FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Evaluation of Tools and Methodology for Property-Based Logic Synthesis (Deliverable 2.3/1)

Due date of deliverable: 31 December 2006
Actual submission date: 31 December 2006

Start date of project: January 1, 2004 Duration: Three years

Organisation name of lead contractor for this deliverable: TU Graz

Revision 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	November 3, 2006	Creation	Jobstmann, Bloem, Galler, Weiglhofer, Pnueli	All
0.2	November 15, 2006	Restructured GenBuf, added frontmatter, general edits	Jobstmann, Bloem	All but AMBA section
0.3	December 8, 2006	Addressing comments of project manager	Jobstmann, Bloem	Frontmatter, Introduction, Preliminaries
1.0	December 15, 2006	Final approval by project manager	Eisner	-

Authors

Roderick Bloem
Stefan Galler
Barbara Jobstmann
Amir Pnueli
Martin Weiglhofer[1]

Executive Summary

In this document we describe two case studies that we performed using the property-based synthesis techniques developed in the PROSYD project. The first case study is the generalized buffer, an IBM tutorial design used also in the error localization case study. The second case study is the arbiter for ARM's AHB bus. This case study demonstrates that property synthesis can be used for real life commercial control blocks.

Automatic synthesis of logic from its specification has long been considered an unattainable dream. The conclusions of the case studies are that it is no longer a dream. Although we still run in to size restrictions relatively early, we are able to synthesize realistic examples.

In this document, we show the specifications that we derived for the two case studies, we describe how to generate circuits from the results of the synthesis algorithm, and we discuss the benefits and disadvantages of property-based synthesis.

[1]This document is based in part on a paper coauthored with Nir Piterman.

Purpose

The purpose of this document is to describe the effort done in Prosyd Workpackage 2 on property synthesis, in particular, Deliverable 2.3/1, the case studies.

Intended Audience

This guide is intended for people that want to evaluate the promises of synthesis from specifications. A basic understanding of PSL and formal verification is assumed but this document does not go into great technical detail.

Background

Synthesis of linear-time formulas is closely related to *Church's problem*, whether one can automatically synthesize circuits for specificatins given in S1S [Chu62]. It was formalized for LTL by Pnueli and Rosner [PR89a] and their approach carries over to PSL. Recent work performed within Prosyd (Deliverable 2.2/1, Deliverable 1.2/7, and [PPS06a]) has yielded a powerful method for synthesis of a major subset of LTL and PSL. We employ this method here.

Contents

Table of Revisions iii
Authors iii
Executive Summary iii
Purpose iii
Intended Audience iv
Background iv
Contents v
Table of Figures vi
Glossary vii
1 Introduction 1
2 Preliminaries 3
  2.1 Property Specification Language 3
  2.2 Synthesis of GR(1) Properties 3
3 IBM GenBuf 7
  3.1 Handshake between Senders and GenBuf 8
  3.2 Handshake between Receivers and GenBuf 8
  3.3 Formal Specification 9
  3.4 Synthesis 12
4 AMBA AHB 15
  4.1 The AMBA AHB Protocol 15
  4.2 Specification 16
  4.3 Synthesis 19
5 Generating a Circuit 23
6 Discussion 25
7 Conclusions 27
8 References 29
A GenBuf Specification 31
  A.1 Original Specification 31
  A.2 GenBuf Config File 37
B AMBA Specification 43
  B.1 AMBA Config File for Two Masters 43

Table of Figures

Figure 1 - Block diagram of GenBuf 7
Figure 2 - An example of a Sender-to-GenBuf handshake 8
Figure 3 - An example of a GenBuf-to-Receiver handshake 9
Figure 4 - Monitor for Guarantee 12.. 13
Figure 5 - An example of AMBA bus behavior 17
Figure 6 - Deterministic Monitor for Assumption 1 for one master 20
Figure 7 - Deterministic Monitor for Assumption 1 for all masters........ 20
Figure 8 - Monitor for Guarantee 2 for one master 21
Figure 9 - Monitor for Guarantee 3 21
Figure 10 - Monitor for Guarantee 10 for one master 21

Glossary

Acceptance Condition
A condition defining how an infinite automaton accepts an input object. We use Büchi and co-Büchi acceptance conditions both defined by a set of states F. An input word is Büchi accepted by an automaton, if the set of states visited infinitely often while reading the input word intersects the set F. Dually, a word is co-Büchi accepted if the set of states visited infinitely often does not intersect F.

Alternating Tree Automaton
An automaton with an arbitrary branching mode running on trees.

Atomic Proposition
An atomic proposition of a formula in a propositional logic corresponds to signals in a design or implementation.

AWT
Alternating Weak Tree Automaton. An alternating tree automaton with a particularly structured state space. The states are partitioned into partially ordered sets. Each set is classified as accepting or rejecting. The transition function is restricted so that in each transition, the automaton either stays at the same set or moves to a set smaller in the partial order.

Branching Mode
The branching mode is a way to classify automata. We distinguish between four branching modes: Deterministic, nondeterministic, universal, and alternating. In a deterministic automaton, the transition function maps from state and letter to a single state. The transition functions of nondeterministic and universal automata map to sets of states. The automata differ in the way they accept an input word or tree. In a nondeterministic automaton the suffix of the word or tree should be accepted by one of the states in the set. In the universal automaton all states in the set have to accept the suffix. An alternating automaton can have nondeterministic and universal edges.

Infinite Game
A finite state machine on which two players, the protagonist and the antagonist, determine the run, by each determining part of the input. The game comes with a winning condition and the task of the protagonist is to make sure that the run satisfies this condition.

Language Emptiness
The language of an automaton is empty iff the automaton accepts no input object (word or tree), that means there is no accepting run for this automaton.

LTL
Linear Temporal Logic or Linear-time temporal logic. LTL is a temporal logic for property specification in formal verification [Pnu77].

LTL Game
An infinite game where the winning condition is given as LTL formula. All plays in which the sequence of states visited fulfill the given formula are winning for the protagonist. Otherwise the antagonist wins.

Mu Calculus
A calculus of predicates and binary relations which enables writing and solving relational equations among states.

NBT
Nondeterministic Büchi Tree Automaton. An alternating tree automaton with Büchi acceptance condition and nondeterministic branching mode.

NBW
Nondeterministic Büchi Word Automaton. An alternating automaton with Büchi acceptance condition and nondeterministic branching mode. The automaton runs on words.

PSL
Property Specification Language, the language for specification of designs upon which PROSYD is based.

PSL Game
Similar to an LTL game but with a PSL formula as winning condition.

Realizable
A given formula ψ over a sets of input I and output O signal is realizable if there exists a strategy $f : (2^I)^* \to 2^O$ such that all the computations of the system generated by f satisfy ψ. Intuitively, a specification is realizable if there exists a system that can respond in such a way that independent of the input values the environment chooses the combination of inputs and outputs always fulfills the given formula.

RTL
Register Transfer Level (RTL) is a way of describing the operation of a digital circuit. In RTL design, a circuit's behavior is defined in terms of the flow of signals or transfer of data between registers, and the logical operations performed on those signals.

Synthesis
The process of automatically generating a design from a given specification. Formally, check if the given specification is realizable and find a witness.

UCT
Universal co-Büchi Word Automaton. An alternating tree automaton with co-Büchi acceptance condition and universal branching mode.

Verilog
A hardware description language used to design electronic systems at the component, board and system level.

VHDL
VHSCI Hardware Description Language (VHDL) is a hardware description language used to design electronic systems at the component, board and system level. VHDL allows models to be developed at a very high level of abstraction.

VHSCI
A VHSCI is a Very-High-Speed Integrated Circuit, a type of digital logic circuit.

Winning Strategy
A recipe with which a player is guaranteed to win an infinite game, no matter what the other player does. A finite state strategy may depend on a finite memory of the

past, i.e., the move the strategy suggests can depend on previous moves of the two players. A memoryless strategy depends only on the current state of the game.

1 Introduction

We propose to use PSL as a high-level hardware description language. PSL allows for a compact and unambiguous representation of a block of hardware. When used as a basis for synthesis, it yields a design that is correct by construction. The idea of automatic synthesis from specifications is old, but used to be completely impractical. Recently, great strides towards efficient synthesis from specifications have been made. In this document we show that these methods can be used to synthesize real-life blocks of control logic from specification. We use two case studies in which we synthesize a controller for a generalized buffer and an arbiter for ARM's AMBA AHB bus from their PSL specifications.

To our knowledge, this is the first realistic example that has been synthesized automatically from its specification. Previous work has been limited to toy examples such as elevator controllers and traffic light controllers [HRS05, JB06, PPS06a]. This work is the first to tackle a block of logic that is employed commercially.

As the complexity of digital circuits keeps increasing, there are two central ideas that have been proposed and widely adopted in order to deal with this mounting complexity. The first idea is that the development of designs should start at a very high level of specification, much higher than the common practice of starting design developments at the RTL level. Once such a high-level specification is created and analyzed, the development should continue by step-wise refinement of the design going through the phases of RTL (equivalently, VERILOG or VHDL) description, gate-level, etc. The second idea is that each inter-level transformation should be formally verified, in order to guarantee the integrity of the development.

In this document we consider an alternative. Instead of manually deriving a VERILOG implementation to be followed by formal verification, we apply an automatic *high-level synthesis* process which generates a correct-by-construction VERILOG code directly from the PSL specification. For simplicity, we will refer to this form of high-level synthesis as "synthesis", but emphasize that it should not be confused with the synthesis of a gate-level description from VERILOG, RTL code, or from a high-level behavioral description.

In this deliverable, we demonstrate the viability of the synthesis approach for the derivation of a correct VERILOG code from a PSL specification.

Automatic synthesis of digital designs from (temporal) logical specifications has always engaged the imagination of many designers and has been considered as one of the most ambitious and challenging problems in circuit design. First identified as Church's problem [Chu63], several methods have been proposed for its solution [BL69, Rab72]. The problem has been considered again in [PR89b] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL), which is a subset of PSL. The method proposed in [PR89b] for a given LTL specification φ starts by constructing a Büchi automaton $\mathcal{B}_\varphi$, which is then converted into a deterministic Rabin automaton. This double translation may reach a doubly exponential complexity in the size of φ.

The high complexity established in [PR89b] caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to use it for any sizeable system development. Yet there exist several interesting cases where, if the specification of the design to be synthesized is restricted to simpler automata or partial fragments of LTL, it has been shown that the synthesis problem can be solved in polynomial time. Major progress has been achieved in PROSYD Deliverable 2.2/1 ([PPS06b]), which shows that designs can be automatically synthesized from LTL formulas belonging to the class of *generalized reactivity* of rank 1 (GR(1)), in time N^3 where N is the size of the state space of the design (see Section 3.4 and 4.3). The class GR(1) covers the vast majority of properties that appear in specifications of circuits.

In this document we synthesize a generalized buffer that is also used in the error localization case studies and that constitutes a good example because a relatively complete set of specifications had already been written. Furthermore, we demonstrate the application of the synthesis method to a specification of one of the AMBA buses [ARM99]. We chose this example because it represents a characteristic industrial case which is not too big.

Apart from the case studies we discuss how to generate circuits from the BDDs that the synthesis approach generates, and we discuss the benefits and disadvantages of automatic property synthesis.

The paper continues as follows: in Section 2, we (briefly) introduce the PSL specification language and the synthesis method as developed in Deliverable 2.2/1. In Section 3, we discuss the case study of the generalized buffer and in Section 4, we discuss the AMBA arbiter. In Section 5 we describe how the circuit is generated. Finally, in Section 7 we present our conclusions.

A paper containing the experiences described in this document has been accepted for publication at the 2007 Conference on Design Automation and Test in Europe.

2 Preliminaries

2.1 Property Specification Language

We will briefly recapitulate the definition of PSL inasfar as it is needed in this deliverable. For a full exposition we refer the reader to [EF06].

We consider a set of propositions AP and infinite sequences of truth assignments to AP. We refer to such sequences as infinite words over the alphabet 2^{AP}. We denote the set of all infinite words over 2^{AP} by $(2^{AP})^{\omega}$. We denote by $B(AP)$ the set of Boolean formulas over AP.

The syntax of PSL formulas used in this document is defined as $\varphi ::=$

$$\begin{array}{l}
b \mid\mid \neg\varphi \mid\mid \varphi \vee \varphi \mid\mid \varphi \wedge \varphi \mid\mid \varphi \rightarrow \varphi \mid\mid \varphi \leftrightarrow \varphi \mid\mid \\
\varphi \ \texttt{until\_} \ \varphi \mid\mid \\
\varphi \ \texttt{before} \ \varphi \mid\mid \varphi \ \texttt{before\_} \ \varphi \mid\mid \\
\texttt{eventually!} \ \varphi \mid\mid \texttt{always} \ \varphi \mid\mid \\
\texttt{next!} \ \varphi \mid\mid \texttt{prev}(\varphi) \\
\mathsf{next\_event!} \ (b)\,(\varphi) \mid\mid \\
\texttt{rose}(b) \mid\mid \texttt{fell}(b), \text{ where } b \in B(AP)
\end{array}$$

Furthermore, we use the abbreviation φ `until_` $[j]$ φ (proposed in Deliverable 4.3/1 as extension to PSL) to state nested `until_` formulas.

We do not give the formal semantics of PSL but rather explain intuitively what the temporal operators stand for. Boolean connectives are interpreted in the usual way. The formula φ_1 `until` φ_2 holds if φ_1 holds continuously until φ_2 holds (or forever, if that never happens). The formula φ_1 `until_` $[j]$ φ_2 means that φ_1 holds up to and including the jth time φ_2 holds (or forever, if that never happens). The formula φ_1 `before` φ_2 holds if the next occurrence of φ_1 holds strictly before the next occurence of φ_2 (if any such occurence exists). The formula φ_1 `before_` φ_2 holds if the next occurence of φ_1 holds before or simultaneous with the next occurence of φ_2 (if any such occurence exists). The formula `eventually!` φ holds if φ holds in some future (or present) time. The formula `always` φ holds if φ holds forever. The formula `next!` φ holds if in the next time step φ holds and `prev`(φ) holds if φ held in the previous step. Finally, the formula next_event! $(b)\,(\varphi)$ holds if φ holds at the next time b holds, `rose`(b) and `fell`(b) hold when signal b has made a transition from low to high or from high to low, respectively.

2.2 Synthesis of GR(1) Properties

We briefly review the results presented in Deliverable 2.2/1, Deliverable 1.2/7, and [PPS06b] on checking realizability and synthesizing of GR(1) properties. We are interested in the question of *realizability* of PSL specifications (cf. [PR89b]). Assume two sets of Boolean variables $\mathcal{X}$ and $\mathcal{Y}$. Intuitively $\mathcal{X}$ is the set of input variables controlled by the environment and $\mathcal{Y}$ is the set of system variables. *Realizability* amounts to checking whether there exists an *open controller* that satisfies the specification. Such a controller can be represented as an automaton which, at any step, reads values of the $\mathcal{X}$ variables and outputs values for the $\mathcal{Y}$ variables. We consider the realizability and *synthesis*, namely, the construction of such controllers, by solving *games* over a graph whose nodes are all the possible truth assignments to $\mathcal{X}$ and $\mathcal{Y}$.

Here we concentrate on a subset of PSL for which realizability and synthesis can be solved efficiently. The specifications we consider are of the form $\varphi = \varphi^e \rightarrow \varphi^s$. We require that φ^α for $\alpha \in \{e, s\}$ can be rewritten as a conjunction of the following parts.

- φ_i^α – a Boolean formula which characterizes the initial states of the implementation.
- φ_t^α – a formula of the form $\bigwedge_{i \in I}$ `always` B_i where each B_i is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form `next!` v where $v \in \mathcal{X}$ if $\alpha = e$, and $v \in \mathcal{X} \cup \mathcal{Y}$ otherwise.
- φ_g^α – has the form $\bigwedge_{i \in I}$ `always eventually!` B_i where each B_i is a Boolean formula.

In order to allow formulas of other forms (e.g., `always` $(p \rightarrow (q$ `until!` $r))$ where p, q, and r are Boolean, or the formulas in Section 3.3 and 4.2) we augment the set of variables by adding deterministic monitors. Deterministic monitors are variables whose behavior is deterministic according to the choice of the inputs and the outputs. These monitors follow the truth value of the expression nested inside the `always` operator. We rewrite these types of formulas to the form `always eventually!` B where B is a Boolean formula using the variables of the monitor.

We reduce the realizability problem of a PSL formula to the decision of the winner in an infinite two-player game played between a system and an environment. The goal of the system is to satisfy the specification regardless of the actions of the environment. A *game structure* is the multi-graph whose nodes are all the truth assignments to $\mathcal{X}$ and $\mathcal{Y}$. A node v is connected by edges to all the nodes v' such that the truth assignments to $\mathcal{X}$ and $\mathcal{Y}$ satisfy $\varphi_t^e \wedge \varphi_t^s$ where v supplies the assignments to the current values and v' to the next values. We then group all the edges that agree on the assignment of $\mathcal{X}$ in v' to one multi-edge. A play starts by the environment choosing an assignment to $\mathcal{X}$ and the system choosing a state in $\varphi_i^e \wedge \varphi_i^s$ that agrees with this assignment. A play proceeds by the environment choosing a multi-edge and the system choosing one of the nodes connected to this multi-edge. When this interaction produces an infinite play the system wins if in addition it satisfies $\varphi_g^e \rightarrow \varphi_g^s$.

We *solve* the game, attempting to decide whether the game is winning for the environment or the system. If the environment is winning the specification is *unrealizable*. If the system is winning, we *synthesize* a winning strategy which is a *working implementation* for the system. Formally, we have the following.

Theorem 1. [PPS06b] *Given sets of variables $\mathcal{X}$ and $\mathcal{Y}$ and a* PSL *formula φ of the form presented above with m and n conjuncts, we can determine using a symbolic algorithm whether φ is realizable in time proportional to $(mn2^{d+|\mathcal{X}|+|\mathcal{Y}|})^3$ where d is the number of variables added by the monitors for φ.*

3 IBM GenBuf

The generalized buffer (henceforth referred to as *GenBuf*) is a design that has been developed by IBM as a tutorial for the Rulebase verification tool[2]. GenBuf comes with a relatively complete specification in PSL. We have synthesized the control logic of GenBuf.

Figure 1 contains a block diagram of the design and its interface. Dashed boxes represent the environment.

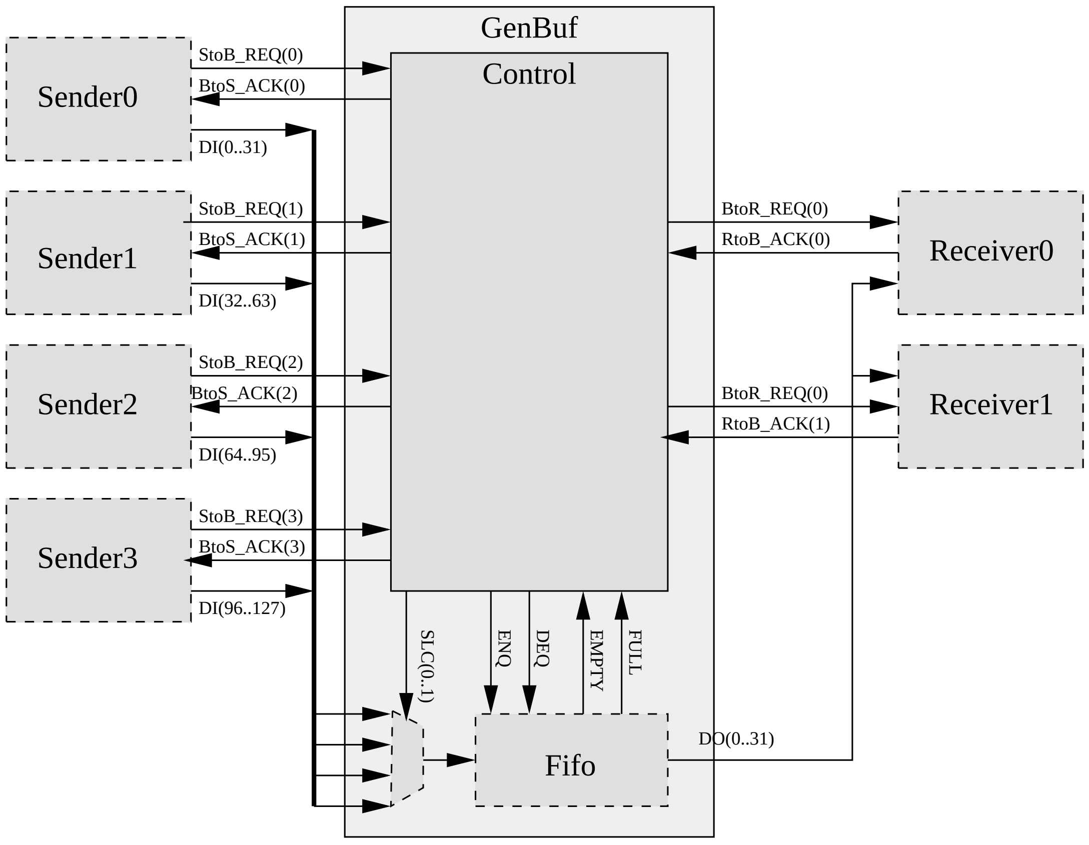

Figure 1: Block diagram of GenBuf

GenBuf connects four senders to two receivers. Data is offered by the senders in an arbitrary order, and is received by the receivers in round-robin order. The buffer has a handshake protocol with each sender and each receiver. For each sender i, GenBuf has an input StoB_REQ(i) (sender to buffer request), which signals a request to send. It also has an acknowledge output, BtoS_ACK(i) (buffer to sender acknowledge). Furthermore, each sender has a 32-bit databus to send data to the buffer. The buffer contains a four-slot FIFO to hold the data.

On the receiver side, a similar interface exists. It connects the buffer to each receiver using the output BtoR_REQ(j) (buffer to receiver request) and the input

[2]See http://www.haifa.ibm.com/projects/verification/RB_Homepage/tutorial3/.

denoted RtoB_ACK(j) (receiver to Buffer acknowledge). The receivers share a single 32-bit data bus.

Genbuf consists of a controller, a FIFO, and a multiplexer. We synthesize the controller from its specification, and we assume that the implementation of the FIFO and the multiplexer are given. FIFOs and multiplexers are standard pieces of logic and synthesizing them from specifications would make the task unnecessarily complex, especially because they involve the 32-bit data buses.

The control logic communicates with the FIFO through two outputs and two inputs. The outputs ENQ (enqueue date) and DEQ (dequeue oldest data) are used to fill and empty the FIFO. The inputs FULL and EMPTY tell the controller whether the FIFO is ready to receive or send data. The controller communicates with the multiplexer using a two-bit output called SLC determines which signal from the clients is loaded when ENQ is asserted.

3.1 Handshake between Senders and GenBuf

The interface between a sender and GenBuf is a four-phase handshake, illustrated in Figure 2:

1. When sender *i* has data to send, it initiates the transfer by raising StoB_REQ(i). One cycle later, it puts its data on the bus.
2. When the GenBuf is ready to receive the data, but at least one tick after StoB_REQ(i) is raised, it raises BtoS_ACK(i) and reads the data.
3. One tick after BtoS_ACK(i) is raised, the client lowers StoB_REQ(i). From this time on, it is no longer required to keep the data on the bus.
4. GenBuf eventually lowers BtoS_ACK(i). It may take several cycles to do so. A new transfer may not be initiated by sender *i* until one cycle after BtoS_ACK(i) is lowered.

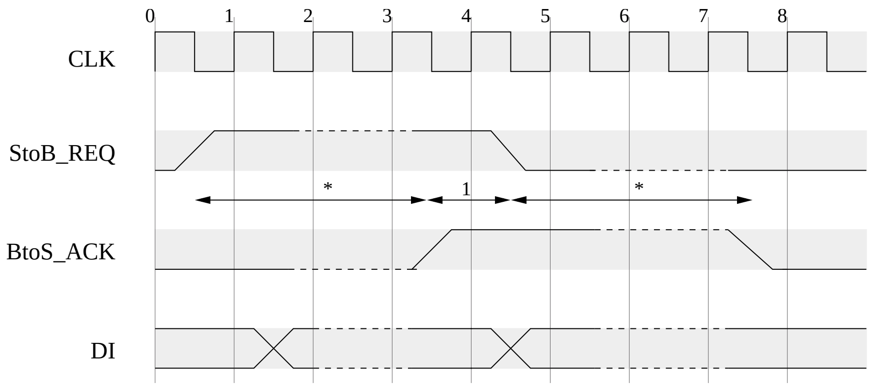

Figure 2: An example of a Sender-to-GenBuf handshake

3.2 Handshake between Receivers and GenBuf

The handshake between GenBuf and the receivers also consists of four phases, illustrated in Figure 3.

1. When GenBuf wants to send data to a receiver, say receiver j, it asserts BtoR_REQ(j) and puts the data on the D0 bus.
2. When receiver j is ready to receive data, it raises RtoB_ACK(j) and reads the data.
3. One cycle later, GenBuf lowers BtoR_REQ(j). It may now remove the data from the bus.
4. One cycle later, receiver j finalizes the transfer by lowering RtoB_ACK(j). GenBuf does not initiate another transfer until one cycle after RtoB_ACK(j) has been lowered.

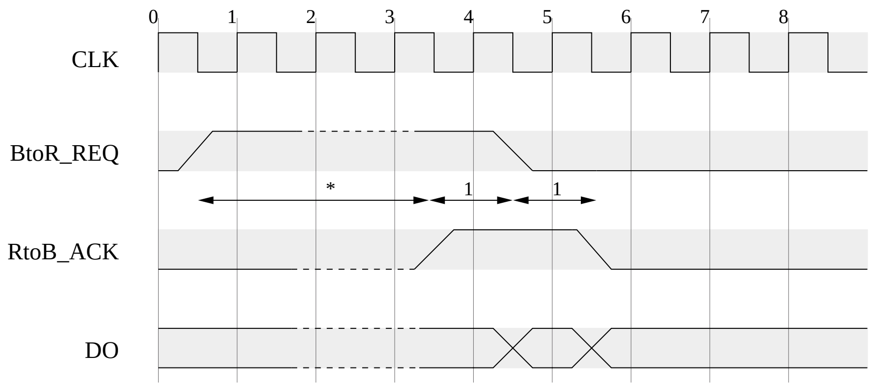

Figure 3: An example of a GenBuf-to-Receiver handshake

3.3 Formal Specification

We will now present the specification that we have developed for GenBuf. It is closely related to IBM's original specification which can be found in Section A.1 of the appendix. Since we do not synthesize the FIFO and multiplexer automatically, we have removed the specifications that stated that they work correctly and we have added formulas that specify the interaction with the FIFO and multiplexer. In the course of synthesizing GenBuf we found and corrected several mistakes in the original specification.

In the following, we use $i \in \{0,1,2,3\}$ to denote the number of a sender. We use $j \in \{0,1\}$ to denote a receiver.

Communication with Senders

Guarantee 1. *A request from a sender is always acknowledged.*

$$\forall i : \texttt{always}\ (\mathit{StoB\_REQ}(i) \rightarrow \texttt{eventually!}\ \mathit{BtoS\_ACK}(i))$$

Furthermore, the acknowledgement is eventually lowered.

$$\forall i : \texttt{always}\ (\neg \mathit{StoB\_REQ}(i) \rightarrow \texttt{eventually!}\ \neg \mathit{BtoS\_ACK}(i))$$

Guarantee 2. *Immediate acknowledges are forbidden, because the data of the sender are not valid until one step after the assertion of request.*

$$\forall i : \texttt{always}\ (\texttt{rose}(\mathit{StoB\_REQ}(i)) \rightarrow \neg \mathit{BtoS\_ACK}(i))$$

Guarantee 3. *There is no acknowledgement without a request.*

$$\forall i : \texttt{always}\ (\texttt{rose}(\mathit{BtoS\_ACK}(i)) \rightarrow \texttt{prev}(\mathit{StoB\_REQ}(i)))$$

Guarantee 4. *An acknowledge is not deasserted unless the sender deasserts its request first.*

$$\forall i : \texttt{always}\ ((\mathit{BtoS\_ACK}(i) \wedge \mathit{StoB\_REQ}(i)) \rightarrow \texttt{next!}\ \mathit{BtoS\_ACK}(i))$$

Assumption 1. *A request is not lowered until it is served.*

$$\forall i : \texttt{always}\ (\mathit{StoB\_REQ}(i) \wedge \neg \mathit{BtoS\_ACK}(i) \rightarrow \texttt{next!}\ \mathit{StoB\_REQ}(i))$$

The signal StoB_REQ(i) is lowered one cycle after BtoS_ACK(i) is raised and it cannot be raised until one cycle after BtoS_ACK(i) is lowered.

$$\forall i : \texttt{always}\ (\mathit{BtoS\_ACK}(i) \rightarrow \texttt{next!}\ \neg \mathit{StoB\_REQ}(i))$$

Guarantee 5. *Only one sender sends data at any one time.*

$$\forall i \forall i' \neq i : \texttt{always}\ \neg(\mathit{BtoS\_ACK}(i) \wedge \mathit{BtoS\_ACK}(i'))$$

Communication with Receivers

Assumption 2. *A request from the buffer is always acknowledged.*

$$\forall j : \texttt{always}\ (\mathit{BtoR\_REQ}(j) \rightarrow \texttt{eventually!}\ \mathit{RtoB\_ACK}(j))$$

Furthermore, the acknowledgement is lowered one tick after the request is lowered.

$$\forall j : \texttt{always}\ (\neg \mathit{BtoR\_REQ}(j) \rightarrow \texttt{next!}\ \neg \mathit{RtoB\_ACK}(j))$$

Assumption 3. *An acknowledgement is not deasserted unless the buffer deasserts* *its request first.*

$$\forall j : \texttt{always}\ (BtoR\_REQ(j) \wedge RtoB\_ACK(j) \rightarrow \texttt{next!}\ (RtoB\_ACK(j)))$$

Assumption 4. *There is no acknowledgement without a request.*

$$\forall j : \texttt{always}\ (RtoB\_ACK(j) \rightarrow \texttt{prev}(BtoR\_REQ(j)))$$

Guarantee 6. *A request is not lowered until it is served.*

$$\forall j : \texttt{always}\ (BtoR\_REQ(j) \wedge \neg RtoB\_ACK(j) \rightarrow \texttt{next!}\ BtoR\_REQ(j))$$

The request is lowered one cycle after the acknowledgement is raised and it cannot be raised until one cycle after the acknowledgement is lowered.

$$\forall j : \texttt{always}\ (RtoB\_ACK(j) \rightarrow \texttt{next!}\ \neg BtoR\_REQ(j))$$

Guarantee 7. *GenBuf does not request both receivers simultaneously.*

$$\texttt{always}\ \neg(BtoR\_REQ(0) \wedge BtoR\_REQ(1)).$$

GenBuf will not make two consecutive requests to any receiver. (This guarantees round-robin scheduling.)

$$\forall j : \texttt{always}\ (\texttt{rose}(BtoR\_REQ(j) \rightarrow \texttt{next!}$$
$$\mathsf{next\_event!}\ (\texttt{rose}(BtoR\_REQ(0)) \vee \texttt{rose}(BtoR\_REQ(1))(\neg BtoR\_REQ(j)))).$$

Guarantee 8. *GenBuf will deassert its request to receiver j one cycle after receiver j acknowledged the request.*

$$\forall j : \texttt{always}\ (RtoB\_ACK(j) \rightarrow X(\neg BtoR\_REQ(j)))$$

Interface to the FIFO and the Multiplexer

Guarantee 9. *The select and enqueue signals follow the acknowledgements to the senders.*

$$\texttt{always}\ (ENQ \leftrightarrow \exists i : rose(BtoS\_ACK(i)))$$
$$\forall i : \texttt{always}\ (rose(BtoS\_ACK(i) \rightarrow SLC = i)$$

Guarantee 10. *Data is dequeued when the transfer to the receiver has completed.*

$$\texttt{always}\ (DEQ \leftrightarrow (\texttt{fell}(RtoB\_ACK(0)) \vee \texttt{fell}(RtoB\_ACK(1)))$$

Guarantee 11. *No enqueue when the FIFO is full and we do not dequeue data,* *and no dequeue when it is empty.*

$$\texttt{always}\ ((\mathit{FULL} \wedge \neg \mathit{DEQ}) \rightarrow \neg \mathit{ENQ})$$
$$\texttt{always}\ (\mathit{EMPTY} \rightarrow \neg \mathit{DEQ})$$

Guarantee 12. *If the FIFO is not empty, a dequeue will ensue eventually.*

$$\texttt{always}\ (\neg \mathit{EMPTY} \rightarrow \texttt{eventually!}\ (\mathit{DEQ}))$$

Assumption 5. *The FIFO behaves correctly. If we enqueue and dequeue the same number of data the status of the FIFO does not change. If data are only enqueued (dequeued resp.), the FIFO must not be empty (full) in the next cycle.*

$$\texttt{always}\ ((\mathit{DEQ} \leftrightarrow \mathit{ENQ}) \rightarrow (\mathit{EMPTY} \leftrightarrow \texttt{next!}\ \mathit{EMPTY}))$$
$$\texttt{always}\ ((\mathit{DEQ} \leftrightarrow \mathit{ENQ}) \rightarrow (\mathit{FULL} \leftrightarrow \texttt{next!}\ \mathit{FULL}))$$
$$\texttt{always}\ ((\mathit{ENQ} \wedge \neg \mathit{DEQ}) \rightarrow \texttt{next!}\ \neg \mathit{EMPTY})$$
$$\texttt{always}\ ((\mathit{DEQ} \wedge \neg \mathit{ENQ}) \rightarrow \texttt{next!}\ \neg \mathit{FULL})$$

3.4 Synthesis

As explained in Section 2.2, not all PSL specifications can be synthesized directly. We first have to translate the formulas of Guarantee 1, 2, 7, 12 and Assumption 2 into a format suitable for the GR(1) approach.

Taking the Guarantee 4, 6, and Assumption 4 into account, Guarantee 1, 2 and Assumption 2 can simply be rewritten to match the form `always eventually!` B_i where B_i is a Boolean formula. More precisely, Guarantee 1 and 2 are combined and rewritten to

$$\forall i : \texttt{always eventually!}\ (\text{StoB\_REQ}(i) \leftrightarrow \text{BtoS\_ACK}(i)),$$

and we rewrite Assumption 2 to

$$\forall j : \texttt{always eventually!}\ (\text{BtoR\_REQ}(i) \leftrightarrow \text{RtoB\_ACK}(i)).$$

For Guarantee 12 and the second part of Guarantee 7 we have to build deterministic monitors. Although there are formulas for which no deterministic monitor exists, and constructing such monitors is hard in general [KV98], constructing them is very simple for the formulas considered in this document.

For instance, Figure 4 shows the deterministic automaton for Guarantee 12 stating that $\texttt{always}\ (\neg\text{EMPTY} \rightarrow \texttt{eventually!}\ (\text{DEQ}))$. We used the standard approach

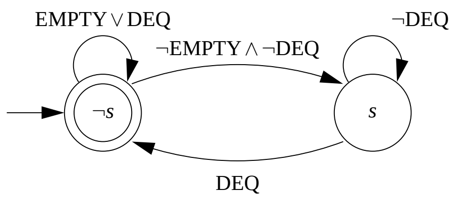

Figure 4: Monitor for Guarantee 12

to construct Büchi automata from LTL formulas (e.g., [SB00]) with a slightly modified form of the standard expansion rules. In particular, we used the expansion rule `eventually!` q equals $q \vee (\neg q \wedge$ `eventually!` $q)$ and the fact that $\neg\mathrm{EMPTY} \rightarrow \varphi$ equals $\mathrm{EMPTY} \vee (\neg\mathrm{EMPTY} \wedge \varphi)$.

Note that deterministic automata are easily represented in PSL by a set of formulas: (1) For each edge of the automaton one formula of the form `always` $(s \wedge i \rightarrow$ `next!` $(s'))$, where s and s' identify states and i is an input, (2) a Boolean formula B representing the initial state, and (3) to encode the fairness condition a formula of the form `always eventually!` (B) where B is a Boolean formula representing a set of states.

After the specification has been brought into the proper form (see Appendix A.2), it is synthesized using the algorithm of [PPS06b]. Subsequently, a circuit is constructed using the techniques described in Section 5, optimized using SIS' `script.rugged` and then mapped by SIS using `stdcell2_2` [SSM+92]. Table 1 shows the time used to synthesize the buffer and the size of the resulting design before and after optimization with `script.rugged`, given in SIS standard-cell grid count.

Table 1: Time to synthesize Genbuf and size of resulting design

Results for generalized buffer	
Overall Time	1.91 sec
Before `script.rugged`:	
Gates	1888
Area (in SIS standard-cell grid count)	77896
Delay	84.82
After `script.rugged`:	
Gates	1893
Area (in SIS standard-cell grid count)	62672
Delay	60.22

4 AMBA AHB

4.1 The AMBA AHB Protocol

ARM's *Advanced Microcontroller Bus Architecture* (AMBA) consists of several buses [ARM99]. The one that concerns us here is the *Advanced High-Performance Bus* (AHB). The AHB is an on-chip communication standard connecting such devices as processor cores, cache memory, DMA controllers, DSPs, and the interface to off-chip memory. Up to 16 *masters* and up to 16 *slaves* are connected to the bus. The masters initiate communication (read or write) with a slave of their choice. Slaves are passive and can only respond to a request. Master 0 is the *default master* and is selected whenever there are no requests for the bus.

The AHB is a pipelined bus. This means that different masters can be in different stages of communication. At one instant, multiple masters can request the bus, while another master transfers address information, and a yet another master transfers data. A bus *access* can be a single *transfer* or a *burst*, which consists of a specified or unspecified number of transfers. Access to the bus is controlled by the *arbiter*, which is the subject of the current paper. All devices that are connected to the bus are Moore machines, that is, the reaction of a device to an action at time t can only be seen by the other devices at time $t+1$.

The AMBA standard leaves many aspects of the bus unspecified. The protocol is at a logic level, which means that timing and electric parameters are not specified. Apart from that, aspects such as the arbitration protocol are also left unspecified.

We will now introduce the signals used in the AHB. The notation S[n:0] denotes an $(n+1)$-bit signal.

- HBUSREQi – A request from master *i* to access the bus. Driven by the masters.
- HLOCKi – A request from master *i* to receive a locked (uninterruptible) access to the bus. (Raised in combination with HBUSREQi.) Driven by the masters.
- HMASTER [3:0] – The master that currently owns the address bus (binary encoding). Driven by the arbiter.
- HREADY – High if the slave has finished processing the current data. Change of bus ownership and commencement of transfers only takes place when HREADY is high. Driven by the slave.

- HGRANTi – Signals that if HREADY is high, HMASTER $= i$ will hold in the next tick. Driven by the arbiter.
- HMASTLOCK – Indicates that the current master is performing a locked access. If this signal is low, a burst access may be interrupted when the bus is assigned to a higher-priority master. Driven by the arbiter

The following set of signals is multiplexed using HMASTER as the control signal. Thus, although every master has an address bus, only the address provided by the currently active master is visible on HADDR.

- HADDR[31:0] – The address for the next transfer. The address determines the destination slave.
- HBURST[1:0] – One of SINGLE (a single transfer), BURST4 (a four-transfer burst access), or INCR (unspecified length burst).

The list of signals does not contain the data transfer signals as these do not concern the arbiter. (Ownership of the data bus follows ownership of the address bus in a simple, delayed manner.) Bursts of length 8 or 16 are not taken into account, nor are the different addressing types for bursts. Adding longer bursts only lengthens the specification and the addressing types do not concern the arbiter. Furthermore, as an optional feature of the AHB, a slave is allowed to interrupt a burst access and request that it be continued later; we have left this feature out.

A typical set of accesses is shown in Fig. 5. (Please ignore the DECIDE and START signals for now.) At time 1, masters 1 and 2 request an access. master 1 requests a locked transfer. The access is granted to master 1 at the next time step, and master 1 starts its access at time 3. Note that HMASTER changes and HMASTLOCK goes up. The access is a BURST4, which can not be interrupted. At time step 6, when the last transfer in the burst starts, the arbiter prepares to hand over the bus to master 2 by changing the grant signals. However, HREADY is low, so the last transfer is extended and the bus is only handed over in time step 8, after HREADY has become high again.

4.2 Specification

This section contains the specification of the arbiter. To simplify the specification, we have added three auxiliary variables, START, LOCKED, and DECIDE, which are driven by the arbiter. Signal START indicates the start of an access. (A single transfer, a four-transfer burst, or an unspecified length burst.) In Fig. 5, for instance, START is high in Steps 3 and 8 and low otherwise. The master only switches when START is high. The signal LOCKED indicates if the bus will be locked at the next start of an access. Signal DECIDE is described below (in Paragraph “Deciding the Next Access”.)

We group the properties into three sets. The first set of properties defines when a new access is allowed to start. The second describes how the bus has to be handed

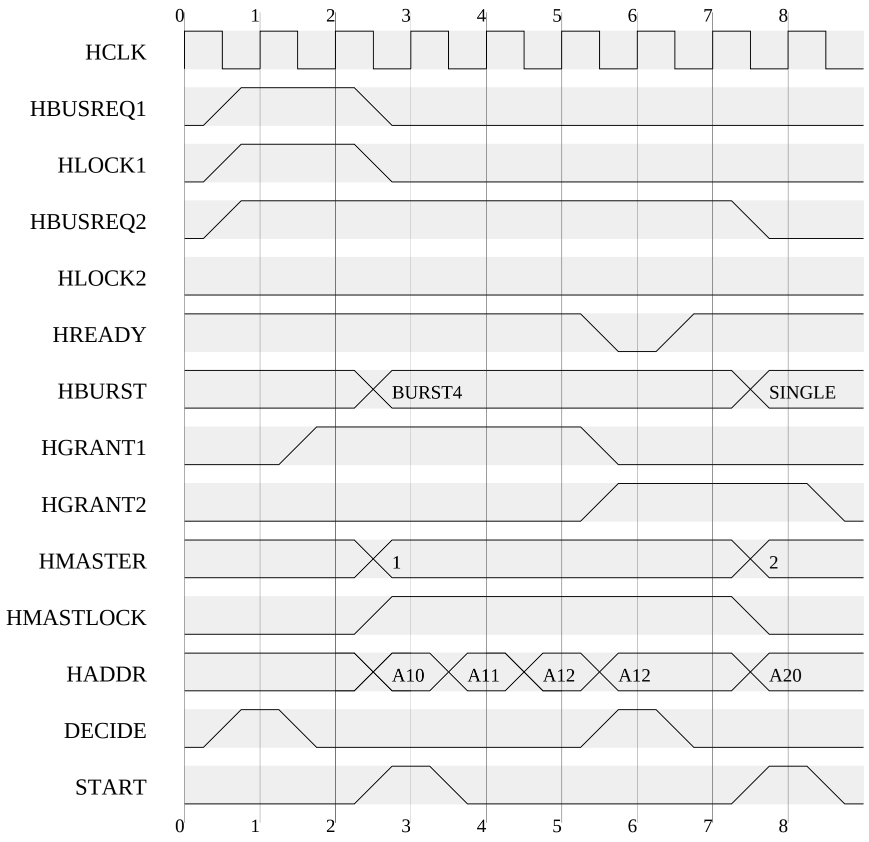

Figure 5: An example of AMBA bus behavior

over, and the third describes which decisions the arbiter makes. We distinguish *guarantees*, which are properties that the arbiter must fulfill, and *assumptions*, which are properties that the arbiter's environment must fulfill.

Starting an Access

Assumption 1. *During a locked unspecified length burst, leaving HBUSREQi high locks the bus. This is forbidden by the standard, and we assume that the environment obeys this rule.*

$$\forall i : \texttt{always}\ (\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{INCR} \wedge \\ \mathit{HMASTER} = i \rightarrow \texttt{next!}\ \texttt{eventually!}\ \neg \mathit{HBUSREQi})$$

Assumption 2. *Leaving HREADY low locks the bus, the standard forbids it.*

$$\texttt{always}\ \texttt{eventually!}\ \mathit{HREADY}$$

Assumption 3. *The lock signal is asserted by a master at the same time as the bus request signal.*

$$\forall i : \texttt{always}\ (\mathit{HLOCKi} \rightarrow \mathit{HBUSREQi})$$

Guarantee 1. *A new access can only start when HREADY is high.*

$$\texttt{always}\ (\neg \mathit{HREADY} \rightarrow \texttt{next!}\ \neg \mathit{START})$$

Guarantee 2. *When a locked unspecified length burst starts, a new access does not start until the current master (i) releases the bus by lowering HBUSREQi.*

$$\begin{aligned}\forall i : \texttt{always}\ (&\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{INCR} \wedge \mathit{START} \wedge \\ &\mathit{HMASTER} = i \rightarrow \texttt{next!}\ \neg \mathit{START}\ \texttt{until\_}\ \neg \mathit{HBUSREQi})\end{aligned}$$

Guarantee 3. *When a locked burst starts, no other accesses start until the end of the burst. We can only transfer data when HREADY is high, so the current burst ends at the fourth occurrence of HREADY.*

$$\begin{aligned}\forall i : \texttt{always}\ (&\mathit{HMASTLOCK} \wedge \mathit{HBURST} = \mathit{BURST4} \wedge \mathit{START} \rightarrow \\ &(\mathit{HREADY} \wedge \texttt{next!}\ (\neg \mathit{START}\ \texttt{until\_}\,[3]\ \mathit{HREADY})) \vee \\ &(\neg \mathit{HREADY} \wedge (\texttt{next!}\ \neg \mathit{START}\ \texttt{until\_}\,[4]\ \mathit{HREADY})))\end{aligned}$$

Granting the Bus

Guarantee 4. *The HMASTER signal follows the grants: When HREADY is high, HMASTER is set to the master that is currently granted. This implies that no two grants may be high simultaneously when HREADY is high. (Which means never, because the arbiter does not control HREADY.) It also implies that we cannot change HMASTER without giving a grant.*

$$\forall i : \texttt{always}\ (\mathit{HREADY} \rightarrow (\mathit{HGRANTi} \leftrightarrow \texttt{next!}\ \mathit{HMASTER} = i))$$

Guarantee 5. *Whenever HREADY is high, the signal HMASTLOCK follows the signal LOCKED in the next time step.*

$$\forall i : \texttt{always}\ (\mathit{HREADY} \rightarrow (\mathit{LOCKED} \leftrightarrow \texttt{next!}\ \mathit{HMASTLOCK}))$$

Guarantee 6. *If we do not start an access in the next time step, the bus is not reassigned and HMASTLOCK does not change.*

$$\begin{aligned}\forall i : \texttt{always}\ (\texttt{next!}\ \neg \mathit{START} &\rightarrow (\mathit{HMASTER} = i \leftrightarrow \texttt{next!}\ \mathit{HMASTER} = i)) \\ \texttt{always}\ (\texttt{next!}\ \neg \mathit{START} &\rightarrow (\mathit{HMASTLOCK} \leftrightarrow \texttt{next!}\ \mathit{HMASTLOCK}))\end{aligned}$$

Deciding the Next Access

Signal DECIDE indicates the time slot in which the arbiter decides who the next master will be, and whether its access will be locked. The decision is based on HBUSREQi and HLOCKi. (For instance, DECIDE is high in Step 1 and 6 in Fig. 5.) Note that a decision is executed at the next START signal, which can occur at the earliest two time steps after the HBUSREQi and HLOCKi signals are read (see Fig. 5, the signals are read in Step 1 and the corresponding access starts at Step 3.)

Guarantee 7. *If the arbiter decides to give the bus to master i, it stores the corresponding HLOCKi signal in LOCKED to correctly lock or unlock the bus the next time an access starts (see Guarantee 5).*

$$\forall i : \texttt{always}\ ((\mathit{DECIDE} \wedge \texttt{next!}\ \mathit{HGRANTi}) \rightarrow (\mathit{HLOCKi} \leftrightarrow \texttt{next!}\ \mathit{LOCKED}))$$

Guarantee 8. *We do not change the grant and the locked signals if DECIDE is low, so the arbiter needs to make a decision to change the signals HGRANTi and LOCKED.*

$$\texttt{always}\ (\neg\mathit{DECIDE} \rightarrow \bigwedge_i (\mathit{HGRANTi} \leftrightarrow \texttt{next!}\ \mathit{HGRANTi}))$$
$$\texttt{always}\ (\neg\mathit{DECIDE} \rightarrow (\mathit{LOCKED} \leftrightarrow \texttt{next!}\ \mathit{LOCKED}))$$

Guarantee 9. *We have a fair bus. Note that this is not required by the* AMBA *standard, and there are valid alternatives, such as a fixed-priority scheme. A property like this is needed to make the example realistic. Otherwise, an arbiter need not to serve any master to fulfill the specification.*

$$\forall i : \texttt{always}\ (\mathit{HBUSREQi} \rightarrow \texttt{eventually!}\ (\neg\mathit{HBUSREQi} \vee \mathit{HMASTER} = i))$$

Guarantee 10. *We do not grant the bus without a request, except to master 0. If there are no requests for the bus, the bus is granted to Master 0. This requirement is not stated in the AMBA AHB standard but needed to ensure that an arbiter only serves masters that actually request the bus (besides the default master).*

$$\forall i \neq 0 : \texttt{always}\ (\neg\mathit{HGRANTi} \rightarrow (\mathit{HBUSREQi}\ \texttt{before}\ \mathit{HGRANTi}))$$
$$\texttt{always}\ (\mathit{DECIDE} \wedge \forall i : \neg\mathit{HBUSREQi} \rightarrow \texttt{next!}\ \mathit{HGRANT0})$$

Guarantee 11. *An access by Master 0 starts in the first clock tick and simultaneously, a decision is taken. Thus, the signals DECIDE, START, and HGRANT0 are high and all others are low.*

$$\mathit{DECIDE} \wedge \mathit{START} \wedge \mathit{HGRANT0} \wedge \mathit{HMASTER} = 0$$
$$\wedge \neg\mathit{HMASTLOCK} \wedge \forall i \neq 0 : \neg\mathit{HGRANTi}$$

Assumption 4. *We assume that all input signals are low initially.*

$$\forall i (\neg\mathit{HBUSREQi} \wedge \neg\mathit{HLOCKi}) \wedge \neg\mathit{HREADY}$$

4.3 Synthesis

First we have to build deterministic monitors for the formulas A1, G2, G3, and G10. Again we use the slightly modified expansion rules presented in Section 3.4.

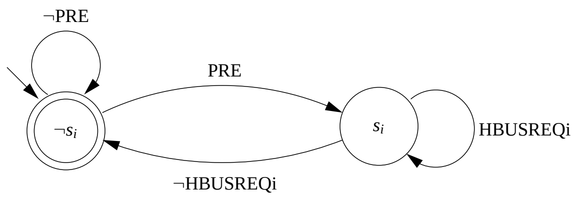

Figure 6: Deterministic Monitor for Assumption 1 for one master

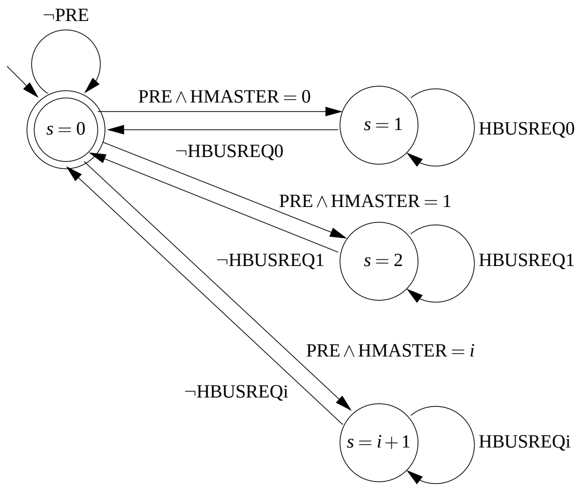

Figure 7: Deterministic Monitor for Assumption 1 for all masters

Assumption 1
Let PRE be the conjunct $\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \wedge \text{HMASTER} = i$, then the formula for Assumption 1 for a master i equals

$$\texttt{always}\,(\neg\text{PRE} \vee \texttt{next!}\ \texttt{eventually!}\ \neg\text{HBUSREQi}).$$

Figure 6 shows the deterministic monitor we build for each master. Two concentric circles denote an accepting state. Merging the monitors for different masters into a single one yields the monitor shown in Figure 7.

Guarantee 2, 3, and 10
Using the same techniques we build the deterministic monitors shown in Figure 8, 9, and 10. A complete specification of the arbiter for two masters suitable for the GR(1)-approach can be found in the Appendix B.1.

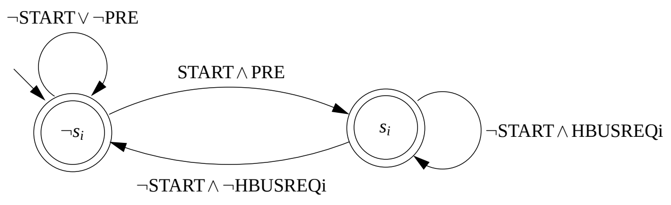

Figure 8: Monitor for Guarantee 2 for one master

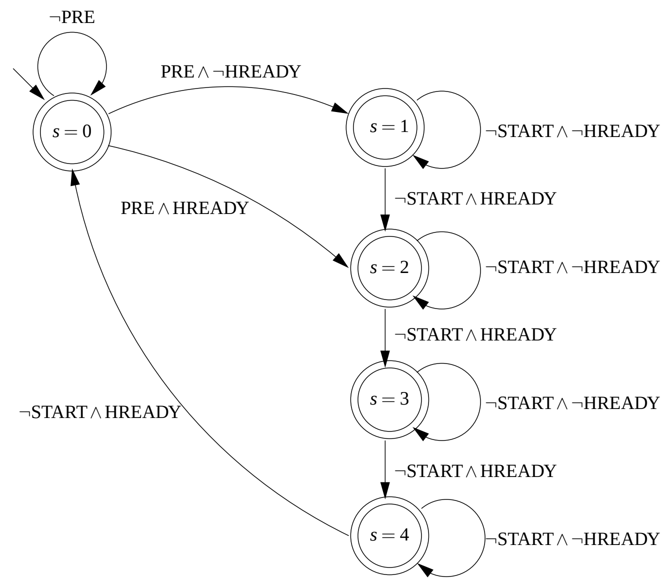

Figure 9: Monitor for Guarantee 3

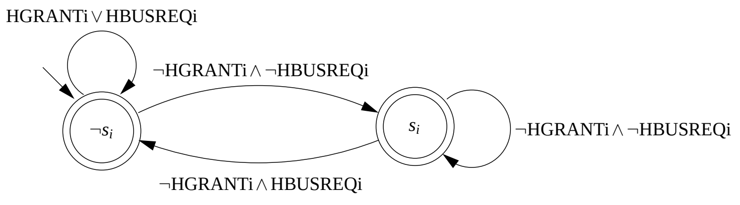

Figure 10: Monitor for Guarantee 10 for one master

Table 2: Time to synthesize arbiters for one to four masters and size of resulting designs

	Handwritten	**1 Master**	**2 Masters**	**3 Masters**	**4 Masters**
Overall time	-	0.7 sec	7.1 sec	203.4 sec	2689.0 sec
Before `script.rugged`:					
Gates	1004	115	1426	3489	-
Area	35264	4440	49848	120064	-
Delay	81.22	24.02	70.84	132.22	-
After `script.rugged`:					
Gates	246	72	1015	2475	-
Area	10792	2936	32432	79128	-
Delay	24.64	16.84	45.24	63.02	-

Once the specification is in a proper format, the arbiter is synthesized using the algorithm of [PPS06b] and a circuit is constructed as described in Section 5, optimized using SIS' `script.rugged` and then mapped by SIS using `stdcell2_2` [SSM+92].

Table 2 shows the time used to synthesize arbiters for one to four masters and the size of the resulting designs before and after optimization with `script.rugged`, given in SIS standard-cell grid count. Synthesis for a master with 1 client takes 0.7s (time spent by SIS not included) and yields a circuit of size 2.9k (SIS standard-cell grid count) with a delay of 17. For two Masters we have 7.1s, size 32k, and delay 45. For three masters we get 203.4s, size 79k, delay 63. For four masters we need around 2700s and SIS is unable to map the circuit. Minimization by SIS yielded an improvement in size of one third throughout. In contrast, a manual implementation for 16 masters has size 11k and delay 25.

The automatically generated arbiter implements a round-robin arbitration scheme. This can be explained from the construction of the strategy in the synthesis algorithm, but it is also the simplest implementation of a fair arbiter. We have validated our specification by combining the resulting arbiter with manually written masters and clients, with which it cooperates without problems.

5 Generating a Circuit

We have implemented the synthesis approach in a tool called Anzu, which relies heavily on the CUDD package [Som].

The synthesis approach presented in [PPS06b] constructs a strategy from which a sequential circuit can be constructed. The strategy, represented as a BDD, is a relation between the inputs and the current states on the one side and the outputs and next states on the other.

In [KS00], Kukula and Shiple present an approach to construct a circuit from a relation. A relation allows for several compatible behaviors. Kukula and Shiple construct one circuit that implements all allowed behavior and use an additional set of input signals to choose between the different behaviors. In synthesis we are interested in a single valid behavior, so we fixed the additional input signals to arbitrary values. Initial experiments using this approach to generate combinational logic from the strategy yielded extremely large circuits. So we followed a completely different approach. Our current approach generates combinational logic using the following pseudo code, where `S` is the Strategy and `O`$\backslash$`o` denotes set of combinational outputs excluding output `o`.

We refer to [Bry92] for a detailed description of BDDs and the opearators used to manipulate them below.

```
for all combinational outputs o do
  S′ = exists O\o . S
  p = positive cofactor of o in S′
  n = negative cofactor of o in S′
  // note: p and n in general incomparable
  careset = p * !n + !p * n
  f[o] = p minimized wrt. careset
  // keep relation between outputs
  S = S in which o is substituted by f[o]
od
```

The result is an array `f` of BDDs, which is written to a file using CUDD's DumpBlif command and then converted to VERILOG. This approach, in combination with minimization of the strategy with respect to the reachable states yields an improvement of more than an order of magnitude in the size of the resulting circuit.

6 Discussion

In this section we give an overview of the most important benefits and drawbacks of automatic synthesis of the arbiter, as we perceive them.

Writing a complete formal specification for a circuit is not trivial. First, many aspects of the circuit are not defined and need to be inferred from the apparent design intent. This problem was apparent when we synthesized the AMBA arbiter, as its specifications are quite informal and incomplete. In particular, we were continuously asking ourselves how the implementor of the *environment* could interpret the specifications, and whether the arbiter should handle every such interpretation. In our opinion, it would be very beneficial to have a formal specification early on, so that these issues are resolved before implementation is started.

Ambiguity was less of a problem for GenBuf, where a good specification in PSL had already been constructed. We would like to point out that ambiguities also have to be resolved when one writes a hand implementation, and that writing a good set of properties is a prerequisite for effective verification.

Second, it is not always trivial to translate an informal specification to formulas. When specifying the arbiter, one of the important insights was that two additional signals, START and DECIDE, were needed. This problem also occurs when we attempt to formally verify a manually coded arbiter, where the same signals would be useful. (In fact, these signals occur, in one form or other, in our manual implementation as well.)

The AMBA arbiter was parameterized by the number of masters and clients. Likewise, it is easy to imagine a version of GenBuf that is parameterized by the number of senders and receivers. These parameters have a very small influence on the size of a manual implementation or on effort needed to produce it. The same holds for the formal specification, but not for the process or the result of synthesis. The time to synthesize the arbiter grows quickly with the number of masters as does the size of the generated circuit. Unfortunately, the generated gate-level output is complicated and cannot be changed by hand. The resulting circuit can likely be improved further by using more intelligent methods to generate the circuits, which will be important if this methodology is to become accepted. The problem is related to synthesis of partially specified functions [HS96] with the important characteristic that the space of allowed functions is very large.

On the upside, when designers use automatic synthesis, they no longer have to be concerned with the VERILOG implementation, they can concentrated solely on the PSL specification. The PSL specifications that we constructed for our two examples are short, readable, and easy to modify, much more so than a manual implementation. For the arbiter in particular, we expect that it is easier to learn the way it functions from the formal specifications than from a manual VERILOG implementation.

It should be noted that automatic synthesis is first and foremost applicable to control circuitry. We are looking into methods to beneficially combine manually coded data paths with automatically synthesized control circuitry.

Although this approach removes the need for verification of the resulting circuit, the specification itself still needs to be validated. The lack of tools for debugging specifications was apparent in our exercise. Some work on such tools has taken place [PSC+06], but further research, in particular in connection with realizability, is needed.

7 Conclusions

When specifications are available early, automatic synthesis can be used to obtain a first implementation, yielding a functional test environment when critical blocks are replaced by manual implementations. Furthermore, these implementations function as a valuable sanity check for the specification, which is very important when the implementation is based on the formal specification.

Although automatic synthesis has long been pursued, only recent developments have made it applicable to industrial examples. This deliverable presents the first time that a real-life block has been synthesized from its specification. Automatic synthesis is still very young and only a few avenues for optimization have been pursued. Our simple algorithm to generate circuits , for instance, yielded an improvement of more than an order of magnitude (see Section 5). We expect that future research will yield further large improvements in the size of the systems that can be synthesized, making automatic synthesis a real alternative to manual coding of some types of circuits.

Acknowledgments
We would like to thank Nir Piterman for helping with the AMBA case study. We would like to thank Karin Greiml and Milan Milinkovic for their help implementing Anzu. Finally, we would like to thank Colin Campbell, Margaret Rugira, and Chris styles from ARM’s Product Support Team for their help figuring out AHB protocol. (We would like to note that ARM does not in any way endorse the contents of this document.)

8 References

[ARM99] ARM Ltd. AMBA Specification (Rev. 2). Available from www.arm.com, May 1999.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

[Chu62] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.

[Chu63] A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, Upsala, 1963.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer-Verlag, 2006.

[HRS05] A. Harding, M. Ryan, and P. Schobbens. A new algorithm for strategy synthesis in LTL games. In *Tools and Algorithms for the Construction and the Analysis of Systems (TACAS'05)*, pages 477–492, 2005.

[HS96] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, MA, 1996.

[JB06] B. Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.

[KS00] J. H. Kukula and T. R. Shiple. Building circuits from relations. In E. Allen Emerson and A. Prasad Sistla, editors, *12th Conference on Computer Aided Verification (CAV'00)*, pages 113–123, 2000. LNCS 1855.

[KV98] O. Kupferman and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[PPS06a] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, pages 364–380, 2006.

[PPS06b] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proc. 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lect. Notes in Comp. Sci.*, pages 364–380. Springer-Verlag, 2006.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.

[PSC$^+$06] Ingo Pill, Simone Semprini, Roberto Cavada, Marco Roveri, Roderick Bloem, and Alessandro Cimatti. Formal analysis of hardware requirements. In *Design Automation Conference*, pages 821–826, 2006.

[Rab72] M.O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics*. Amer. Math. Soc., 1972.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[Som] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[SSM$^+$92] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, pages 328–333, Cambridge, MA, October 1992.

A GenBuf Specification

IBM's original specifications consists of the twelve rules shown in Section A.1. In Table 3 we state for each part of our specification the corresponding part of IBM's original specification.

Table 3: Guarantees/Assumptions and corresponding IBM's rules

Guarantee/ Assumption	*Rule*
G1	R1, R5.A
G2	Stated only in the description of the handshake protocol
G3	R3.A
G4	R5.B
G5	R6
G6	Stated only in the description of the handshake protocol
G7	R4.B, R4.C, R4.D
G8	R7, R4.A
G9	Constraints the interface to the FIFO
G10	Constraints the interface to the FIFO
G11	R8, R9
G12	R10
A1	Stated only in the description of the handshake protocol
A2	R2
A3	Stated only in the description of the handshake protocol
A4	R3.B
A5	States that the FIFO behaves correctly

Note that rules constraining the data path (R4.E, R11, and R12) are not taken into account.

A.1 Original Specification

```
R1: Sender request get acknowledged

vunit sender_req_get_ack
{
"Sender request get acknowleged"
%for i in 0..3 do
    assert always (StoB_REQ(%{i}) -> eventually! (BtoS_ACK(%{i})));
%end
```

```
}
--------------------------------------------------------------------
R2: GenBuf requests get acknowledged

vunit genbuf_req_get_ack
{
"GenBuf requests eventually get acknowledged"
%for i in 0..1 do
    assert always (BtoR_REQ(%{i}) -> eventually! ( RtoB_ACK(%{i})));
%end
}

--------------------------------------------------------------------
R3.A/B: No acknowledge without a request

vunit no_ack_without_a_req
{
"GenBuf does not acknowledge senders unless requested"
%for i in 0..3 do
    assert
      always (rose(BtoS_ACK(%{i})) -> prev(StoB_REQ(%{i})));
%end

%for j in 0..1 do
    assert
      always ((RtoB_ACK(%{j})) -> prev(BtoR_REQ(%{j})));
%end
}

These assertions mean that an acknowledge signal can only be on if the
appropriate request signal was also on during the previous cycle.
--------------------------------------------------------------------
R4.A/B/C/D/E:
vmode default { define B2R_REQ := BtoR_REQ(0) | BtoR_REQ(1) ; }

vunit receiver_ack
{
"Several receivers acknowledge properties"

%for i in 0..1 do


      assert "GenBuf will deassert its request to receiver
        %{i} a cycle after receiver %{i} acknowledged the request"

      always (RtoB_ACK(%{i}) -> next(!BtoR_REQ(%{i})));
%end
```

```
  assert "GenBuf does not request both receivers at the same time"
    always !(BtoR_REQ(0) & BtoR_REQ(1));

  assert "GenBuf will not make two consecutive requests to receiver 0"
    always (rose(BtoR_REQ(0)) -> next next_event(rose(B2R_REQ))( BtoR_REQ(1)));

  assert "GenBuf will not make two consecutive requests to receiver 1"
    always (rose(BtoR_REQ(1)) -> next next_event(rose(B2R_REQ))( BtoR_REQ(0)));

  assert "GenBuf deasserts BtoR_REQ in the cycle it puts the data on bus"
    always (rose(DO(0..31)!= 0) -> fell(B2R_REQ));
}
-------------------------------------------------------------------
R5.A/B: GenBuf - sender handshake

vunit genbuf_to_sender_handshake
{
"GenBuf to sender handshake"

%for i in 0..3 do

  assert "In case the sender deasserts its request GenBuf will
    eventually deassert its acknowledgement"
    always ( !StoB_REQ(%{i}) -> eventually! ( !BtoS_ACK(%{i})));

  assert "GenBuf will not deassert its acknowledgement unless the
    sender first deasserted its request"
    always ( BtoS_ACK(%{i}) -> ( BtoS_ACK(i) until_ (!StoB_REQ(%{i}))));
%end
}
-------------------------------------------------------------------
R6: Only one sender at a time

vunit only_one_sender_at_a_time
{
  "Only one sender can send data at any given time."

  %for i in 0..3 do
     %for j in 0..3 do
        %if i != j %then
           assert "The %{i} th and %{j} th senders do not send together"
           always ( !(BtoS_ACK(%{i}) & BtoS_ACK(%{j})));
        %end
     %end
  %end
```

```
}
-------------------------------------------------------------------
R7: Only one receiver at a time

vunit only_one_receiver_at_a_time
{
"Only one receiver can receive data at any given time."

    assert "Only one receiver can receive data at any given time"
      always ( !(BtoR_REQ(0) & BtoR_REQ(1)));
}
-------------------------------------------------------------------
R8: GenBuf does not receive when the queue is full

vmode Queue
{
-- This is a mode keeping track of buffers queue (which is of depth 4)


  define rose_B2S_ACK := rose(BtoS_ACK(0)) | rose(BtoS_ACK(1))
                                   | rose(BtoS_ACK(2)) | rose(BtoS_ACK(3)) ;

  define rose_R2B_ACK := rose(RtoB_ACK(0)) | rose(RtoB_ACK(1)) ;

  define rose_B2R_REQ := rose(BtoR_REQ(0)) | rose(BtoR_REQ(1)) ;

  var old_QC : -1..5 ;
  var Q_counter : -1..5 ;

  assign
         init(old_QC) := 0 ;
         next(old_QC) := Q_counter ;

  assign
         Q_counter := /* Note : Q_counter is assigned NOT next( Q_counter ) */
                case
                  RST :0 ;
                  old_QC=-1                          :-1 ; /* Underflow */
                  old_QC=5                           : 5 ;  /* Overflow */
                  rose_B2S_ACK & prev(rose_R2B_ACK) : old_QC ;
                                                   /* Both read & write */
                  rose_B2S_ACK                       : old_QC + 1 ;
                  prev(rose_R2B_ACK)                 : old_QC-1 ;
                  else                               : old_QC ;
                esac ;
}
```

```
vunit genbuf_does_not_receive_when_queue_full
{
"GenBuf does not receive when the queue is full"

inherit Queue;

%for i in 0..3 do
    assert "GenBuf does not acknowledge sender %{i} when the queue is full"
      always( rose( BtoS_ACK(%{i}) ) -> prev(Q_counter <=3));
%end
}
------------------------------------------------------------------
R9: GenBuf does not send when the queue is empty

vunit genbuf_does_not_send_when_queue_empty
{
"non of BtoS_ACK will assert when queue is full"

inherit Queue;

%for i in 0..1 do
    assert "GenBuf does not request receivers when the queue is empty"
      always (rose(BtoR_REQ(%{i})) -> prev( Q_counter > 0 ));

%end
}
------------------------------------------------------------------
R10: Data read is eventually written

vmode default
{
  var xx(0..1) : boolean;
  assign next(xx(0..1)) := xx(0..1);

  define data_x_is_written :=
       ((fell(BtoR_REQ(0)) | fell(BtoR_REQ(1))) & (DO(0..1)=xx(0..1)));

%for i in 0..3 do
    define  data_x_is_read_from_%{i} :=
        ( rose(BtoS_ACK(i)) & (DI(%{i}*32..%{i}*32+1)=xx(0..1)));

    /* Note : We "cheat" on this point. The bus is read by GenBuf one cycle
              before the rise of BtoS_ACK(i). However, since the sender
              does not release the bus until later - this will do for the
              tutorial. */
```

```
%end

  define data_x_is_read :=  ( data_x_is_read_from_0 |  data_x_is_read_from_1
                            | data_x_is_read_from_2 |  data_x_is_read_from_3);
}

vunit data_read_is_eventually_written
{
"Every data word read by GenBuf will be eventually sent out to a receiver"
%for i in 0..3 do
    assert "if sender %{i}  is read and has data then data is
      eventually written"
      always (data_x_is_read_from_%{i} -> eventually!( data_x_is_written));
%end
}
-------------------------------------------------------------------
R11: GenBuf keeps FIFO order

vunit genbuf_keeps_fifo_order
{
"GenBuf is queue keeps order i.e. GenBuf is indeed FIFO"

  inherit Queue;
  define some_data_is_written := (fell(BtoR_REQ(0)) | fell(BtoR_REQ(1)));

  %for j in 1..4 do
    assert "If data is read when the queue is at depth %{j} then the
    data will be written in %{j} write operations"
      always (data_x_is_read & (Q_counter = j)
               -> next(next_event(
                      some_data_is_written )[j](DO(0..1)=xx(0..1))));
%end
}
-------------------------------------------------------------------
R12: GenBuf writes only what it read before

-- We cheat to avoid explosion
vmode stuck_input_on_0
{
%for i in 0..3 do
    define DI(%{i}*32..%{i+1}*32-1) := 0H ;
%end
}

vunit genbuf_is_not_creative
{
```

```
"all the data sent out of GenBuf was once read into GenBuf"

  inherit stuck_input_on_0 ;

  %for j in 0..1 do
    assert "Data sent to receivier %{j} is not invented by GenBuf"
        always (fell(BtoR_REQ(%{j})) -> (DO(0..31)=0H) );
  %end
}
```

A.2 GenBuf Config File

```
###########################################
# Input variable definition
###########################################
[INPUT_VARIABLES]
StoB_REQ0;
StoB_REQ1;
StoB_REQ2;
StoB_REQ3;
RtoB_ACK0;
RtoB_ACK1;
FULL;
EMPTY;

###########################################
# Output variable definition
###########################################
[OUTPUT_VARIABLES]
BtoS_ACK0;
BtoS_ACK1;
BtoS_ACK2;
BtoS_ACK3;
BtoR_REQ0;
BtoR_REQ1;

ENQ;
DEQ;
SLC0; #defined be BtoS_ACKi
SLC1; #defined be BtoS_ACKi

stateR4_0;
stateR4_1;
```

```
stateR9;

##########################################
# Environment specification
##########################################
[ENV_INITIAL]
StoB_REQ0=0;
StoB_REQ1=0;
StoB_REQ2=0;
StoB_REQ3=0;
RtoB_ACK0=0;
RtoB_ACK1=0;
FULL=0;
EMPTY=1;

[ENV_TRANSITIONS]
# Genbuf-to-sender handshake
# 3.
G(BtoS_ACK0=1 -> X(StoB_REQ0=0));
G(BtoS_ACK1=1 -> X(StoB_REQ1=0));
G(BtoS_ACK2=1 -> X(StoB_REQ2=0));
G(BtoS_ACK3=1 -> X(StoB_REQ3=0));

G((BtoS_ACK0=0 * StoB_REQ0=1) -> X(StoB_REQ0=1));
G((BtoS_ACK1=0 * StoB_REQ1=1) -> X(StoB_REQ1=1));
G((BtoS_ACK2=0 * StoB_REQ2=1) -> X(StoB_REQ2=1));
G((BtoS_ACK3=0 * StoB_REQ3=1) -> X(StoB_REQ3=1));

# Genbuf-to-receiver handshake
G((BtoR_REQ0=0 * RtoB_ACK0=1) -> X(RtoB_ACK0=0));
G((BtoR_REQ1=0 * RtoB_ACK1=1) -> X(RtoB_ACK1=0));

##########################################
# G3.B.R0 "Receiver does not acknowledge GenBuf unless requested"
G((BtoR_REQ0=0 * RtoB_ACK0=0) -> X(RtoB_ACK0=0));
G((BtoR_REQ0=1 * RtoB_ACK0=1) -> X(RtoB_ACK0=1));
# G3.B.R1 "Receiver does not acknowledge GenBuf unless requested"
G((BtoR_REQ1=0 * RtoB_ACK1=0) -> X(RtoB_ACK1=0));
G((BtoR_REQ1=1 * RtoB_ACK1=1) -> X(RtoB_ACK1=1));

##########################################
# correct behavior of the FIFO
G((ENQ=1 * DEQ=0) -> X(EMPTY=0));
G((ENQ=0 * DEQ=1) -> X(FULL=0));
G((ENQ=1 <-> DEQ=1) -> ((FULL=1 <-> X(FULL=1)) * (EMPTY=1 <-> X(EMPTY=1))) );


[ENV_FAIRNESS]
```

```
##########################################
# R2 "GenBuf requests eventually get acknowledged"
G(F((BtoR_REQ0=0 * (RtoB_ACK0=0)) + (BtoR_REQ0=1 * (RtoB_ACK0=1))));
G(F((BtoR_REQ1=0 * (RtoB_ACK1=0)) + (BtoR_REQ1=1 * (RtoB_ACK1=1))));

##########################################
# System specification
##########################################
[SYS_INITIAL]
BtoS_ACK0=0;
BtoS_ACK1=0;
BtoS_ACK2=0;
BtoS_ACK3=0;
BtoR_REQ0=0;
BtoR_REQ1=0;
ENQ=0;
DEQ=0;
SLC0=0;
SLC1=0;

stateR4_0=0;
stateR4_1=1;
stateR9=0;

[SYS_TRANSITIONS]
# Genbuf-to-sender handshake
##########################################
# R3.A.Si "GenBuf does not acknowledge senders unless requested"
G((BtoS_ACK0=0 * StoB_REQ0=0) -> X(BtoS_ACK0=0));
G((BtoS_ACK1=0 * StoB_REQ1=0) -> X(BtoS_ACK1=0));
G((BtoS_ACK2=0 * StoB_REQ2=0) -> X(BtoS_ACK2=0));
G((BtoS_ACK3=0 * StoB_REQ3=0) -> X(BtoS_ACK3=0));

# R5.B Si "GenBuf will not deassert its acknowledge unless the
# sender first deasserted its request"
G((BtoS_ACK0=1 * StoB_REQ0=1) -> X(BtoS_ACK0=1));
G((BtoS_ACK1=1 * StoB_REQ1=1) -> X(BtoS_ACK1=1));
G((BtoS_ACK2=1 * StoB_REQ2=1) -> X(BtoS_ACK2=1));
G((BtoS_ACK3=1 * StoB_REQ3=1) -> X(BtoS_ACK3=1));

# Genbuf-to-receiver handshake (3.)
##########################################
# R4.A "GenBuf will deassert its request to receiver
# a cycle after receiver aknowledged the request"
G((RtoB_ACK0=1 * BtoR_REQ0=1) -> X(BtoR_REQ0=0));
G((RtoB_ACK1=1 * BtoR_REQ1=1) -> X(BtoR_REQ1=0));

G((RtoB_ACK0=1 * BtoR_REQ0=0) -> X(BtoR_REQ0=0));
```

```
G((RtoB_ACK0=0 * BtoR_REQ0=1) -> X(BtoR_REQ0=1));
G((RtoB_ACK1=1 * BtoR_REQ1=0) -> X(BtoR_REQ1=0));
G((RtoB_ACK1=0 * BtoR_REQ1=1) -> X(BtoR_REQ1=1));

##########################################
# R4.B "GenBuf does not request both receivers
# at the same time"  = G7 (see blow)

##########################################
# R4.C.R0/R1 "GenBuf will not make two consecutive
# requests to receiver i"
G((BtoR_REQ0=1 * BtoR_REQ1=1) -> FALSE); #make the DBW complete
G((stateR4_1=0 * BtoR_REQ0=0 * BtoR_REQ1=1) -> X(stateR4_1=1 * stateR4_0=0));
G((stateR4_1=1 * BtoR_REQ0=1 * BtoR_REQ1=0) -> X(stateR4_1=0 * stateR4_0=0));
G((stateR4_1=0 * BtoR_REQ0=0 * BtoR_REQ1=0) -> X(stateR4_1=0 * stateR4_0=1));
G((stateR4_1=1 * BtoR_REQ0=0 * BtoR_REQ1=0) -> X(stateR4_1=1 * stateR4_0=1));

G((stateR4_1=0 * stateR4_0=0 * BtoR_REQ0=1 * BtoR_REQ1=0) ->
 X(stateR4_1=0 * stateR4_0=0));
G((stateR4_1=1 * stateR4_0=0 * BtoR_REQ0=0 * BtoR_REQ1=1) ->
 X(stateR4_1=1 * stateR4_0=0));

G((stateR4_1=0 * stateR4_0=1 * BtoR_REQ0=1) -> FALSE);
G((stateR4_1=1 * stateR4_0=1 * BtoR_REQ1=1) -> FALSE);


##########################################
# R6 "Only one sender can send data at any
# given time"
G(BtoS_ACK0=0 + BtoS_ACK1=0);
G(BtoS_ACK0=0 + BtoS_ACK2=0);
G(BtoS_ACK0=0 + BtoS_ACK3=0);
G(BtoS_ACK1=0 + BtoS_ACK2=0);
G(BtoS_ACK1=0 + BtoS_ACK3=0);
G(BtoS_ACK2=0 + BtoS_ACK3=0);

##########################################
# R7 "Only one receiver can receive data
# at any given time"
G(BtoR_REQ0=0 + BtoR_REQ1=0);

##########################################
# FIFO (Genbuf-to-Fifo)
# ENQ = rose(BtoS_ACKi)
G((BtoS_ACK0=0 * X(BtoS_ACK0=1)) -> X(ENQ=1));
G((BtoS_ACK1=0 * X(BtoS_ACK1=1)) -> X(ENQ=1));
G((BtoS_ACK2=0 * X(BtoS_ACK2=1)) -> X(ENQ=1));
G((BtoS_ACK3=0 * X(BtoS_ACK3=1)) -> X(ENQ=1));
```

```
G(((BtoS_ACK0=1 + X(BtoS_ACK0=0)) *
   (BtoS_ACK1=1 + X(BtoS_ACK1=0)) *
   (BtoS_ACK2=1 + X(BtoS_ACK2=0)) *
   (BtoS_ACK3=1 + X(BtoS_ACK3=0))) -> X(ENQ=0));

#specify data select signals
G((BtoS_ACK0=0 * X(BtoS_ACK0=1))  -> X(SLC0=0 * SLC1=0));
G((BtoS_ACK1=0 * X(BtoS_ACK1=1)) <-> X(SLC0=1 * SLC1=0));
G((BtoS_ACK2=0 * X(BtoS_ACK2=1)) <-> X(SLC0=0 * SLC1=1));
G((BtoS_ACK3=0 * X(BtoS_ACK3=1)) <-> X(SLC0=1 * SLC1=1));

###########################################
# DEQ = fell(RtoB_ACKi)

G((RtoB_ACK0=1 * X(RtoB_ACK0=0)) -> X(DEQ=1));
G((RtoB_ACK1=1 * X(RtoB_ACK1=0)) -> X(DEQ=1));

G(((RtoB_ACK0=0 + X(RtoB_ACK0=1)) *
   (RtoB_ACK1=0 + X(RtoB_ACK1=1))) -> X(DEQ=0));

##########################################
# R8 "GenBuf does not acknowledge sender when
# the queue is full "
G((FULL=1 * DEQ=0) -> ENQ=0);

# R8 "GenBuf doesn not request receivers when
# the queue is empty"
G(EMPTY=1 -> DEQ=0);

##########################################
# R9 "If FIFO is not empty, a send will ensue
# eventually.
# G(!EMPTY -> F(DEQ));
G((stateR9=0 * EMPTY=1) -> X(stateR9=0));
G((stateR9=0 * DEQ=1  ) -> X(stateR9=0));
G((stateR9=0 * EMPTY=0 * DEQ=0) -> X(stateR9=1));
G((stateR9=1 * DEQ=0  ) -> X(stateR9=1));
G((stateR9=1 * DEQ=1  ) -> X(stateR9=0));

##########################################
# R10 "Data read is eventually written - SKIPPED
# Follows from R8 and R9

##########################################
# R11 "GenBuf keeps FIFO order" - SKIPPED

##########################################
```

```
# R12 "Data sent to receiver is not invented
# by GenBuf" - SKIPPED


##########################################
# no immediate acknowledges
G(((StoB_REQ0=0) * X(StoB_REQ0=1)) -> X(BtoS_ACK0=0));
G(((StoB_REQ1=0) * X(StoB_REQ1=1)) -> X(BtoS_ACK1=0));
G(((StoB_REQ2=0) * X(StoB_REQ2=1)) -> X(BtoS_ACK2=0));
G(((StoB_REQ3=0) * X(StoB_REQ3=1)) -> X(BtoS_ACK3=0));


##########################################
# Force the buffer make use of the FIFO
G(((StoB_REQ0=1 + StoB_REQ1=1 + StoB_REQ2=1 + StoB_REQ3=1) * FULL=0)
-> (ENQ=1 + X(ENQ=1)));



[SYS_FAIRNESS]
##########################################
# R1 "Sender request get acknoledged" +
# R5.A "In the case the sender deasserts its request GenBuf will
# eventually deassert its acknowledge"
G(F((StoB_REQ0=0 * BtoS_ACK0=0) + (StoB_REQ0=1 * BtoS_ACK0=1)));
G(F((StoB_REQ1=0 * BtoS_ACK1=0) + (StoB_REQ1=1 * BtoS_ACK1=1)));
G(F((StoB_REQ2=0 * BtoS_ACK2=0) + (StoB_REQ2=1 * BtoS_ACK2=1)));
G(F((StoB_REQ3=0 * BtoS_ACK3=0) + (StoB_REQ3=1 * BtoS_ACK3=1)));
G(F(stateR9=0));
```

B AMBA Specification

B.1 AMBA Config File for Two Masters

```
# Encoding of the HBURST(0..1) signal
# BURSTTYPE HBURST0 HBURST1
#   INCR       0      0
#   SINGLE     1      0
#   BURST4     0      1
##########################################
# Input variable definition
##########################################
[INPUT_VARIABLES]
hready;
hbusreq0;
hbusreq1;
hlock0;
hlock1;
hburst0;
hburst1;


##########################################
# Output variable definition
##########################################
[OUTPUT_VARIABLES]
hmaster0;
hmastlock;
start;
decide;
hgrant0;
hgrant1;
hlocked;
stateA1_0;
stateA1_1;
stateG2_0;
stateG2_1;
```

```
stateG3_0;
stateG3_1;
stateG3_2;
stateG10_1;

###########################################
# Environment specification
###########################################
[ENV_INITIAL]
hready=0;
hbusreq0=0;
hbusreq1=0;
hlock0=0;
hlock1=0;
hburst0=0;
hburst1=0;

[ENV_TRANSITIONS]
#Assumption 3: no lock-request without bus-request
G(hlock0=1 -> hbusreq0=1);
G(hlock1=1 -> hbusreq1=1);

[ENV_FAIRNESS]
#Assumption 1:
G(F((stateA1_0=0)*(stateA1_1=0)) );

#Assumption 2:
G(F(hready=1));

###########################################
# System specification
###########################################
[SYS_INITIAL]
hmaster0=0;
hmastlock=0;

start=1;
decide=1;

hgrant0=1;
hgrant1=0;
hlocked=0;

#Assumption 1:
stateA1_0=0;
stateA1_1=0;

stateG2_0=0;
```

```
stateG2_1=0;

stateG3_0=0;
stateG3_1=0;
stateG3_2=0;

stateG10_1=0;

[SYS_TRANSITIONS]
#Assumption 1:
G(((stateA1_0=0)*(stateA1_1=0)*
   ((hmastlock=0)+((hburst0=1)+(hburst1=1))))->
 X((stateA1_0=0)*(stateA1_1=0)));
# Master 0:
G(((stateA1_0=0)*(stateA1_1=0)*
   ((hmastlock=1)*((hmaster0=0))*((hburst0=0)*(hburst1=0)))) ->
  X((stateA1_0=1)*(stateA1_1=0)));
G(((stateA1_0=1)*(stateA1_1=0)*(hbusreq0=1)) ->
  X((stateA1_0=1)*(stateA1_1=0)));
G(((stateA1_0=1)*(stateA1_1=0)*(hbusreq0=0)) ->
  X((stateA1_0=0)*(stateA1_1=0)));
# Master 1:
G(((stateA1_0=0)*(stateA1_1=0)*
   ((hmastlock=1)*((hmaster0=1))*((hburst0=0)*(hburst1=0)))) ->
  X((stateA1_0=0)*(stateA1_1=1)));
G(((stateA1_0=0)*(stateA1_1=1)*(hbusreq1=1)) ->
  X((stateA1_0=0)*(stateA1_1=1)));
G(((stateA1_0=0)*(stateA1_1=1)*(hbusreq1=0)) ->
  X((stateA1_0=0)*(stateA1_1=0)));

#Guarantee 1:
G(hready=0 -> X(start=0));

#Guarantee 2 (with i automata):
#Master 0
G(((stateG2_0=0)*(hmastlock=0))->X(stateG2_0=0));
G(((stateG2_0=0)*(start=0))    ->X(stateG2_0=0));
G(((stateG2_0=0)*(hburst0=1))  ->X(stateG2_0=0));
G(((stateG2_0=0)*(hburst1=1))  ->X(stateG2_0=0));
G(((stateG2_0=0)*!(hmaster0=0))->X(stateG2_0=0));
G(((stateG2_0=0)*(hmastlock=1)*(start=1)*
   (hburst0=0)*(hburst1=0)*(hmaster0=0))->X(stateG2_0=1));
G(((stateG2_0=1)*(start=0)*(hbusreq0=1))->X(stateG2_0=1));
G(((stateG2_0=1)*(start=1))->FALSE);
G(((stateG2_0=1)*(start=0)*(hbusreq0=0))  ->X(stateG2_0=0));

#Master 1
G(((stateG2_1=0)*(hmastlock=0))->X(stateG2_1=0));
```

```
G(((stateG2_1=0)*(start=0))    ->X(stateG2_1=0));
G(((stateG2_1=0)*(hburst0=1))  ->X(stateG2_1=0));
G(((stateG2_1=0)*(hburst1=1))  ->X(stateG2_1=0));
G(((stateG2_1=0)*!(hmaster0=1))->X(stateG2_1=0));
G(((stateG2_1=0)*(hmastlock=1)*(start=1)*
   (hburst0=0)*(hburst1=0)*(hmaster0=1))->X(stateG2_1=1));
G(((stateG2_1=1)*(start=0)*(hbusreq1=1))->X(stateG2_1=1));
G(((stateG2_1=1)*(start=1))->FALSE);
G(((stateG2_1=1)*(start=0)*(hbusreq1=0))  ->X(stateG2_1=0));

#Guarantee 3:
G(((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=0)*
   ((hmastlock=0)+(start=0)+((hburst0=1)+(hburst1=0)))) ->
  X((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=0)));
G(((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=0)*
  ((hmastlock=1)*(start=1)*((hburst0=0)*(hburst1=1))*(hready=0))) ->
  X((stateG3_0=1)*(stateG3_1=0)*(stateG3_2=0)));
G(((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=0)*
  ((hmastlock=1)*(start=1)*((hburst0=0)*(hburst1=1))*(hready=1))) ->
  X((stateG3_0=0)*(stateG3_1=1)*(stateG3_2=0)));

G(((stateG3_0=1)*(stateG3_1=0)*(stateG3_2=0)*((start=0)*(hready=0))) ->
  X((stateG3_0=1)*(stateG3_1=0)*(stateG3_2=0)));
G(((stateG3_0=1)*(stateG3_1=0)*(stateG3_2=0)*((start=0)*(hready=1))) ->
  X((stateG3_0=0)*(stateG3_1=1)*(stateG3_2=0)));
G(((stateG3_0=1)*(stateG3_1=0)*(stateG3_2=0)*((start=1))) -> FALSE);

G(((stateG3_0=0)*(stateG3_1=1)*(stateG3_2=0)*((start=0)*(hready=0))) ->
  X((stateG3_0=0)*(stateG3_1=1)*(stateG3_2=0)));
G(((stateG3_0=0)*(stateG3_1=1)*(stateG3_2=0)*((start=0)*(hready=1))) ->
  X((stateG3_0=1)*(stateG3_1=1)*(stateG3_2=0)));
G(((stateG3_0=0)*(stateG3_1=1)*(stateG3_2=0)*((start=1))) -> FALSE);

G(((stateG3_0=1)*(stateG3_1=1)*(stateG3_2=0)*((start=0)*(hready=0))) ->
  X((stateG3_0=1)*(stateG3_1=1)*(stateG3_2=0)));
G(((stateG3_0=1)*(stateG3_1=1)*(stateG3_2=0)*((start=0)*(hready=1))) ->
  X((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=1)));
G(((stateG3_0=1)*(stateG3_1=1)*(stateG3_2=0)*((start=1))) -> FALSE);

G(((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=1)*((start=0)*(hready=0))) ->
  X((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=1)));
G(((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=1)*((start=0)*(hready=1))) ->
  X((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=0)));
G(((stateG3_0=0)*(stateG3_1=0)*(stateG3_2=1)*((start=1))) -> FALSE);

#Guarantee 4 and 5:
#  Master 0:
G((hready=1) -> ((hgrant0=1) <-> X((hmaster0=0))));
```

```
#  Master 1:
G((hready=1) -> ((hgrant1=1) <-> X((hmaster0=1))));
# Lock
G((hready=1) -> (hlocked=0 <-> X(hmastlock=0)));

#Guarantee 6.1:
G((X(start=0)) -> ((hmaster0=0) <-> (X(hmaster0=0))));

#Guarantee 6.2:
G((X(start=0)) -> ((hmastlock=1) <-> X(hmastlock=1)));

#Guarantee 7:
G((decide=1 * X(hgrant0=1)) -> (hlock0=1 <-> X(hlocked=1)));
G((decide=1 * X(hgrant1=1)) -> (hlock1=1 <-> X(hlocked=1)));

#default master is master 0
G((decide=1 * hbusreq0=0 * hbusreq1=0) -> X(hgrant0=1));

#Guarantee 8:
G(decide=0 -> (hgrant0=0 <-> X(hgrant0=0)));
G(decide=0 -> (hgrant1=0 <-> X(hgrant1=0)));
G(decide=0 -> (hlocked=0 <-> X(hlocked=0)));

#Guarantee 10:
#  Master 1:
G(((stateG10_1=0)*(((hgrant1=1))+(hbusreq1=1)))->X(stateG10_1=0));
G(((stateG10_1=0)*(((hgrant1=0))*(hbusreq1=0)))->X(stateG10_1=1));
G(((stateG10_1=1)*(((hgrant1=0))*(hbusreq1=0)))->X(stateG10_1=1));
G(((stateG10_1=1)*(((hgrant1=1))*(hbusreq1=0)))->FALSE);
G(((stateG10_1=1)*(hbusreq1=1))->X(stateG10_1=0));

[SYS_FAIRNESS]
#Guarantee 2: all states are fair

#Guarantee 3: all states are fair

#Guarantee 9:
G(F(hmaster0=0 + hbusreq0=0));
G(F(hmaster0=1 + hbusreq1=0));
```