Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 190 (2007) 3–16
www.elsevier.com/locate/entcs

Specify, Compile, Run: Hardware from PSL

Roderick Bloem[1] Stefan Galler[1] Barbara Jobstmann[1]
Nir Piterman[2] Amir Pnueli[3] Martin Weiglhofer[1]

[1] *Graz University of Technology,* [2] *EPFL Lausanne* [3] *Weizmann Institute*
{rbloem, bjobst, mweiglh}@ist.tugraz.at
nir.piterman@epfl.ch amir@wisdom.weizmann.ac.il

Abstract

We propose to use a formal specification language as a high-level hardware description language. Formal languages allow for compact, unambiguous representations and yield designs that are 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 paper we extend these recent methods to generate compact circuits and we show their practicality by synthesizing a generalized buffer and an arbiter for ARM's AMBA AHB bus from specifications given in PSL. These are the first industrial examples that have been synthesized automatically from their specifications.

Keywords: temporal logic, synthesis, games, binary decision diagrams

1 Introduction

In the standard hardware design flow, an implementation is first written and then verified, often using a formal specification. In this paper we consider an alternative: we apply an automatic *high-level synthesis* process which generates a correct-by-construction gate-level implementation directly from a specification written in the Property Specification Language (PSL). 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 RTL code. In this paper, we demonstrate the viability of the synthesis approach for the derivation of correct code from a PSL specification.

The most obvious benefit of synthesis is that it removes the need for hand-coding the circuit. Less ambitious benefits include the possibility to construct rapid prototypes from specification and the fact that synthesis is an extremely good way to debug a specification, something that will gain importance as formal specification start to be used as the basis for a manual implementation.

Automatic synthesis of digital designs from (temporal) logical specifications has always engaged the imagination of designers and has been considered as one of

doi:10.1016/j.entcs.2007.09.004

the most ambitious and challenging problems in circuit design. First identified as Church's problem [6], several methods have been proposed for its solution [5,18]. The problem was considered again in [17] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL), a subset of PSL. The method proposed in [17] for a given LTL specification φ starts by constructing a Büchi automaton, which is then converted into a deterministic Rabin automaton. This translation may reach a doubly exponential complexity in the size of φ.

The high complexity established in [17] caused synthesis to be deemed hopelessly intractable and discouraged many practitioners from ever attempting to use it for system development. Yet, there are several interesting cases where, if the specification of the design to be synthesized is restricted to simpler automata or partial fragments of LTL, the synthesis problem can be solved more efficiently [14,21,1,8,11]. Major progress has been achieved in [16], 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. The class GR(1) covers the vast majority of properties that appear in specifications of circuits. We have implemented the approach of [16] in a tool called Anzu [1], and extended it to produce not only a BDD representing a set of possible implementations, but also an actual circuit.

We demonstrate the application of the synthesis method by means of two examples. The first is a generalized buffer from IBM, a tutorial design for which a good specification is available. The second is the arbiter for one of the AMBA buses [2], a characteristic industrial design that is not too big. Previous work on synthesis has only considered toy examples such as a simple mutual exclusion protocol, an elevator controller, or a traffic light controller [8,16,10]. This is the first time realistic industrial examples have been tackled.

This paper is a companion paper to [4]. The current paper shows the details of the GenBuf case study, whereas [4] focuses on the AMBA example. This paper gives a detailed description of the algorithm we developed to construct a circuit from a BDD, describes some extensions that were not included in [4], and shows a major improvement in the AMBA example.

The paper continues as follows: in 2, we describe how to synthesize a circuit from specifications. In Section 3, we describe the Generalized Buffer, give its formal specification, and show the results of synthesis. In Section 4, we do the same for the AMBA AHB arbiter. We discuss lessons learned in Section 5 and present our conclusions in Section 6.

2 Synthesis

In this section, we discuss how circuits can be obtained automatically from their PSL specifications. A thorough introduction to PSL can be found in [7]. The specifications shown in this paper should be easy to read for someone familiar

[1] www.ist.tugraz.at/staff/jobstmann/anzu/ contains Anzu and the specifications described here.

with LTL. In particular, `always` , `eventually!` , and `next!` correspond to G, F, and X, respectively; for an atomic proposition p, $\texttt{prev}(p)$ holds if p held in the previous cycle, $\texttt{rose}(p) = \neg\texttt{prev}(p) \wedge p$, and $\texttt{fell}(p) = \texttt{prev}(p) \wedge \neg p$. Finally, $\mathsf{next\_event!}\,(p)(\varphi) = (\neg p)\,\mathsf{U}\,(p \wedge \varphi)$.

2.1 Synthesis of GR(1) Properties

We briefly review the results presented in [16] on synthesizing GR(1) properties. We are interested in the question of *realizability* of PSL specifications (cf. [17]). 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 is a Mealy machine that, at any step, reads values of the $\mathcal{X}$ variables and outputs values for the $\mathcal{Y}$ variables.

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

In order to allow formulas of other forms (e.g., $\texttt{always}\,(p \rightarrow (q\ \texttt{until\_}\ r))$ where p, q, and r are Boolean), we augment the set of variables by adding *deterministic monitors*. Deterministic monitors are Büchi automata 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. Deterministic automata are easily represented in PSL by three sets of formulas: (1) One formula for each edge of the automaton, of the form $\texttt{always}\,(s \wedge i \rightarrow \texttt{next!}\,(s'))$, where s and s' identify states and i is an input, (2) a Boolean formula representing the initial state, and (3) a formula of the form $\texttt{always eventually!}\,(B)$ to represent the fairness condition, where B is a Boolean formula representing a set of states. (An example can be found in Section 3.3.) It should be noted that even with these restrictions, all possible (finite state) designs can be expressed as a set of properties.

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 a 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

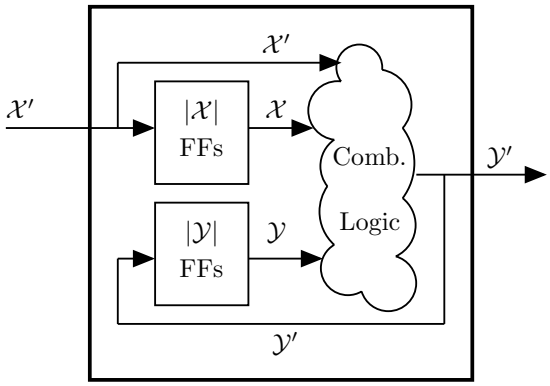

Fig. 1. Diagram of generated circuit

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. The system wins if this interaction produces an infinite play that 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. This strategy, a BDD, is a nondeterministic representation of a working implementation. Formally, we have the following.

Theorem 2.1 [16] *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.*

2.2 Generating Circuits from BDDs

In this section, we describe how to construct a circuit from the strategy. The strategy is a BDD over the variables $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{X}'$, and $\mathcal{Y}'$, where $\mathcal{X}$ are input variables, $\mathcal{Y}$ are output variables, and the primed versions represent next state variables. The corresponding circuit contains $|\mathcal{X}| + |\mathcal{Y}|$ flipflops to store the values of the inputs and outputs in the last clock tick. (See Figure 1.) In every step, the circuit reads the next input values $\mathcal{X}'$ and determines the next output values using combinational logic with inputs $\mathtt{I} = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}'$ and outputs $\mathtt{O} = \mathcal{Y}'$. Note that the strategy does not prescribe a unique combinational output for every combinational input. In most cases, multiple outputs are possible, in states that are not reachable (assuming that the system adheres to the strategy), no outputs may be allowed.

We have attempted two methods to build the combinational logic, one based on [12] and one based on computing cofactors. The approach of [12] yields a circuit that can generate, for a given input, any output allowed by the strategy. To this end, it uses a set of extra inputs to the combinational logic. Note that this is more general than what we need: a circuit that always yields one valid output given an input. We will see later that this generality comes at a heavy price in terms of the

```
for all o in O do
  S' = exists O\o . S
  p = positive cofactor of o in S'
  n = negative cofactor of o in S'
  // (*)
  careset = p*!n + !p*n
  f[o] = p minimized wrt. careset
  S = S[substitute f[o] for o]
od
```

Fig. 2. Algorithm to construct a circuit from a BDD

```
p = p * !n
n = n * !p
for all inputs i
  p' = exists i. p
  n' = exists i. n
  if p' * n' = 0 then
    p = p'; n = n';
  fi
end
```

Fig. 3. Extension to algorithm

size of the logic.

The second method to build the combinational logic uses the pseudo code shown in Figure 2. We write $\texttt{o} \in \texttt{O}$ for a combinational output and $\texttt{i} \in \texttt{I}$ for a combinational input. The strategy is denoted by `S` and `O\o` is the set of combinational outputs excluding output `o`. For every combinational output `o` we construct a function f in terms of `I` that is compatible with the given strategy BDD. The algorithm proceeds through the combinational outputs `o` one by one: First, we build `S'` to get a BDD that restricts only `o` in terms of `I`. Then we build the *positive* and *negative cofactors* (`p`,`n`) of `S'` with respect to `o`, that is, we find the sets of inputs for which `o` can be 1 (0, respectively). For the inputs that occur in the positive and in the negative cofactor, both values are allowed. The combinational inputs that are neither in the positive nor in the negative cofactor are outside of the winning region and thus represent situations that cannot occur (as long as the environment satisfies the assumptions). Thus, f has to be 1 in $\texttt{p} \wedge \neg\texttt{n}$ and 0 in $\neg\texttt{p} \wedge \texttt{n}$, which give us the set of care states. We minimize the positive cofactors with the care set to obtain the function f. Finally, we substitute variable `o` in `S` by f, and proceed with the next variable. The substitution is necessary since a combinational outputs may be related.

The resulting circuit is constructed by writing the BDDs for the functions using CUDD's DumpBlif command [19]. We then optimize the result using ABC [3] and map it to a library of standard cells. We also use ABC to estimate the number of gates needed.

In the following we describe two extensions that are simple and effective. (Cf. Section 3.3 and Section 4.)

Optimizing the Cofactors

The algorithm presented in Figure 2 generates a function in terms of the combinational inputs for every combinational output. Some outputs may not depend on all inputs and we would like to remove unnecessary inputs from the functions. Given the positive and the negative cofactor of a variable `o`, if the cofactors do not overlap when we existentially quantify variable `i`, variable `i` is not needed to distinguish between the states where `o` has to be 1 and where `o` has to be 0, and we can simply leave it out. We adapt the algorithm in Figure 2 by inserting the code shown in Figure 3 at the spot marked with `(*)`.

Removing Dependent Variables

After computing the combinational logic, we perform *dependent variables analysis* [9] on the set of reachable states to simplify the generated circuit. Given a Boolean function f over $x_0, \ldots x_n$, a variable x_i is *functionally dependent* in f if and only if $\forall x_i.f = 0$. Note that if x_i is functionally dependent, it is uniquely determined by the remaining variables of f and can be replaced by a function $g(x_0, \ldots x_{i-1}, x_{i+1} \ldots x_n)$.

Suppose our generated circuit has the set $R(\mathcal{X} \cup \mathcal{Y})$ of reachable states. If a state variable s is functionally dependent in R, we can remove the corresponding flipflop in the circuit, and instead compute its value as a function of the values of the other flipflops.

3 Generalized Buffer Case Study

3.1 Description of the Generalized Buffer

The generalized buffer (henceforth *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.

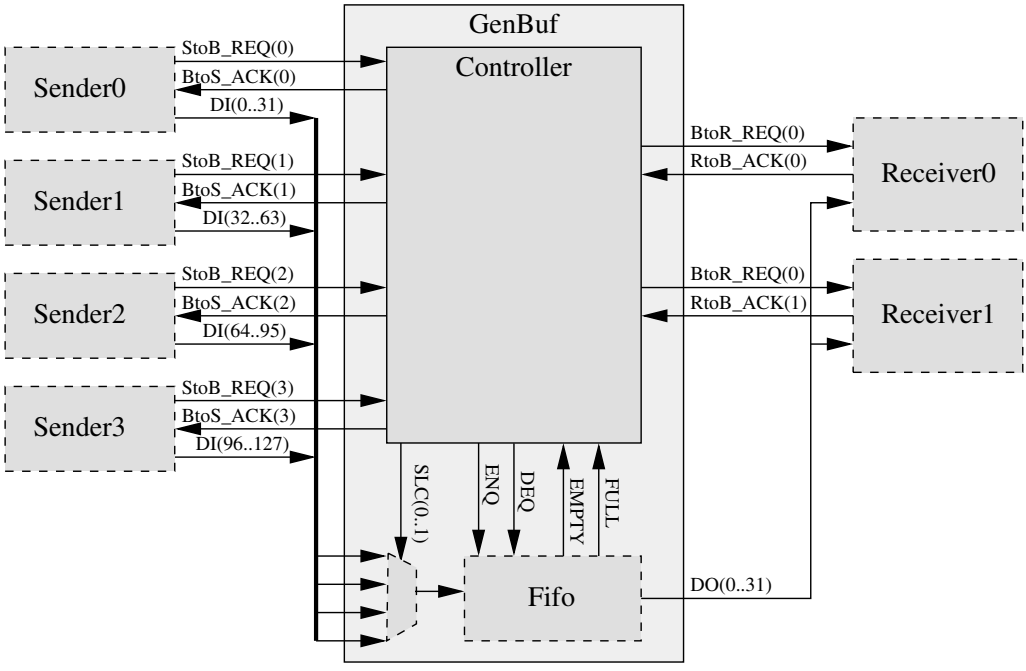

Fig. 4. Block diagram of GenBuf with four senders

Figure 4 contains a block diagram of the design and its interface. Dashed boxes represent the environment. GenBuf is a family of buffers parameterized by a number n. It transmits data from n senders to two receivers. Data is offered by the senders

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

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, and an 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 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, while assuming 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 32-bit data buses.

The control logic communicates with the FIFO through two outputs and two inputs. The outputs ENQ (enqueue data) 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 multi-bit output called SLC determines which signal from the clients is loaded when ENQ is asserted.

The interface between a sender and GenBuf is a four-phase handshake:

(i) Sender i initiates the transfer by raising StoB_REQ(i). One cycle later, it puts its data on the bus.
(ii) At least one tick after StoB_REQ(i) is raised, GenBuf raises BtoS_ACK(i) and reads the data.
(iii) One tick after BtoS_ACK(i) is raised, the sender lowers StoB_REQ(i). From this time on, it is no longer required to keep the data on the bus.
(iv) 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.

The handshake between GenBuf and the receivers is similar, except that in this case GenBuf initiates the transfer and with the exception that in Step 4 the acknowledge signal is lowered one cycle after the request is lowered.

3.2 Formal Specification

We will now present the specification that we have developed for GenBuf. It is closely related to IBM's original specification. 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.

The PSL formulas for the specification can be found in Table 1. In the table, we use $i \in \{0, \ldots, n\}$ 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. Furthermore, the acknowledgement is eventually lowered.*

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

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

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

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

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

Communication with Receivers

Assumption 2 *A request from the buffer is always acknowledged. Furthermore, the acknowledgement is lowered one tick after the request is lowered.*

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

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

Guarantee 6 *A request is not lowered until it is served. The request is lowered one cycle after the acknowledgement is raised and it cannot be raised until one cycle after the acknowledgement is lowered.*

Guarantee 7 *GenBuf does not request both receivers simultaneously. GenBuf will not make two consecutive requests to any receiver. (This guarantees round-robin scheduling.)*

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

Interface to the FIFO and the Multiplexer

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

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

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

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

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

Initially, the buffer we synthesized from the specification above ignored the FIFO. Instead it would wait until it could send data to a receiver before accepting data from a sender. Hence, we added the following property, which ensures that the FIFO is used.

Guarantee 13 *If the FIFO is not full and a sender requests to send data, the data is enqueued either in this or in the next step.*

Table 1
PSL specification

G1	$\forall i$: always (StoB_REQ(i) → eventually! BtoS_ACK(i)) $\forall i$: always (¬StoB_REQ(i) → eventually! ¬BtoS_ACK(i))
G2	$\forall i$: always (rose(StoB_REQ(i)) → ¬BtoS_ACK(i))
G3	$\forall i$: always (rose(BtoS_ACK(i)) → prev(StoB_REQ(i)))
G4	$\forall i$: always ((BtoS_ACK(i) ∧ StoB_REQ(i)) → next! BtoS_ACK(i))
A1	$\forall i$: always (StoB_REQ(i) ∧ ¬BtoS_ACK(i) → next! StoB_REQ(i)) $\forall i$: always (BtoS_ACK(i) → next! ¬StoB_REQ(i))
G5	$\forall i \forall i' \neq i$: always ¬(BtoS_ACK(i) ∧ BtoS_ACK(i'))
A2	$\forall j$: always (BtoR_REQ(j) → eventually! RtoB_ACK(j)) $\forall j$: always (¬BtoR_REQ(j) → next! ¬RtoB_ACK(j))
A3	$\forall j$: always (BtoR_REQ(j) ∧ RtoB_ACK(j) → next! RtoB_ACK(j))
A4	$\forall j$: always (RtoB_ACK(j) → prev(BtoR_REQ(j)))
G6	$\forall j$: always (BtoR_REQ(j) ∧ ¬RtoB_ACK(j) → next! BtoR_REQ(j)) $\forall j$: always (RtoB_ACK(j) → next! ¬BtoR_REQ(j))
G7	always ¬(BtoR_REQ(0) ∧ BtoR_REQ(1)). $\forall j$: always (rose(BtoR_REQ(j) → next! next_event! (rose(BtoR_REQ(0)) ∨ rose(BtoR_REQ(1))(¬BtoR_REQ(j)))).
G8	$\forall j$: always (RtoB_ACK(j) → next! (¬BtoR_REQ(j)))
G9	always (ENQ ↔ $\exists i$: rose(BtoS_ACK(i))) $\forall i$: always (rose(BtoS_ACK(i)) → SLC = i)
G10	always (DEQ ↔ (fell(RtoB_ACK(0)) ∨ fell(RtoB_ACK(1)))
G11	always ((FULL ∧ ¬DEQ) → ¬ENQ) always (EMPTY → ¬DEQ)
G12	always (¬EMPTY → eventually! DEQ)
A5	always ((DEQ ↔ ENQ) → (EMPTY ↔ next! EMPTY))) always ((DEQ ↔ ENQ) → (FULL ↔ next! FULL)) always ((ENQ ∧ ¬DEQ) → next! ¬EMPTY) always ((DEQ ∧ ¬ENQ) → next! ¬FULL)
A13	always ((¬FULL ∧ $\exists i$: StoB_REQ(i)) → (ENQ ∨ next! ENQ))

3.3 Synthesis

As explained in Section 2.1, not all PSL specifications can be synthesized directly. We first have to translate Guarantees 1, 2, 7, 12 and Assumption 2 into a suitable form.

Taking the Guarantee 4, 6, and Assumption 4 into account, we can combine

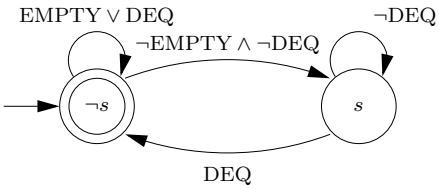

Fig. 5. Monitor for Guarantee 12

Guarantee 1 and 2 to

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

and we can 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 [13], constructing them is very simple for the formulas considered in this paper.

For instance, Figure 5 shows the deterministic automaton for Guarantee 12 stating that `always` (¬EMPTY → `eventually!` DEQ). We used the standard approach to construct Büchi automata from LTL formulas (e.g., [20]) with a slightly modified form of the standard expansion rules. In particular, we used the expansion rule `eventually!` q equals $q \vee (\neg q \wedge$ `next! eventually!` $q)$ and the fact that ¬EMPTY → φ equals EMPTY ∨ (¬EMPTY ∧ φ).

After the specification has been brought into the proper form, it is synthesized using the algorithm described in Section 2. In Figure 6 we show the time needed to synthesize GenBuf for different numbers of senders, excluding the time taken by ABC to optimize the circuit, which is typically a few seconds. We have plotted the time taken to generate the circuits using the method based on [12], the time needed by our algorithm, our algorithm with the optimization of the cofactors, and our algorithm with optimization of the cofactors and removal of dependent variables. (See Section 2.2.) The time for synthesis remains under one minute and is similar for all methods. (We can not explain why synthesis is much faster when we have nine senders.) We are able to synthesize specifications of GenBuf up to 60 senders. Our implementation needs approximately 13 hours for 60 senders. Synthesis of a specification containing 70 senders seems to be a matter of time and not memory.

In Fig. 7 we show the number of gates of the resulting circuits after optimization by ABC. The method based on [12] yields circuits that are about an order of magnitude larger than ours. (For more than 6 senders, this method yields circuits that are too large for ABC to handle.) Optimizing the cofactors yields about 16%. Removing the dependent variables reduces the number of latches by 5–12%. Which dependent variables are found is hard to predict, but usually includes ENQ and some or all of the SLC signals. (These signals can be inferred from the BtoS_ACK(s)ignals.)

Optimization by ABC yields an improvement in number of gates of about 20%.

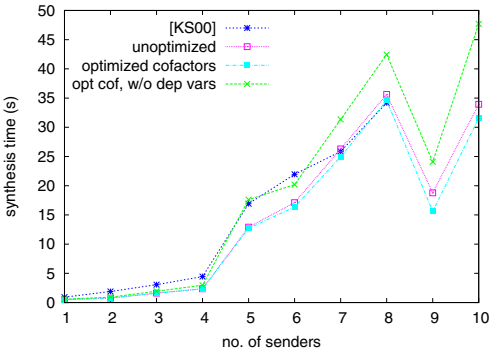

Fig. 6. Time to synthesize GenBuf

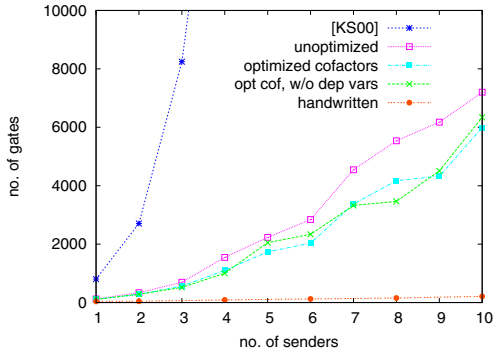

Fig. 7. Size of the GenBuf circuits

It should be noted that the growth of the circuit is well-behaved, but a circuit of 5 000 gates is still very large.

4 AMBA AHB Case Study

In this section we summarize a case study that we performed on the *Advanced High-Performance Bus* (AHB). We present new results that are significantly better than the ones in [4].

The AHB is an on-chip communication standard that connects such devices as processor cores, cache memory, and DMA controllers. The bus allows up to 16 *masters* to communicate (read or write) with up to 16 *clients*. The bus consists of a data bus and an address bus. At any time, only one master is allowed to access each of the buses. Access to the address bus is controlled by the *arbiter*, which is the subject of this section.

An *access* to the bus can be *locked* or *unlocked*, and either a single *transfer* or a *burst*, which consists of a specified or unspecified number of transfers. A locked access may not be interrupted, so the arbiter has to take the different access modes into account.

To access the bus, a master drives the address and control signals to indicate the type of transfer it wants. Slaves are passive and can only respond to a request. The arbiter decides the next owner of the bus and whether its access will be locked. Then, it asserts the corresponding control signals to indicates its decision, and when the current transfer is finished, the bus is handed over.

We derived a formal specification from the AMBA AHB standard for the arbiter. The standard allows for a variety of arbitration schemes including priority-based and fair buses. We wrote a specification for a fair bus, synthesized it, and constructed a circuit. Subsequently, we constructed a circuit as described in Section 2.2.

In our initial experiments [4], we were only able to synthesize arbiters for up to four masters, for larger arbiters the synthesis algorithm ran out of memory when building the strategy. (2GB of memory were available.) After rewriting the specification, without changing its meaning, we can handle up to ten masters. The time

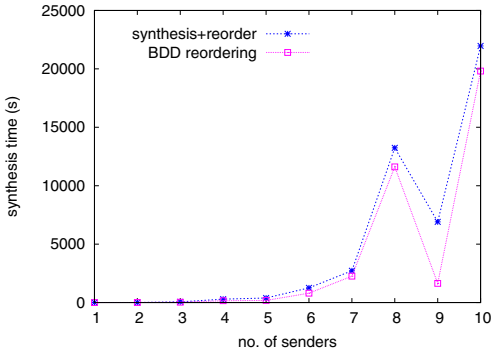

Fig. 8. Time to synthesize AMBA bus

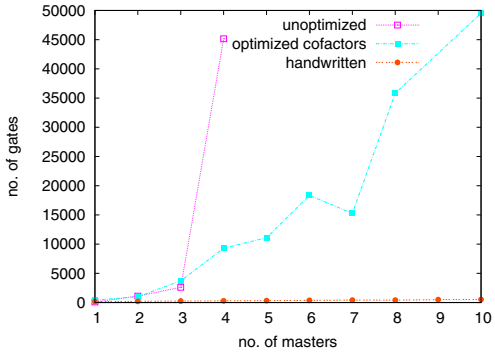

Fig. 9. Size of the AMBA circuits

for synthesis is shown in Figure 8 and ranges from a few second to 6.5 hours. Most of the time is spent in reordering BDDs. (We do not know why synthesis for nine masters is faster then for eight.)

In Figure 9, we show the number of gates of the arbiter as a function of the number of masters using our algorithms and a manual implementation. For one master the manual and the automatically generated implementation have approximately the same size. The automatically generated implementations grow rapidly with the number of masters, while the manual implementations are nearly independent of the number of masters. The automatically generated implementation for ten master is about a hundred times larger than the manual implementation.

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 Discussion

In this section we discuss the most important benefits and drawbacks of automatic synthesis, as we perceive them.

Writing the formal specification for the generalized buffer was straightforward. This may be ascribed in part to the simplicity of the block and in part to the clear specification provided by IBM (although the specification was neither complete, nor free of mistakes).

On the other hand, writing a complete formal specification for the AMBA arbiter was not trivial. First, many aspects of the arbiter are not defined in ARM's standard. Such ambiguities would lead to long discussions on how someone implementing a bus device could read the standard, and which behavior the arbiter should allow. Note that the same problem occurs when writing a manual implementation for the arbiter.

Construction of a complete specification is an iterative process. For the arbiter in particular, this process was cumbersome, and we encountered problems

formulating certain requirements. These problems were best solved by introducing additional signals (much like one does when writing a manual implementation). In the process, we wrote several unrealizable specifications, and some specifications that yielded circuits that did not adhere to our expectations. (A simple example of unexpected behavior for GenBuf is described and resolved in Section 3.2, Guarantee 13.) The tool complains about unrealizable specifications, but does not offer any help in pinpointing the problem. Likewise, unexpected behavior is typically very easy to find, but not always easy to remedy. Some work on tools for debugging specifications has taken place [15], but further research, in particular in connection with realizability, is needed.

The effort for a manual implementation of a parameterized circuit usually does not depend strongly on the parameter. (The parameter is the number of senders in case of GenBuf and the number of masters in case of the arbiter). The same is not true for automatic synthesis: the time for synthesis and the size of the resulting circuit grow with the parameter. Unfortunately, the generated gate-level output is complicated and cannot easily be changed by hand.

Finding a small implementation for a given specification is hard. A specification corresponds to a (possibly infinite) set of open controllers that implement it. Synthesis proceeds in two steps. First, the algorithm of [16] prescribes a set of flipflops and constructs a strategy that corresponds to a finite (but typically large) set of combinational blocks that implement a correct open controller. Second, we must pick one controller with a small representation from this set. Not every small implementation that is allowed by the specification survives step one. Even if it does, it is hard to find a small circuit from among the ones allowed by the strategy in step two. We are researching methods to improve each step and we expect that we will be able to significantly reduce the size of the resulting circuits.

On the upside, the resulting PSL specification is short, readable, and easy to modify, much more so than a manual implementation in VERILOG. The synthesis algorithm was also a excellent tool to get the specifications consistent and complete. Although the construction of the specifications was sometimes bothersome, we doubt we would have managed to write a complete and consistent specification without the synthesis tool.

Automatic synthesis is first and foremost applicable to control circuitry. We are looking into methods to combine manually coded data paths with automatically synthesized control circuitry, which takes the form of a controller synthesis problem.

6 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 a manual implementation is to be based on the formal specification.

Although automatic synthesis has long been pursued, only recent developments

have made it applicable to realistic examples. This paper, together with its companion [4], presents the first time that real-life blocks have been synthesized from their specifications. The circuits that we obtain are quite large, but the approach is still young and only a few avenues for optimization have been pursued. We attempted to generate circuits using an approach of [12]. A second attempt using cofactors yielded circuits that are an order of magnitude smaller, and optimizations to that approach yielded a significant improvement. We expect that future research will yield further large improvements, making automatic synthesis a real alternative to manual coding of some types of circuits.

Acknowledgement

This work was supported in part by the European Commission under contract number 507219 (PROSYD). We are grateful to Karin Greimel and Milan Milinkovic for their help with the implementation.

References

[1] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. *ACM Transactions on Computational Logic*, 5(1):1–25, Jan. 2004.

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

[3] Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and verification, release 61208. http://www.eecs.berkeley.edu/ alanmi/abc/.

[4] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2007.

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

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

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

[8] 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*, pages 477–492, 2005.

[9] A. J. Hu and D. Dill. Reducing BDD size by exploiting functional dependencies. In *Proceedings of the Design Automation Conference*, pages 266–271, 1993.

[10] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *Conference on Formal Methods in Computer Aided Design*, pages 117–124, 2006.

[11] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238, 2005.

[12] J. H. Kukula and T. R. Shiple. Building circuits from relations. In *Conference on Computer Aided Verification*, pages 113–123, 2000.

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

[14] M. Maidl. The common fragment of CTL and LTL. In *Proc. Foundations of Computer Science*, pages 643–652, 2000.

[15] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In *Design Automation Conference*, 2006.

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

[17] 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.

[18] M. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics*. American Mathematical Society, 1972.

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

[20] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Conference on Computer Aided Verification (CAV'00)*, pages 248–263, 2000.

[21] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Proceedings of the International Conference on the Implementation and Application of Automata*. Springer-Verlag, 2003.