Automatic Hardware Synthesis from Specifications: A Case Study

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

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 an arbiter for ARM*'s* AMBA AHB *bus and a generalized buffer from specifications given in* PSL*. These are the first industrial examples that have been synthesized automatically from their specifications.*

1 Introduction

In the standard design flow for a block of hardware, 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), thus removing the need for hand-coding the circuit. 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.

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 [3], several methods have been proposed for its solution [2, 12]. The problem was considered again in [11] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL), a subset of PSL. The method proposed in [11] 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 [11] caused synthesis to be deemed hopelessly intractable and discouraged many practitioners from ever attempting to use it for 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 [10], 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 [10] 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 one of the AMBA buses [1], a characteristic industrial case which is not too big. The second is a generalized buffer from IBM, for which a good specification was available. Previous work on synthesis has only considered toy examples such as a simple mutual exclusion protocol, an elevator controller, or a traffic light controller [5, 10, 6]. This is the first time a realistic industrial example has been tackled.

The paper continues as follows: in Section 2, we (briefly) introduce the synthesis method developed in [10]. In Section 3, we describe the AMBA bus protocol, give a formal specification of the arbiter, and discuss the results of synthesis. In Section 4, we do the same for the generalized buffer. In Section 5 we describe how the circuit is generated. We discuss lessons learned in Section 6 and present our conclusions in Section 7.

[1] Anzu and the specifications described here can be found at http://www.ist.tugraz.at/staff/jobstmann/anzu/

978-3-9810801-2-4/DATE07 © 2007 EDAA

2 Preliminaries

Property Specification Language. We will not give an introduction to PSL. A thorough introduction to PSL can be found in [4]. The specifications shown in this paper should be easy to read for someone familiar with LTL. In particular, `always` , `eventually!` , and `next!` correspond to G, F, and X, respectively. The `until_` operator requires the first operand to hold either forever or up to and including the time that the second operand holds. The construct ϕ `before` ψ is equivalent to $\neg\psi$ `until_` ϕ. We use one operator that is not in PSL: ϕ `until_`$[i]$ ψ means that ϕ holds either forever or up to and including the ith time that ψ holds.

Synthesis of GR(1) Properties We briefly review the results presented in [10] on synthesizing GR(1) properties. We are interested in the question of *realizability* of PSL specifications (cf. [11]). 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.

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 $\bigwedge_i$ `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) 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. (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 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. (Efficiently extracting a circuit from this BDD is one of the subjects of the paper.) Formally, we have the following.

Theorem 1. [10] 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 AMBA AHB Case Study

3.1 Protocol

ARM's *Advanced Microcontroller Bus Architecture* (AMBA) [1] defines the *Advanced High-Performance Bus* (AHB), an on-chip communication standard connecting such devices as processor cores, cache memory, and DMA controllers. Up to 16 *masters* and up to 16 *slaves* can be 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 this section. 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; neither are aspects such as the arbitration protocol.

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 different master. Driven by the arbiter

The following set of signals is multiplexed using HMASTER as the control signal. For instance, 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 straightforward 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 "split" a burst access and request that it be continued later. We have left this feature out for simplicity, but it can be handled by our approach.

A typical set of accesses is shown in Fig. 1. (Please ignore the DECIDE, START, and LOCKED 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 that cannot be interrupted. At time 6, when the last transfer in the burst starts, the arbiter prepares to hand over the bus to Master 2 by changing the

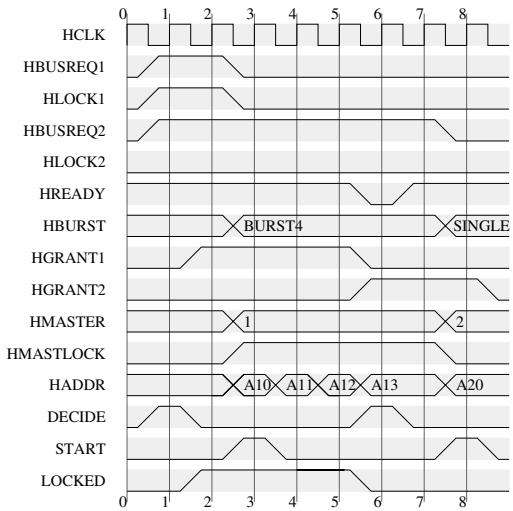

Figure 1. An example of AMBA bus behavior

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.

3.2 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. In Fig. 1, for instance, START is high in Step 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.

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 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. The formal PSL specification is given in Table 1.

Starting an Access

Assumption 1. During a locked unspecified length burst, leaving HBUSREQi high locks the bus. This is forbidden by the standard.

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

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

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

Table 1. PSL **specification**

A1	$\forall i$: always ((HMASTLOCK $\wedge$ HBURST = INCR $\wedge$ HMASTER = i) $\rightarrow$ next! eventually! $\neg$HBUSREQi)
A2	always eventually! HREADY
A3	always (HLOCKi $\rightarrow$ HBUSREQi)
G1	always ($\neg$HREADY $\rightarrow$ next! $\neg$START)
G2	$\forall i$: always ((HMASTLOCK $\wedge$ HBURST = INCR $\wedge$ START $\wedge$ HMASTER = i) $\rightarrow$ next! ($\neg$START until_ $\neg$HBUSREQi))
G3	$\forall i$: always (HMASTLOCK $\wedge$ HBURST = BURST4 $\wedge$ START $\rightarrow$ (HREADY $\wedge$ next! ($\neg$START until_ [3] HREADY)) $\vee$ ($\neg$HREADY $\wedge$ (next! $\neg$START until_ [4] HREADY)))
G4	$\forall i$: always (HREADY $\rightarrow$ (HGRANTi $\leftrightarrow$ next! HMASTER = i))
G5	always (HREADY $\rightarrow$ (LOCKED $\leftrightarrow$ next! HMASTLOCK))
G6	$\forall i$: always (next! $\neg$START $\rightarrow$ ((HMASTER = $i \leftrightarrow$ next! HMASTER = i) $\wedge$ (HMASTLOCK $\leftrightarrow$ next! HMASTLOCK)))
G7	$\forall i$: always ((DECIDE $\wedge$ next! HGRANTi) $\rightarrow$ (HLOCKi $\leftrightarrow$ LOCKED))
G8	always ($\neg$DECIDE $\rightarrow \bigwedge_i$(HGRANTi $\leftrightarrow$ next! HGRANTi)) always ($\neg$DECIDE $\rightarrow$ (LOCKED $\leftrightarrow$ next! LOCKED))
G9	$\forall i$: always (HBUSREQi $\rightarrow$ eventually! ($\neg$HBUSREQi $\vee$ HMASTER = i))
G10	$\forall i \neq 0$: always ($\neg$HGRANTi $\rightarrow$ (HBUSREQi before HGRANTi)) always (DECIDE $\wedge \forall i$: $\neg$HBUSREQi $\rightarrow$ next! HGRANT0)
G11	DECIDE $\wedge$ START $\wedge$ HGRANT0 $\wedge$ HMASTER = 0 $\wedge$ $\neg$HMASTLOCK $\wedge \forall i \neq 0$: $\neg$HGRANTi
A4	$\forall i$($\neg$HBUSREQi $\wedge$ $\neg$HLOCKi) $\wedge$ $\neg$HREADY

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.

Guarantee 3. When a length-four 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. (In the formula, we treat the cases where HREADY is true initially separately from the case in which it is not.)

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 and that the arbiter cannot change HMASTER without giving a grant.

Guarantee 5. Whenever HREADY is high, the signal HMASTLOCK copies the signal LOCKED.

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

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. 1.) 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. 1, the signals are read in Step 1 and the corresponding access starts at Step 3.)

Guarantee 7. When the arbiter decides to grant the bus, it uses LOCKED to remember whether a locked access was requested.

Guarantee 8. We do not change the grant or locked signals if DECIDE is low.

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. (Without this property, there is no need for the arbiter to serve any master at all.)

Guarantee 10. We do not grant the bus without a request, except to Master 0. If there are no requests, the bus is granted to Master 0.

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.

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

3.3 Synthesis

As explained in Section 2, not all PSL specifications can be synthesized directly. Rather, we first have to build deterministic monitors for the formulas A1, G2, G3, and G10. Although there are formulas for which no deterministic monitor exists, and constructing such monitors is hard in general [8], constructing them is very simple for the formulas considered in this paper.

For instance, let PRE be HMASTLOCK $\wedge$ HBURST = INCR $\wedge$ HMASTER = i. Guarantee 2 (for one master) reads

$$\texttt{always}\,((\text{START} \wedge \text{PRE}) \rightarrow \texttt{next!}\,(\neg\text{START}\ \texttt{until\_}\ \neg\text{HBUSREQi})).$$

Figure 2 shows the automaton for this formula, constructed using the standard approach to construct Büchi automata from LTL formulas (e.g., [15]) using a slightly modified form of the standard expansion rules. In particular, we used the expansion rule ϕ until_ q equals $(q \wedge \phi) \vee (\neg q \wedge \phi \wedge$ next! (ϕ until_ q)) and the fact that (START $\wedge$ PRE) $\rightarrow \phi$ equals ($\neg$START $\vee \neg$PRE) $\vee$ (START $\wedge$ PRE $\wedge \phi$).

Note that deterministic automata are easily represented in PSL by a set of formulas of the form always ($s \wedge i \rightarrow$ next! (s')), one for each edge, where s and s' identify states and i is an input.

After the specification has been brought into the proper form, it is synthesized using the algorithm of [10]. Sub-

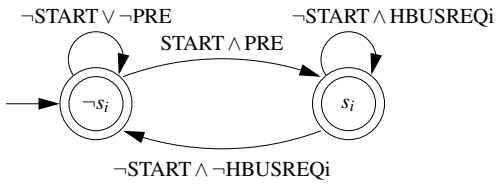

Figure 2. Monitor for Guarantee 2

sequently, a circuit is constructed using the techniques described in Section 5, optimized using SIS' `script.rugged` and then mapped by SIS using `stdcell2_2` [13].

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 200s, size 79k, delay 63. For four masters we need 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.

4 Generalized Buffer Case Study

In this section we briefly describe a case study that we performed on a generalized buffer[2]. The buffer communicates with four senders and two receivers using two different four-stage handshake protocols. It receives data from the senders in an arbitrary order and serves the receivers in round-robin order. The buffer contains a FIFO to hold data until it can be sent to the receivers.

Since the buffer is a tutorial design used by IBM for teaching PSL, it comes with a good informal specification and a relatively complete formal specification. It was quite easy to complete the specification. Most importantly, we only specified the control logic, assuming that an implementation of the FIFO was given. In order for the design to be realizable, we needed an abstract specification of the FIFO. It was sufficient to assume that putting data into the queue results in a nonempty queue and removing data results in a queue that is not full. The initial circuit would ignore the FIFO: it waited until it could send data to a receiver before accepting data from a sender. This was easily remedied by requiring that the buffer not remain idle when a request from a sender occurs. The full specification consists of 12 guarantees and 4 assumptions.

The buffer synthesizes in 2 seconds and the resulting circuit has size 63k and a delay of 60.

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

5 Generating Circuits from BDDs

The synthesis approach presented in [10] 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. Initial experiments using the approach of [7] to generate combinational logic from the strategy yielded extremely large circuits. Our current approach generates combinational logic using the following pseudo code, where `S` is the Strategy and `O\o` denotes set of combinational outputs excluding output `o`. The algorithm proceeds one combinational output at a time, generating a function that is consistent with the relation for that output.

```
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 [14]. 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 over [7]. (Note that [7] solves a more general problem: that of extracting a circuit that can exhibit any behavior consistent with the relation. Our circuit only implements one such behavior.) The generation of the circuit from the BDD currently takes the major part of the time to synthesize a circuit.

6 Discussion

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

Writing a complete formal specification for the arbiter was not trivial. Many aspects of the arbiter are not defined in ARMs 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 VERILOG implementation.

Second, it is not trivial to translate the informal specification to formulas. One of the important insights when writing the specification of the arbiter was that the additional

signals, START, DECIDE, and LOCKED were needed. This problem also occurs when we attempt to formally verify a manually coded arbiter, in which case the same signals are useful. (In fact, these signals occur, in one form or other, in our manual implementation as well.)

We did not have the same problems with the generalized buffer. The informal specification was clear and the formal specification could be reused to a large extent.

The effort for a manual implementation of an arbiter for, say, four masters is not much different from that for 16 masters. The same is not true for automatic synthesis: the time to synthesize the arbiter grows 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 with the important characteristic that the space of allowed functions is very large.

On the upside, the resulting PSL specification is short, readable, and easy to modify, much more so than a manual implementation in VERILOG. We expect that it is easier to learn the way the arbiter functions from the formal specification than from a manual VERILOG implementation. The synthesis algorithm was also a great tool to get the specifications to be consistent and complete.

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 [9], 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 a future 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 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 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. We expect that future research will yield further large improvements, making automatic synthesis a real alternative to manual coding of some types of circuits.

Acknowledgments 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 and to Margaret Rugira, Chris Styles and Colin Campbell at ARM for their help with understanding the specification. (This does not constitute an endorsement of the paper by ARM.)

References

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

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

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

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

[5] 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.

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

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

[8] 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*, 1998.

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

[10] 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.

[11] 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.

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

[13] 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 *International Conference on Computer Desgin*, pages 328–333, 1992.

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

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