# Possibilities, Limitations and Problems in Retiming — a View from a Logical Perspective

Dirk Eisenbiegler
Institut for Circuit Design and Fault Tolerance
(Prof. Dr.-Ing. D. Schmid)
University of Karlsruhe
e–mail: Dirk.Eisenbiegler@informatik.uni-karlsruhe.de

December 6, 1996

**Abstract**

This paper gives a formal description of retiming and analyzes its possibilities. The paper is based on a theory for automata in HOL, which is dedicated towards formal hardware representation and transformations. In this approach hardware is represented by automata descriptions and formal synthesis is performed by applying formally proven theorems.

## 1 Introduction

Performing the synthesis of circuits correctly is essential due to the costs arising from error prone implementations. In the beginning of circuit design, circuits have beeen designed by hand. Nowadays, the synthesis process has been more and more automated. This increases the reliability of circuit design, especially for large sized circuits (socalled correctness by design). However, nowadays the correctness of synthesis essentially depends on the correctness of the synthesis programs involved.

This paper is dedicated towards one specific synthesis step: the retiming of synchronous circuits [LeRS83, SSLM92]. Retiming can be performed on the gate level as well as on the RT level. Simple synthesis programs for logic minimization (two-level optimization, multi-level optimization) and technology mapping only transform the combinatorial part of the circuit whereas the memory part is not affected. Such transformations are primitive in a sense that they only substitute the ncombinatorial part by an equivalent one. On the gate level such transformations can easily be described by simple rule applications within some boolean calculus.

Retiming goes beyond this. Retiming also affects the state representation. Therefore retiming cannot be verified by means of simple boolean argumentation. This makes retiming a magnitude more complex. Applying automated

general purpose verification techniques is not suitable due to the extreme time and memory consumption [BCLM94, Melh93]. Also specialized general-purpose verification techniques [EiJe96] and even retiming specific verification techniques [HuCC96], that are only designed for verifying pure retiming steps, are not applicable for medium or large sized circuits.

## 2 What this Paper is not about

This paper is restricted to retiming where the synchronous is not affected. In some publications, the term retiming is also used in a different manner.

In [Shee88], for example, only acyclic data flow graphs with intermediate registers are considered. The operations performed definitely change the synchronous behaviour. Registers are not only shifted but the operations performed also affect the number of intermediate register layers thus leading to different execution times in terms of clock cycles.

In such approaches, the relation between the original and the transformed circuit is not that trivial any more. It is sort of an equivalence from an algorithmic level point of view: the algorithm remains unchanged but there is a different interface behaviour. Such techniques go beyond the limits of RT-level descriptions. Due to the fact that such transformations cannot be considered from the RT-level only but are equivalent only in terms of a common algorithmic description, they are pretty close to scheduling as it is performed in high-level synthesis [CaWo91].

## 3 Automata Descriptions

The definition of the automaton constant is to be performed in a conservative manner. It has to be prevented to add some extra axioms to the HOL theory that violate consistency. Therefore HOL provides a set of four basic mechanisms for extending the theory in a consistent manner, and there are also some compound mechanisms based on them.

At first glance, the definition of automaton may seem a little awkward and involved: first a primitive recursion is performed for defining some auxiliary function automaton' and then automaton itself is defined via a constant definition. However, the primitive recursion and constant definition are consistency preserving mechanisms.

automaton' is similar to automaton except that the state is also visible (see figure 1). Other than with automaton, the pair $(f, q)$ is mapped to a function that maps the input to a compound output/state-signal. automaton' has the following type:

$$((\iota \times \sigma \to \omega \times \sigma) \times \sigma) \ \to \ (\mathsf{num} \to \iota) \to (\mathsf{num} \to (\omega \times \sigma))$$

automaton' is defined by means of primitive recursion over natural numbers, which represent time. For a given *input*, the expression (automaton'$(f, q)$ *input*)

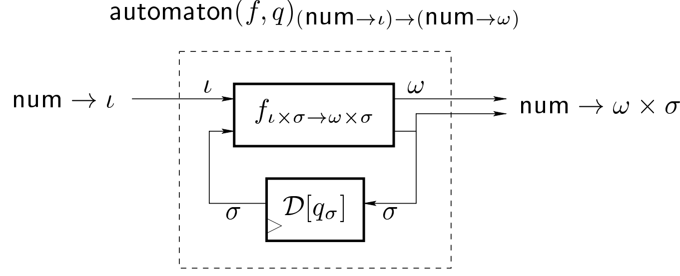


Figure 1: Automaton'

denotes the output and the present state and (automaton' $(f,q)$ *input* $t$) denotes the output and the present state at some time $t$. The definition to follow is performed by using primitive recursion over $t$.

The output and the next state for some time $t$ can be obtained by applying $f$ to the pair of current input $input(t)$ and current state $s$. In the beginning $t$ is 0 and the automata is in the initial state $s = q$. For all other times $t = (\mathsf{SUC} t')$, the next state of the output is defined using the current input $input(\mathsf{SUC}\, t')$ and the current state $s$. Since (automaton' $(f,q)$ $input\, t'$) produces a pair corresponding to the output and the state, the function SND is applied in order to extract the state from this result.

Remark: Throughout this paper, the function FST and SND will be used. They are predefined functions in the HOL theorem prover. FST maps a pair to its first component, SND maps a pair to its second component.

`Definition:` (1)

$$\begin{aligned}
\vdash\ & \big(\,\mathsf{automaton'}\ (f,q)\ i\ 0\ =\ f(i(0),q)\,\big)\ \wedge \\
& \big(\,\mathsf{automaton'}\ (f,q)\ i\ (\mathsf{SUC}\ t')\ = \\
& \quad \mathsf{let} \\
& \quad\ \ s = \mathsf{SND}(\mathsf{automaton'}\ (f,q)\ i\ t') \\
& \quad \mathsf{in} \\
& \quad\ f(i(\mathsf{SUC}\ t'),s)\,\big)
\end{aligned}$$

Now automaton can be defined by as:

`Definition:` (2)

$$\vdash\ \mathsf{automaton}\,(f,q)\ i\ t\ =\ \mathsf{FST}(\mathsf{automaton'}\ (f,q)\ i\ t)$$

## Relation between Input, State and Output

The following theorem describes the bevaviour of automaton in a more natural way. Other than (1) and (2), it has not been invented via definition but has been formally derived.

$\mathsf{automaton}(f, q)$ is a function mapping some time dependent input signal $input_{\mathsf{num}\to\iota}$ to some time dependent signal $output_{\mathsf{num}\to\omega}$. Let $state_{\mathsf{num}\to\sigma}$ be a signal, where the initial state $state(0)$ equals $q$ and where $f$ is used to iteratively produce the succeeding states. Then $state(t)$ represents the state of the automaton at time $t$ and one can determine the output at time $t$ by applying $f$ to $input(t)$ and $state(t)$.

$$\begin{array}{ll} \vdash & (output = \mathsf{automaton}\ (f, q)\ input) \\ & = \\ & \forall state. \\ & \quad (state(0) = q) \wedge \\ & \quad (\forall t.\ state(\mathsf{SUC}(t)) = \mathsf{SND}(f(input(t), state(t))))\ ) \\ & \quad \Rightarrow output(t) = \mathsf{FST}(f(input(t), state(t))) \end{array} \tag{3}$$

Figure 2 sketches the relation between the signals involved. For some specific automaton, the initial state at time 0 as well as the output and transition function $f$ are pregiven. For a given input signal, the states for $t > 0$ and the outputs are computed by applying $f$ according to figure 2.

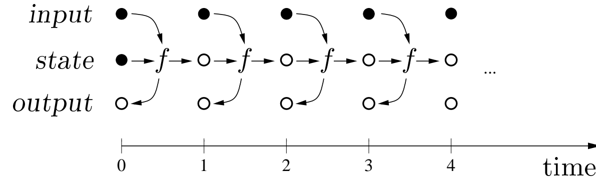


Figure 2: Input, State and Output of an Automaton

# 4 On the Equality of Automata

Two automata represented by $(f^1, q^1)$ and $(f^2, q^2)$ are equal whenever they represent the same input-to-output-function, i.e.

$$\mathsf{automaton}(f^1, q^1) = \mathsf{automaton}(f^2, q^2)$$

This report is concerned with an equivalence transformation on automata. The constant $\mathsf{automaton}$ can be considered as a characteristic function for the equivalence relation on pairs $(f, q)$.

The following theorem describes the the relation between the signals of two equal automata $\mathsf{automaton}(f^1, q^1)$ and $\mathsf{automaton}(f^2, q^2)$ (see also Figure 3). The first starts with state $q^1$, the latter starts with $q^2$. Given that they are both wired to the same input signal, they both produce the same output by

iteratively applying $f^1$ and $f^2$ respectively.

$$
\begin{aligned}
\vdash\ & (\mathsf{automaton}\ (f^1, q^1) = \mathsf{automaton}\ (f^2, q^2)) \\
& = \\
& \forall state^1, state^2. \\
& \quad (state^1(0) = q^1) \wedge \\
& \quad (\forall t.\ state^1(\mathsf{SUC}(t)) = \mathsf{SND}(f^1(input(t), state^1(t)))) \\
& \quad (state^2(0) = q^2) \wedge \\
& \quad (\forall t.\ state^2(\mathsf{SUC}(t)) = \mathsf{SND}(f^2(input(t), state^2(t)))) \\
& \quad \Rightarrow \mathsf{FST}(f^1(input(t), state^1(t))) = \mathsf{FST}(f^2(input(t), state^2(t)))
\end{aligned} \tag{4}
$$

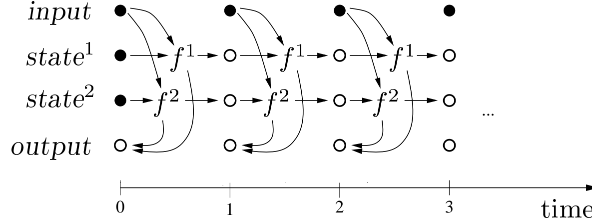


Figure 3: Input, State and Output of two equal Automata

There are two cases for equality of automata. In the trivial case, $f^1$ equals $f^2$ and $q^1$ equals $q^2$. Being equal implies having the same type. Therefore, in this case, $\sigma^1$ and $\sigma^2$ must be equal. When performing bisimulation (figure 3), the states are always equal, i.e. $\mathsf{state}^1(t) = \mathsf{state}^2(t)$.

In the nontrivial case, the two automata are equal although $f^1$ does not equal $f^2$ and $q^1$ does not equal $q^2$. There may even be different data types for the internal states. So the "expressions" $q^1 = q^1$ and $f^1 = f^2$ are not even well-formed, due to a type mismatch.

In circuit design, combinatorial optimizations or simple functional optimizations on the RT-level correspond to the trivial case. It is pretty easy to describe such transformations in logic. For combinatorial optimizations, operations within a boolean calculus will do the job.

More sophisticated synthesis procedures such as state encoding, state minimization and retiming correspond to the nontrivial case. This paper is dedicated towards retiming which requires a nontrivial transformation.

# 5 Retiming

In simplified terms, retiming (more precisely: forward retiming) moves the memory part over $g$. In order to guarantee correctness, the initial state $q$ has to be transformed from $q$ to $g(g)$ (see figure 4). Retiming can significantly change the delay of the combinatorial part of the circuit and therefore increase the clock frequence. One also has to consider, in general retiming also has an impact on

the number of memory units needed. Combining Retiming with combinatorial optimizations may even change the consumption of combinatorial units.

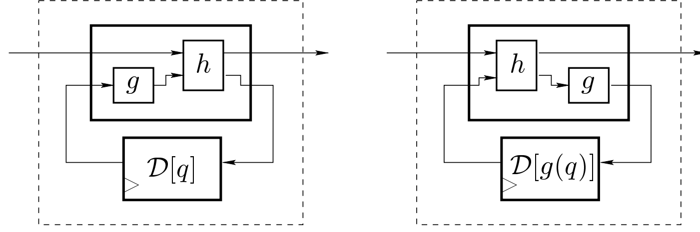


Figure 4: Retiming

## The Retiming Theorem

The following theorem describes retiming in a very general manner. It states that the two automata descriptions in figure 4 are equal. Both automata consist of two combinatorial subparts $f$ and $g$. The theorem is a mighty higher order logic expression stating that this equality holds for all $f$ and all $g$. Therefore the theorem 5 is not dedicated to a specific retiming step but describes a general pattern for retiming. As described later on, it can be adapted to different situations.

$$\vdash \; \mathsf{automaton}(\; (\lambda(i,s).\, h(i,g(s)))\;,\; q\;) \\ = \mathsf{automaton}(\; (\lambda(i,s).\, \mathsf{let}\; (x,y) = h(i,s)\; \mathsf{in}\; (x, g(s)))\;,\; g(q)\;) \tag{5}$$

## Applying the Retiming Theorem

Retiming can be performed in both directions. The synthesis step from left to right (figure 4) is called forward retiming whereas the reverse direction is called backward retiming. In both directions it is possible to apply the theorem in various ways.

Using an automaton as a formal representation, the overall forward 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 may be invoked.

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 5 describes, how a circuit is adapted to the 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.

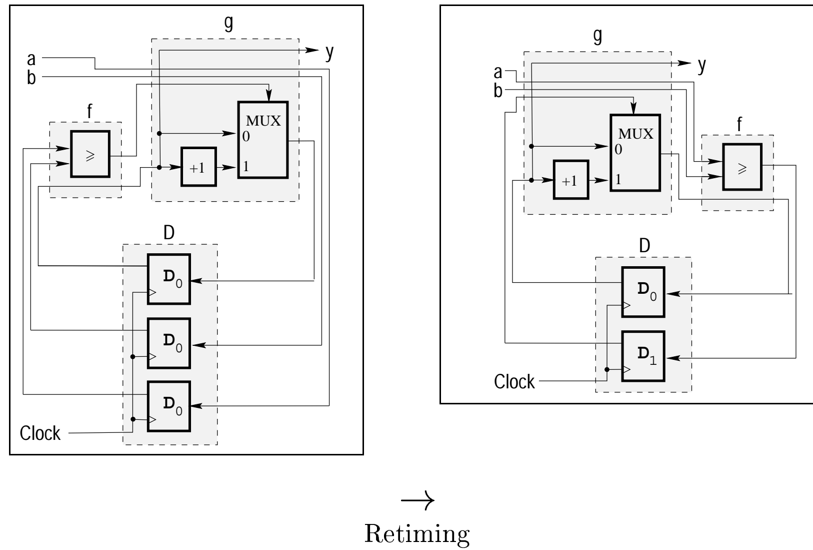


Figure 5: Example for Applying the Retiming Scheme

## Forward Retiming and Backward Retiming

At first glance, backward retiming is just the other way round. The current automaton has to be matched with the right hand side and the theorem has to be applied in reverse direction. However, determining the new initial state is not that easy any more since one has to apply the "inverse" of $g$. In general, there is no such "inverse" or it is not unambiguous. There may be several initial states fulfilling this property and it may even be, that there is none.

Up to now, only unambiguous circuit descriptions have been considered, i.e. each circuit represents exactly one concrete circuit. However, when performing a backward retiming step with several possible initial states, this is a synthesis step where the result should be a set of circuits rather than a single circuit. There are several formalism where circuits need not necessarily be specified in an unambiguous manner but can loosely be specified (don't cares etc.). Such formalisms do not describe single circuits but sets of circuits. Just picking out one of the circuits and omit the rest may lead to a loss of optimization since further optimizations steps may produce good results only for the omited ones.

Dealing with circuit descriptions that do not ensure unambiguity makes things a magnitude more difficult. One has to be aware of the fact that in general such circuit descriptions do not ensure consistency. Deriving inconsistent circuit descriptions is fine as far as logic is concerned. Inconsistent circuit descriptions fulfill any specification. So without looking at consistency, constructing correct circuit descriptions for arbitrary specifications is pretty easy. From the practical point of view, however, such circuit descriptions are both worthless and misleading. There is just no circuit in the real world that such circuit descriptions stand for.

## Possibilities and Limitations

In forward retiming, the combinatorial part has to be cut according to the left hand side of figure 4. During retiming, the components of the combinatorial part have to be assigned to either $f$ or $g$. However, not all assignments are possible. Components can only be assigned to $g$ if they only depend on the states or on the results of other components that are assigned to $g$. Components in $g$ must not — neither directly nor indirectly — depend on the overall inputs of the combinatorial part.

To perform backward retiming, the components assigned to $g$ must not — neither directly nor indirectly — depend on the overall outputs of the combinatorial part. In order to avoid inconsistency, backward retiming should also be restricted to functions $g$ such that there exists an inverse for the current initial state with respect to $g$.

Figure 6 describes a typical situation before retiming. One can statically analyze which retiming steps can be performed. As to forward retiming, C2 and C3 cannot be assigned to $g$ due to the dependencies from the input signals. Furthermore data dependencies within C1, C4, C5 and C6 have to be respected. Assigning C4 to $g$ and assigning the other components to $f$ would lead to a proper split of the combinatorial part. The new initial state would become $(0, 1, 0)$. Assigning C1 and C6 to $g$ and the rest to $f$, however, would fail since C6 (a component within $g$) depends on the result of C4 (a component within $f$).

As to backward retiming, the components C1 and C2 cannot be assigned to $g$ due to their impact on the output signal. Similar to forward retiming, data dependencies again have to be considered. For example, it is not possible to assign C4 to $g$ and to assign C6 to $f$.

In general, backward retiming does not lead to unambiguous initial states. Let C3 be assigned to $g$ and the other components be assigned to $f$. In the retimed circuit, four 1-Bit D-flipflops are required: two at the outputs of C5 and C6 and two at the inputs of C3. The D-flipflops at the inputs of C3 can be initialized with $(1, 0)$, $(0, 1)$ or $(1, 1)$.

Besides data dependencies on the output, there is also a second restriction for backward retiming: for some cuts, there is no proper initial state. Let C5 and C6 be assigned to $g$ and the other components be assigned to $f$. For the output signal of C4 — an input for both C5 and C6 — the requirements are

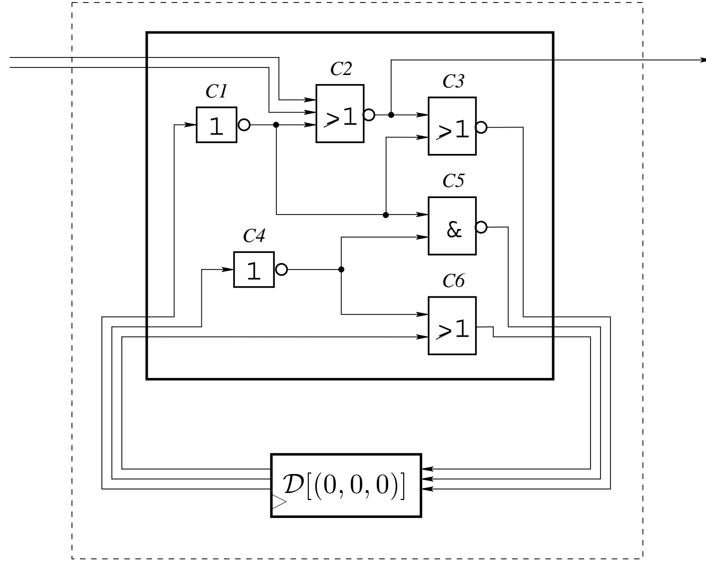


Figure 6: Example for Retiming

contradictory. According to C5 the its initial state should be 1 and according to C6 its initial state should be 0. So for this cut there is no proper initial state.

# 6 Conclusions

In this paper, retiming of circuits has been described in a formal manner. Based on these exact formal representations, variations and possibilities but also possible problems have been discussed.

It shows that it is worthwhile having a closer look at synthesis from a logical point of view. Finding adequate formal representation is a first step towards this goal. Consistency of formal circuit representations is a preliminary. Formal circuit correctness only makes sense if the chosen circuit representations really do represent real circuits.

In higher order logic it is possible to describe and prove general patterns for synthesis steps in general (quantification over functions). This leads to formal synthesis tools, where synthesis is performed by applying such theorems within a theorem prover [EiKB97, BlEK96]. On the other hand, arguing with such theorems also leads to a better understanding of correctness of synthesis and bugs within implementations.

## References


[BCLM94] 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*, 13(4):401–424, April 1994.

[BlEK96] C.Blumenröhr, D. Eisenbiegler, and R.Kumar. Applicability of formal synthesis illustrated via scheduling. In *Workshop on Logic and Architecture Synthesis*, Grenoble, France, December 1996. Institut National Polytechnique de Grenoble.

[CaWo91] R. Camposano and W. Wolf. *High-Level VLSI Synthesis*. Kluwer, Boston, 1991.

[EiJe96] C.A.J. van Eijk and J.A.G. Jess. Exploiting functional dependencies in finite state machine verification. In *The European Design & Test Conference*, pages 9–14, Paris, France, March 1996. IEEE Computer Society and ACM/SIGDA, IEEE Computer Society Press.

[EiKB97] Dirk Eisenbiegler, R.Kumar, and C.Blumenröhr. A constructive approach towards correctness of synthesis application within retiming. In *The European Design & Test Conference*, Paris, France, March 1997. IEEE Computer Society and ACM/SIGDA, IEEE Computer Society Press.

[HuCC96] Huang, Cheng, and Chen. On verifying the correctness of retimed circuits. In *Great Lakes Symposium on VLSI*, Ames, USA, March 1996.

[LeRS83] C. Leisersohn, F. Rose, and J. Saxe. Optimizing synchronous circuits by retiming. In *Caltech Conference on VLSI*, pages 87–116, 1983.

[Melh93] T. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, 1993.

[Shee88] M. Sheeran. Retiming and slowdown in ruby. In George J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 289–308, Glasgow, Scotland, July 1988. IFIP WG10.2 Working Conference, North-Holland.

[SSLM92] E. M. Sentovich, K. J. Singh, L. Lavagno, and C. Moon et al. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, 1992.