A Constructive Approach towards Correctness of Synthesis — Application within Retiming *

Dirk Eisenbiegler, Ramayya Kumar and Christian Blumenröhr

Institute for Circuit Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid), University of Karlsruhe, Germany
e–mail: eisen@ira.uka.de, kumar@fzi.de, blumen@ira.uka.de

Abstract— **This paper is dedicated to correct synthesis. By correct synthesis we mean, that there is a mathematical proof telling us, that the output circuit description fulfills the input circuit description. There are several ways to achieve correct synthesis. In this paper, we present a novel approach which integrates conventional synthesis algorithms thus guaranteeing the same quality of designs. Our approach is fully automatic, although it is based on rule applications within a theorem prover. We compare our results in the area of retiming to other approaches.**

I. Introduction

Performing synthesis steps by hand is critical as far as correctness is concerned. Nowadays most synthesis steps are fully automated and the synthesis results have become much more reliable than hand designs. However, the correctness of synthesis now depends on the correctness of the *synthesis programs*. One could think of verifying synthesis programs and thereby guarantee the correctness of all synthesis results. But in general, synthesis programs are far too complex to apply formal software verification techniques.

There are several reasons why automated synthesis may be error prone:

- Synthesis tools have become more and more complex with an increasing number of people being involved in the design of the synthesis tool.
- Synthesis tools employ complex data types and procedures for representing and transforming circuits.
- Synthesis tools are frequently combined. As intermediate formats, HDLs are used. Very often the semantics of HDLs are not defined as precisely as they should be, and hence the circuit descriptions are interpreted differently by different tools.

There are several methods for increasing the reliability of synthesis results [1], [2], [3], [4]. See [5], [6] for a survey on related work. In this paper, we present our formal synthesis approach HASH (Higher order logic Applied to Synthesis of Hardware), and compare our approach to other approaches.

The paper is structured as follows: We first give an overview of techniques for verifying synthesis results. Then we will describe our formal synthesis approach (HASH) as a general method for achieving correct synthesis. Afterwards we apply this concept to retiming. Finally some experimental results achieved with HASH are compared with other approaches.

*This work has been partly financed by the Deutsche Forschungsgemeinschaft, Project SCHM 623/6-1.

II. Post-Synthesis-Verification Approaches

Since it is practically impossible to verify the correctness of conventional synthesis programs, designers normally validate synthesis results by simulating both input and output circuit descriptions, thus increasing reliability. If one is lucky, one might find errors quickly. However, the absence of errors can only be guaranteed by exhaustive simulation, which is applicable only to very small sized circuits.

Tautology Checkers, Model Checkers

Formal verification techniques are an advanced approach towards guaranteeing correctness. There are fully automated verification techniques, as well as verification techniques that require user interaction. Within the circuit designer community, verification techniques will only be accepted if they are fully automatic! However, full automation can only be achieved at lower levels of abstraction. There are two automatic verification techniques, that are frequently used: tautology checkers and model checkers.

Boolean tautology checkers [4] can only be applied to pure combinatorial circuits and to sequential circuits with same state representation. The timing complexity increases exponentially with the size of the circuits. In order to also verify general synchronous circuits, model checkers [3] are applied. Model checkers perform a breadth first state traversal on the product circuit. The set of states that have been reached so far are represented by BDDs. Step by step, the set of states is increased by states that can directly be reached, starting from one of the states, in the current set. Each traversal step is performed by a BDD-transformation. The algorithm terminates, if no further states are found, i.e. the BDD remains unchanged. There are two aspects that have a major impact on the duration of model checking: the size of the BDDs and the number of traversal steps. Both the number of traversal steps and the size of the BDD grow exponentially with the number of state variables.

Specialized Verification Techniques

A major handicap for general verification techniques is, that they just get the input and the result of the synthesis process, but they cannot exploit the knowledge of *how* the result was derived. Verification can be performed much more efficiently if one knows, that only specific steps have been performed.

The approach presented in [7] is based on a model checker and increases performance by exploiting functional dependencies. For specific synthesis domains (retiming, state minimization,...), this technique can reduce the verification time significantly, as compared to conventional model checking.

Another specialized verification technique that is designed for retiming synthesis steps only is described in [8]. During retiming the overall shape of the structure is not changed entirely. It is only the registers, that have been shifted. The program tries to "match" the former and the retimed circuit description. This can be performed pretty fast. In contrast to [7], this approach is limited to pure retiming.

There are two major drawbacks of these spezialized verification techniques: complexity and combinability. As regards the complexity, the general problem of proving the equivalence of two circuits is NP-complete. For some synthesis steps, there do exist some powerful verification techniques [8]. In [8], it has been exploited, that the implementation was derived by retiming the original circuit. Nevertheless, the information about *how* the retiming was performed has to be extracted by "matching" the two descriptions. The overall scenario "synthesis+verification" could be significantly speeded up, if one fed the information, on *how* the retiming was performed, directly from the synthesis step to the verification step. However, exploiting the information about *how* synthesis was performed eases verification but is impossible for complex synthesis procedures consisting of various single steps. This motivates us to tightly bind synthesis and verification for obtaining an integrated formal synthesis step.

With respect to the combinability, a specialized verification technique can only be applied to its corresponding synthesis step. For example, there are specialized verification techniques for logic minimization (tautology checkers) as well as retiming [8], [7], but there is no efficient technique for a compound "retiming+logic minimization" step, and one would have to resort to a general verification technique. It shows, that splitting synthesis into very basic synthesis steps and combining them with specially adapted verification techniques increases verification performance — divide and conquer.

III. The Formal Synthesis Approach — HASH

A. Concept

HASH (Higher order Logic Applied to Synthesis of Hardware) is a toolbox for implementing formal synthesis programs. HASH provides a set of basic hardware transformations implemented as logical derivation steps within the theorem prover HOL [9]. HASH provides means for embedding existing synthesis heuristics: logical transformations that are parametrized by control information describing how the synthesis step is to be performed. This leads us to formal synthesis programs where the transformational aspect (inside HOL) is clearly separated from the design space explorational aspects (conventional synthesis heuristics, outside HOL).

With respect to the complexity problem mentioned in the previous section, HASH circumvents it by providing *forward derivational steps*, instead of post-synthesis verification. Searching for the proof of equivalence for two circuits is NP-complete — formally transforming one to the other is not!

In HASH — as well as in specialized verification techniques — synthesis is split into a series of basic transformation steps whose correctness aspects can be handled efficiently. HASH also furnishes the means for combining these basic transformation steps towards complex synthesis programs. If for example, one formal synthesis step leads to the theorem $\vdash a = b$ and the succeeding synthesis step leads to $\vdash b = c$, the compound synthesis step $\vdash a = c$ can efficiently be derived by means of a simple transitivity rule in HOL. The first step could be e.g. a retiming step and the second a logic minimization step. Since the complexity of the transitivity step in HOL is constant (pointers — no copying), the overall complexity of the compound synthesis step is the sum of its two parts.

B. Security Aspects

Theorem provers such as HOL provide a set of functions for constructing, destructing and manipulating terms, formulae and theorems. Terms, formulae and theorems have specific data types. These types are encapsulated. There is a *fixed set* of basic functions for producing values having these types. There is no other way to produce terms, formulae and theorems.

Theorem provers guarantee safety. The only way to derive a theorem is by deriving it from axioms and rules, i.e. applying basic functions. So theorem provers are as safe as the implementation of their core of basic functions. Usually these cores are pretty small. The HOL calculus [9], for example, consists of 8 rules and 5 axioms. This makes theorem provers very reliable.

Tautology checkers and model checkers on the other hand do not have such a core. They are nothing but programs that somehow decide whether or not some formula holds. In general, the implementation of such programs result in large source codes, and each programming error may lead to false verification results.

C. Conventional vs. Formal Synthesis

The formal synthesis approach HASH derives the output circuit description within a theorem prover, rather than just computing it as in conventional synthesis. The major difference is the result: Conventional synthesis programs only map the input circuit description to the output circuit description. Formal synthesis programs map the input circuit description to a theorem stating that some implementation, which has been derived during formal synthesis, fulfills the input circuit description. Formal synthesis however presumes, that all circuit descriptions are represented within logic.

The advantage of a formal synthesis program is its implicit correctness. Whenever it produces a result, this result is also correct. Formal synthesis programs are as reliable as the core of the theorem prover that they are based on. This makes them much more reliable than conventional synthesis programs, where there is no such core, and one would have to verify the entire program in order to ensure correctness. We will use the simple retiming synthesis step in order to describe the benefits of formal synthesis as compared to other approaches towards synthesis correctness.

IV. Retiming by Means of Logical Transformations

The implementation of our retiming procedure is based on the theory "Automata" [10], which we implemented in the HOL theorem prover. Automata was designed for synthesis purposes. Automata provides means for representing synchronous circuits and is also the base for synthesis specific transformations such as state minimization, state encoding, logic optimization and retiming.

A. The Procedure

The retiming procedure in HOL is based on a universal retiming theorem. This theorem represents a general pattern, which can be instantiated for various retiming transformations. It can be applied in both directions: forward/backward retiming. Figure 1 informally sketches the meaning of this theorem. Hereby, s denotes the initial state, x the auxiliary variables within the combinatorial part and s' the successor state.

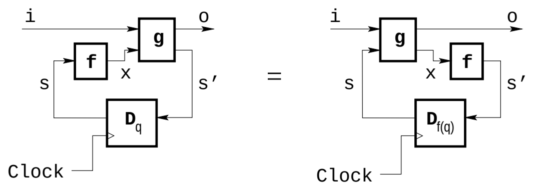

Fig. 1. General Pattern for Rewriting

For forward retiming, the combinatorial part is split into two: one part f over which the registers are shifted and the other part g which is not affected. There is one compound register named D with q as an initial value. In the retimed circuit, the initial state of the new compound register becomes $f(q)$. The theorem [RETIMING_THM] states, that the original and the retimed circuit are equivalent.

[RETIMING_THM]

```
1  ⊢ automaton(
2       (λ(i, s). let x = f(s) in let (o, s') = g(i, x) in (o, s')) ,
3       q
4     )
5     =
6     automaton(
7       (λ(i, s). let (o, x) = g(i, s) in let s' = f(x) in (o, s')) ,
8       f(q)
9     )
```

Backward retiming is more complex since one has to find the q's corresponding to some expression representing $f(q)$. We will not discuss this issue in this paper.

In the Automata theory, circuits are unambiguously represented by pairs consisting of a compound function and an initial state. This compound function describes the output and the next-state behavior. The registers are formalized implicitly. The constant automaton maps such pairs to functions that map time dependent input signals to time dependent output signals.

The output and state transition functions in lines 2 and 7 of the theorem correspond to the structures of the combinatorial parts of the circuits (figure 1). These functions map the pair consisting of input i and the current state s onto the pair consisting of output o and next state s'. Lines 3 and 8 correspond to the initial states of the two circuits.

Using an automaton as a formal representation, the overall retiming procedure consists of four steps:

1. First the combinatorial part is split into f and g. Assigning combinatorial components to f or g can either be performed by hand or some arbitrary external program.
2. Then the general retiming theorem is applied: The current circuit description is matched with the left hand side of the equation and one proceeds with the right hand side.
3. Then f and g are joined to a single combinatorial part.
4. Finally the new initial values of the shifted registers $f(q)$ are determined via evaluation.

Figure 2 shows a retiming example and figure 3 describes, how it is matched to our retiming theorem. In our example, there are three combinatorial parts: $\geq$, $+1$ and MUX. When applying our synthesis procedure, f consists of the $\geq$-component only and g consists of $+1$ and MUX.

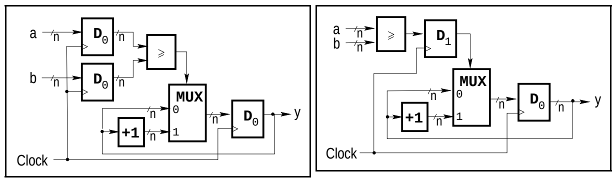

Fig. 2. Retiming Example

B. Where are Logical Skills Needed?

To answer this question, one has to distinguish between the designer of the formal synthesis tool and the circuit designer, who uses this tool. Proving the correctness of theorems such as [RETIMING_THM] and implementing corresponding transformations (four steps of the retiming procedure) requires a thorough understanding of logic, hardware and underlying theorem prover (HOL). The formula in [RETIMING_THM] is true higher order logic (universal quantification over functions f and g, polymorphism). Its proof is tedious

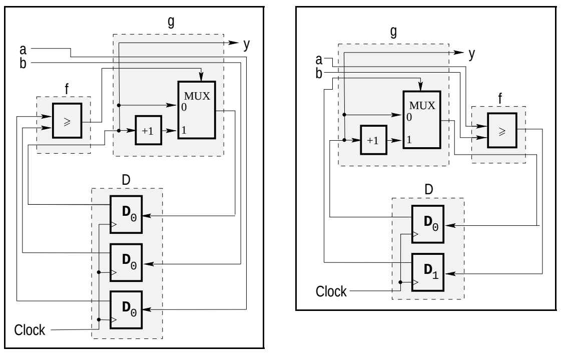

Fig. 3. Example for Applying the Retiming Scheme

and cannot be automated (induction over time, etc.). However it has only to be proved once and for all!

The above mentioned procedure for retiming has a clean interface for integrating heuristics that produce the control information, i.e. the cut between f and g. This demonstrates the clear division between the design space exploration and transformation in our concept. The heuristic has nothing to do with logic, and as a consequence, switching from one heuristic to another requires no change in the theorem or in the retiming procedure.

From the circuit designer's point of view, synthesis tools based on HASH are the same as conventional synthesis tools. During synthesis, everything is performed automatically: the transformational procedure adapts the theorem to the current task. Logic related user interaction (proof search) is not required from the circuit designer's part.

C. Faulty Heuristics

The determination of the cut in step 1 may be performed arbitrarily. It is possible to do it by hand, and it is also possible to invoke some program. This allows us to reuse existing techniques [11], [12]. The decision on how to cut does not violate correctness. If a cut was given, that does not match our pattern, then our transformation would fail, since the general retiming theorem could not be matched and an exception will be raised, implicitly. An "incorrect" theorem however cannot be derived due to the principle of theorem provers.

To illustrate this point, let us choose f to consist of the comparator and the multiplexer and g to consist of the incrementer (fig. 4). During the first step of the retiming procedure the output and transition function is transformed into an equivalent output and transition function consisting of two subfunctions f and g. It is not possible to find such a split and therefore trying to derive such circuit will fail at some point in HOL. In our implementation, the algorithm tries to cut the combinatorial block as described in figure 4. As can be seen, the original function has a triple representing the state variables and the falsely split function has 4 state variables. The equality of the old and the new combinatorial block cannot be derived — it is even impossible to express the equality due to the fact that the left and the right hand side would have different types. In HOL, this results in an exception when trying to build the equality expression.

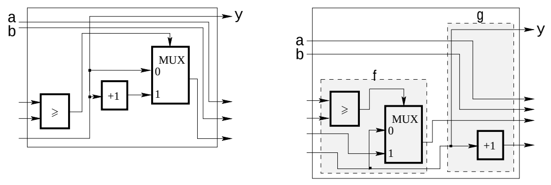

Fig. 4. False Cut of the Combinatorial Part

V. Experimental Results

We applied the formal retiming step to the example given in figure 2 and to the sequential circuits from the IWLS'91 benchmarks set. The results are listed in table I and table II, respectively.

n	#flipflops	#gates	SIS	SMV	HASH
1	3	4	0.4	0.1	0.2
2	6	8	0.4	0.1	0.2
3	9	12	0.4	0.1	0.3
4	12	16	0.8	0.1	0.3
5	15	20	1.2	0.1	0.3
6	18	24	2.4	0.3	0.4
7	21	28	8.4	2.0	0.4
8	24	32	55.3	18.7	0.4
9	27	36	284.0	213.3	0.4
10	30	40	1487.5	-	0.5
40	120	160	-	-	1.5
80	240	320	-	-	2.7
120	360	480	-	-	4.3
160	480	640	-	-	5.8
200	600	800	-	-	7.9
240	720	960	-	-	9.9

TABLE I
Example from Figure 2

name	#flipflops	#gates	Eijk/1	Eijk/2	SIS	HASH
s208.1	8	104	0.5	0.7	1.2	4.4
s298	14	119	29.6	15.7	1.6	14.3
s420.1	16	218	598.4	94.8	1007.6	17.7
s510	6	211	3.2	42.7	7.3	28.7
s526	21	193	178.2	115.8	49.3	37.6
s838.1	32	288	-	-	-	83.0
s1196	18	529	?	?	3.1	61.1
s1423	74	657	?	?	-	149.5
s1488	6	653	1.6	3.3	1.7	155.1
s1494	6	647	?	?	1.6	153.7

TABLE II
IWLS'91 Benchmarks

The example in figure 2 is scalable with the bitwidth n of the data signals. We compared our results to the verification results achieved with different post-synthesis-verification approaches. The synthesis environment SIS provides a finite state machine comparison technique [13]. SMV is a multi-purpose model checker [14]. Van Eijk presented a model checker (indicated with Eijk/1) and an advanced version that exploits functional dependencies (indicated with Eijk/2) [7].

All times are given in seconds. The benchmarks have been run on an Ultra Sparc with 200 Mb except for the results in the columns Eijk/1 and Eijk/2, which have been taken from the paper [7] and were run on an HP 9000. The dash (-) indicates, that the benchmark could not be processed in reasonable time and the question mark indicates that no results were reported in [7].

We found out, that in our approach, the time consumption depends on the size of the circuit but is quite independent from the cut. Due to step 4 (see section IV-A), it becomes a little slower for large sized functions f. In table I and table II, we performed a retiming with f covering a maximum number of retimable gates, i.e. the worst case for our approach.

The complexity of model checking depends on the size of the combinatorial part and on the maximum number of steps needed to reach all states. In general the size of the BDDs for representing the currently covered states and transforming this set increases exponentially with the size of the combinatorial part (see figures I and II).

It turns out, that our approach can be applied to circuits with sizes that are beyond what can be handled using model checking or related techniques. Our approach (indicated with HASH), requires a higher basic time consumption. This makes HASH slower for small sized circuits. For larger sized circuits, however, the time consumption increases in a moderate manner. One comes to the same result when dealing with the IWLS'91 benchmarks. Circuits s208.1, s420.1 and s838.1 are all fractional multipliers with different bitwidths: 8, 16, and 32, respectively. None of the model checkers where able to verify the 32-Bit version. From the 8-Bit version towards the 16-Bit version, the time consumption increased rapidly: factor 1000 for SIS and Eijk/1 and factor 100 for Eijk/2. The corresponding factor for our approach is 4, and it is even possible to handle the 32-Bit version in a reasonable time.

The results achieved with HASH for the example from figure 2 are much better than those achieved for the IWLS'91 benchmarks. This is due to the fact, that we chose to perform the retiming on an RT-level representation, which consists of n-bit circuits whereas the model checking techniques are based on simple temporal logic and can therefore only handle flat, bit-level descriptions at the gate level. In our approach, operating at the RT-level reduces the complexity of steps 1-3. However the complexity of the initial state evaluation step (step 4) is not affected.

VI. Summary and Conclusions

We introduced a formal synthesis approach HASH, where all basic transformation steps are performed by rule applications within a theorem prover and applied this approach to retiming. HASH also provides various other synthesis related transformations on synchronous circuits such as state encoding, signal encoding and the elimination of redundant parts encoding (see also [10]).

This approach increases the reliability of the synthesis program, since the correctness only depends on the core of the theorem prover whereas in conventional synthesis programs there is no such core and every error in the synthesis tool may affect the correctness of synthesis results. We have shown, that it is possible to write formal synthesis programs without really inventing new algorithms but by exploiting conventional synthesis programs and giving them a formal basis. This implies that the quality of designs produced using HASH is the same as that of a conventional synthesis tool. Furthermore, since the interaction within HASH is the same as that of a conventional synthesis tool, its acceptance among designers is eased.

References

[1] S.D. Johnson, R.M. Wehrmeister, and Bhaskar Bose, "On the interplay of synthesis and verification," In Claesen [15], pp. 385–404.

[2] AHL, *Lambda Reference Manual*, 1989.

[3] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401–424, Apr. 1994.

[4] J.C. Madre, "Benchmarks for tautology checking — experimental results," In Claesen [15], pp. 575–579.

[5] A. Gupta, "Formal hardware verification methods: A survey," *Formal Methods in System Design*, vol. 1, no. 2/3, pp. 151–238, 1992.

[6] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid, "Formal synthesis in circuit design - A classification and survey," in *Formal Methods in Computer-Aided Design, FMCAD'96*, Palo Alto, USA, 1996.

[7] C.A.J. van Eijk and J.A.G. Jess, "Exploiting functional dependencies in finite state machine verification," in *The European Design & Test Conference*, Paris, France, Mar. 1996, IEEE Computer Society and ACM/SIGDA, pp. 9–14, IEEE Computer Society Press.

[8] Huang, Cheng, and Chen, "On verifying the correctness of retimed circuits," in *Great Lakes Symposium on VLSI*, Ames, USA, Mar. 1996.

[9] M.J.C. Gordon and T.F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

[10] D. Eisenbiegler and R. Kumar, "An automata theory dedicated towards formal circuit synthesis," in *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, E.T. Schubert, P.J. Windley, and J. Alves-Foss, Eds., Aspen Grove, Utah, USA, Sept. 1995, number 971 in Lecture Notes in Computer Science, pp. 154–169, Springer-Verlag.

[11] C. Leisersohn, F. Rose, and J. Saxe, "Optimizing synchronous circuits by retiming," in *Caltech Conference on VLSI*, 1983, pp. 87–116.

[12] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vinentelli, "Retiming and resynthesis: Optimizing sequential circuits with combinatorial techniques," in *IEEE Transactions on CAD*, Jan. 1991, pp. 74–91.

[13] E. M. Sentovich, K. J. Singh, L. Lavagno, and C. Moon et al, "SIS: A system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, University of California, Berkeley, 1992.

[14] K.L. McMillan, "The SMV system, symbolic model checking - an approach," Tech. Rep. CMU-CS-92-131, Carnegie Mellon University, 1992.

[15] Luc J. M. Claesen, Ed., *Applied Formal Methods For Correct VLSI Design*, vol. 2. IMEC-IFIP, Elsevier Science Publishers, 1989.