology with an approach that, optionally, includes the verification process into the modelling process to make large-scale modelling and verification efforts feasible. Instead of being an ad-hoc effort, modelling becomes a structured process that can be interleaved with proof.

The core framework consists of only a few components: buffers, data modification, and composition operators: two rather standard ones—sequential and parallel composition—and two sophisticated ones that are related to each other—replication and multiplex/arbitrate. Despite the restricted number of components, the framework is both expressive enough to model large parts of a broad variety of communication architectures and restrictive enough to provide properties that allow the verification of global correctness properties to be reduced to local, simpler verification problems. It is this combination that makes the presented work an adept framework for specifying current or upcoming on-chip communication architectures.

The AMBA AHB case study covers key features of a master, a bus controller that is allowed to initiate transactions, in an arbiter-based maater-slave bus protocol: pipelined bus transfers and support for burst transactions. As discussed, both features are widely used in common SoC buses and are tailored at improving performance. Starting from a simple model, end-to-end data communication is proven correct and maintained while extending the model with features.

The PCI Express case study covers transaction layer features which provide logical endpoint-to-endpoint communication while only communicating to the link neighbour: virtual channels, flow control, and packet reordering. The transaction layer features in particular are features that usually cause a complexity blow-up in the modelling and verification process because they add structural complexity and they aim at providing global properties. In this dissertation, they are modelled using transformations which encapsulate the complexity of each of them. This way the modelling process is well-structured and the verification is reduced to proving local assumptions.

The framework and case studies have been formalised in higher-order logic using the Isabelle/HOL theorem prover. The expressiveness and flexibility of higher-order logic allows heavy use of uninterpreted functions for the framework components and the formalisation of correctness properties at the abstract level of framework components. Another key aspect of using a theorem prover is the ability to manage already proven properties and, together with the built-in reasoning and rewriting capabilities, to reuse and instantiate already proven properties when composing new components. The modelling and verification methodology intrinsically relies on this feature to contain the complexity.

But, as with most work using theorem proving, the overall effort required for modelling and verification using the (mostly) manual work flow, is an important feasibility consideration. To tackle this problem, this dissertation presents approaches to integrate automatic reasoning tools such as the NuSMV model checker or the sledgehammer tool. During instantiation of framework components with concrete functions, the model checker can often be used to discharge local assumptions. Additionally, the framework has been designed carefully so that instantiated framework components lie within the executable subset of HOL, which is supported by the built-in code generator of Isabelle/HOL. Using the code generator and the simple simulation environment from Chapter 2, models in the framework can be translated to executable, functional programs and simulated inexpensively for basic sanity checks before embarking on any verification effort.

This dissertation presents a generic, formalised framework based on abstract state machines for incremental modelling that is versatile and particularly suited for the specification and verification of on-chip communication protocols. It gives a positive answer to the fundamental research question: can models of high-

performance architectures be derived incrementally and verified cost efficient?

7.1 Future Work

Future research directions based upon this work, for both short-term and long-term, are as versatile as the framework itself. The following paragraphs try to outline a few of them.

Basic framework. A natural, shorter-term research direction is to expand the set of transformations presented here to cover more features to further increase the applicability of the framework to the extensive number of existing protocols.

But, a more challenging, mid-term research goal would be to provide a clean well-defined interface between basic framework components, transformations, and final models. This could even extend to the development of an *API* for the specification of transformations. Such an interface would be invaluable for the design of *new* communication architectures and for the specification of new protocol features without further knowledge of the actual target architecture. This research would provide a further step towards a completely *feature-driven design process*.

Tool Environment. Since the work here focuses on the fundamentals of a framework, much future work can be done in the tool environment area. Previous chapters already mentioned that automation of the verification process is a crucial point for even better feasibility of the methodology. Therefore, valuable future work would be to integrate further automatic reasoning tools into the modelling and verification workflow: SMT solvers, SAT solvers, but also symbolic

execution, for example. Ideally, manual theorem proving can be eliminated or reduced to trivial rewriting in order to combine and reuse properties, and the actual verification effort is *sourced out* to automatic tools. Then, the theorem prover should only provide a rich specification language and act as a *knowledge management tool* that provides a common interconnect between tools.

Another valuable tool environment extension and research goal would be a link to a hardware description language to generate RTL-level code from a model. Given such an interface, one could generate a verified reference implementation of a model and could make use of the rich tool support for RTL-level descriptions to derive an optimised, but still verified implementation. Ideally, this implementation can be generated completely automatically so that changes on the reference model automatically propagate

A Long-Term CAD Vision. We conclude the chapter and this dissertation with a long-term research vision for a fully feature-oriented CAD tool for on-chip communication architectures. In the long-run, we envision this work to provide the fundamentals for a tool environment that allows the *user* to design a communication architecture simply by selecting the desired features and by providing some core protocol properties (like bus vs. point-to-point). Such a tool would incrementally construct a verified model that provides the specified features. This model can be symbolically simulated to inspect performance-related effects of different feature sets and be used to generate a RTL reference implementation. Developing protocol families with a well-defined core functionality becomes simply a matter of selecting different transformations.

Of course this is indeed a very visionary point of view, but also a very

challenging, long-term research objective which would provide enormous value to both academia and industry.

References

[ABK08] Eyad Alkassar, Peter Böhm, and Steffen Knapp. Correctness of a Fault-Tolerant Real-Time Scheduler and its Hardware Implementation. In *Proceedings of the Sixth ACM & IEEE Internationa Conference on Formal Methods and Models for Co-Design (MEMOCODE 2008)*, pages 175–186. IEEE Computer Society, June 2008.

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[ACDJ01] Mark Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. A Framework for Microprocessor Correctness Statements. In *Proceedings of the 11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, pages 433–448. Springer, 2001.

[ACM03] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. *Formal Aspects of Computing*, 14:215–227, 2003.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[All01] VSI Alliance™. Vitual component interface standard version 2 (ocb 2 2.0). `http://comelec.enst.fr/dessin/canex/VCI.pdf`, April 2001.

[Alt03] Altera. Avalon Bus Specification. `http://www.altera.com`, July 2003. Document Version 2.3.

[Amj04] Hasan Amjad. Model checking the AMBA protocol in HOL. Technical Report UCAM-CL-TR-602, University of Cambridge, Computer Laboratory, September 2004.

[Amj06] Hasan Amjad. Verification of amba using a combination of model checking and theorem proving. *Electron. Notes Theor. Comput. Sci.*, 145:45–61, January 2006.

[ARM99] ARM. AMBA Specification Revision 2.0. `http://www.arm.com`, 1999.

[ARM08] ARM. AMBA 3 AXI Protocol Specification 1.0. `http://www.arm.com`, 2008.

[Asp00] David Aspinall. Proof General: A Generic Tool for Proof Development. In *Proceedings of the Sixth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 38–42. Springer, March 2000.

[BAS03] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI Express System Architecture*. Addison-Wesley Pearson Education, 2003.

[BAWR07] J. Bhadra, M. S. Abadir, L. Wang, and S. Ray. A Survey of Hybrid Technqiues for Functional Verification. *IEEE Design & Test of Computers*, 24(2):112–122, 2007.

[BBC+06] Noah Bamford, Rekha K. Bangalore, Eric Chapman, Hector Chavez, Rajeev Dasari, Yinfang Lin, and Edgar Jimenez. Challenges in System on Chip Verification. In *Proceedings of the Seventh International Workshop on Microprocessor Test and Verification*, MTV '06, pages 52–60, Washington, DC, USA, 2006. IEEE Computer Society.

[BF98] W. J. Bainbridge and S. B. Furber. Asynchronous Macrocell Interconnect using MARBLE. In *Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 0122–, Washington, DC, USA, 1998. IEEE Computer Society.

[BM08] Peter Böhm and Tom Melham. A Refinement Approach to Design and Verification of On-Chip Communication Protocols. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, pages 136–143. IEEE Computer Society, November 2008.

[BN02] Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In *Selected papers from the International Workshop on Types for Proofs and Programs (TYPES'00)*, volume 2277, pages 24–40. Springer, 2002.

[Böh09] Peter Böhm. Incremental Modelling and Verification of the PCI Express Transaction Layer. In *Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, MEMOCODE'09, pages 36–45. IEEE Computer Society, July 2009.

[Böh10a] Peter Böhm. A framework for incremental modelling and verification of on-chip protocols. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 159–166. IEEE Computer Society, October 2010.

[Böh10b] Peter Böhm. Incremental and Verified Modelling of the PCI Express Protocol. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 29:1495–1508, October 2010.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Roveri Marco Pistore, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An Open Source Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, pages 359–364. Springer, 2002.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71. Springer, 1982.

[CGG07] Xiaofang Chen, Steven M. German, and Ganesh Gopalakrishnan. Transaction Based Modeling and Verification of Hardware Protocols. In *FMCAD'07*, pages 53–61. IEEE, 2007.

[CGJ+02] Dominique Cansell, Ganesh Gopalakrishnan, Michael Jones, Dominique Méry, and Airy Weinzoepflen. Incremental proof of the producer/consumer property for the pci protocol. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 22–41, London, UK, UK, 2002. Springer-Verlag.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *ICCD'92*, pages 522–525, 1992.

[Dij68] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, February 1968.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DRS04] Vijay D'silva, S. Ramesh, and Arcot Sowmya. Synchronous Protocol Automata: A Framework for Modelling and Verification of SoC Communication Architectures. In *Proceedings of the conference on*

Design, automation and test in Europe - Volume 1, DATE '04, pages 10390–, Washington, DC, USA, 2004. IEEE Computer Society.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In Jaco de Bakker and Jan van Leeuwen, editors, *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, volume 85 of *LNCS*, pages 169–181. Springer-Verlag, 1980.

[FF93] S. Finn and M. Fourman. *The LAMBDA Logic. Abstract Hardware Limited, September 1993. In LAMBDA 4.3 Reference Manuals.*, 1993.

[Gor00] Mike Gordon. *From LCF to HOL: a short history*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

[Haf09] Florian Haftmann. Code generation from Isabelle/HOL theories. `http://isabelle.in.tum.de/website-Isabelle2009-1/dist/Isabelle/doc/codegen.pdf`, 2009.

[HKV02] D Harel, O Kupferman, and M Y Vardi. On the Complexity of Verifying Concurrent Transition Systems. *Information and Computation*, 173(2):143–161, 2002.

[HS05] Eugin Hyun and Kwang-Su Seong. Design and Verification for PCI Express Controller. In *ICITA '05: International Conference on Information Technology and Applications*, volume 2, pages 581–586. IEEE, 2005.

[HSV94] L. Helmink, P. A. Sellink, M., and W. Vaandrager, F. Proof-checking a data link protocol. In *TYPES'93*, pages 127–165. Springer, 1994.

[Hur] Joe Hurd. Metis theorem prover. `http://www.gilith.com/research/metis/`.

[IBM] IBM Microelectronics. CoreConnect bus architecture. `http://www-306.ibm.com/chips/products/coreconnect/`.

[Isa] Isabelle 2009-1. `http://isabelle.in.tum.de/website-Isabelle2009-1/index.html`.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, April 2003.

[KEM11] Suleiman Abu Kharmeh, Kerstin Eder, and David May. A design-for-verification framework for a configurable performance-critical

communication interface. In *Proceedings of the 9th international conference on Formal modeling and analysis of timed systems*, FORMATS'11, pages 335–351, Berlin, Heidelberg, August 2011. Springer-Verlag.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: the automata-theoretic approach.* Princeton University Press, 1994.

[Ler] Xavier Leroy. The Objective Caml system—documentation and user's manual. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[LSGL95] Victor Luchangco, Ekrem Söylemez, Stephen J. Garland, and Nancy A. Lynch. Verifying timing properties of concurrent algorithms. In *Formal Description Techniques VII*, pages 259–273. Chapman & Hall, Ltd., 1995.

[McM97] Kenneth L. McMillan. A Compositional Rule for Hardware Design Refinement. In *CAV '97*, pages 24–35. Springer, 1997.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell Systems Technical Journal*, 34:1045–1079, September 1955.

[MHJG00] Abdel Mokkedem, Ravi M. Hosabettu, Michael D. Jones, and Ganesh C. Gopalakrishnan. Formalization and Analysis of a Solution to the PCI 2.1 Bus Transaction Ordering Problem. *Form. Methods Syst. Des.*, 16(1):93–119, 2000.

[MHT06] Haja Moinudeen, Ali Habibi, and Sofiene Tahar. Design for Verification of the PCI-X Bus. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '06, pages 187–188, Washington, DC, USA, 2006. IEEE Computer Society.

[Mil72] Robin Milner. Logic for Computable Functions: description of a machine implementation. Technical Report CS-TR-72-288, Stanford University, Department of Computer Science, May 1972.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Müf04] Friedger Müffke. *A Better Way to Design Communication Protocols.* PhD thesis, University of Bristol, May 2004.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* LNCS vol. 2283, Springer, 2002.

[NPW09] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle's Logics: HOL. `http://isabelle.in.tum.de/website-Isabelle2009-1/dist/Isabelle/doc/logics-HOL.pdf`, 2009.

[NS95] Tobias Nipkow and Konrad Slind. I/O automata in Isabelle/HOL. In *TYPES'94*, volume 996 of *LNCS*, pages 101–119. Springer, 1995.

[OCP05] OCP International Partnership. Open Core Protocol Specification, Release 2.1. `http://www.ocpip.org`, 2005.

[Pau09a] Lawrence C. Paulson. Isabelle's Logics. `http://isabelle.in.tum.de/website-Isabelle2009-1/dist/Isabelle/doc/logics.pdf`, 2009.

[Pau09b] Lawrence C. Paulson. Isabelle's Logics: FOL and ZF. `http://isabelle.in.tum.de/website-Isabelle2009-1/dist/Isabelle/doc/logics-ZF.pdf`, 2009.

[PS06] PCI-SIG. *PCI Express Base Specification Revision 2.0*, December 2006.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Proceedings of the 5th Colloquium on International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RMK03] Abhik Roychoudhury, Tulika Mitra, and S. R. Karri. Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 828–833, Washington, DC, USA, 2003. IEEE Computer Society.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15:91–110, August 2002.

[SB06] Julien Schmaltz and Dominique Borrione. Towards a formal theory of on chip communications in the ACL2 logic. In *ACL2'06*, pages 47–56. ACM, 2006.

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[Sch07] Julien Schmaltz. A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware. In Jason Baumgartner and Mary Sheeran, editors, *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Proceedings*, pages 223–230. IEEE Computer Society, 2007.

[Sco93] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121:411–440, December 1993. Annotated version of the 1969 manuscript.

[Seg06] C. Seger. The Design of a Floating Point Unit using the Integrated Design and Verification (IDV) System. In M. Sheeran and T. Melham, editors, *DCC '06: Participants' Proceedings*, March 2006.

[Sei94] Karen Seidel. Case Study: Specification and Refinement of the PI-Bus. In *Proceedings of the Second International Symposium of Formal Methods Europe on Industrial Benefit of Formal Methods*, FME '94, pages 532–546, London, UK, 1994. Springer-Verlag.

[Sle] The Sledgehammer: Let Automatic Theorem Provers write your Isabelle scripts. `www.cl.cam.ac.uk/research/hvg/Isabelle/sledgehammer.html`.

[SMSB05] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. XFM: An incremental methodology for developing formal models. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4):589–609, 2005.

[Son07] An Song, Min. System Level Assertion-Based Verification Environment for PCI/PCI-X and PCI-Express. In *Computational Intelligence and Security (CIS '07)*, pages 1035–1038. IEEE, 2007.

[STM] STMicroelectronics. STBus Interconnect. `http://www.st.com/stonline/products/technologies/soc/stbus.htm`.

[TM98] E. Thomas, Donald and R. Moorby, Philip. *The Verilog hardware description language (4th ed.)*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[Tve05] Sergey Tverdyshev. Combination of Isabelle/HOL with Automatic Tools. In *Proceedings of the Fifth International Workshop on Frontiers of Combining Systems (FroCoS'05)*, volume 3717 of *Lecture Notes in Computer Science*, pages 302–309. Springer, September 2005.

[Wei99] Christoph Weidenbach. System Description: Spass Version 1.0.0. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, CADE-16, pages 378–382, London, UK, 1999. Springer-Verlag.

[Wen09] Markus Wenzel. The Isabelle/Isar Reference Manual. `http://isabelle.in.tum.de/website-Isabelle2009-1/dist/Isabelle/doc/isar-ref.pdf`, 2009.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

[WST03] Adam H. Wilen, Justin P. Schade, and Ron Thornburg. *Introduction to PCO Express: A Hardware and Software Developer's Guide*. Intel Press, 2003.