Automata — A Theory Dedicated towards Formal Circuit Synthesis

Dirk Eisenbiegler
Institut für Rechnerentwurf und Fehlertoleranz
(Prof. Dr.-Ing. D. Schmid)
Universität Karlsruhe
e–mail: Dirk.Eisenbiegler@informatik.uni-karlsruhe.de

June 13, 1997

Abstract

This is a technical report about a theory named Automata. Automata is an arithmetic for synchronous circuits. It provides means for representing and transforming circuit descriptions at the RT level and gate level in a mathemtical manner. Automata has been implemented in the HOL theorem proving environment. Preproven theorems are designed for performing standard synthesis steps such as state encoding, retiming and state minimization in a mathematical manner via logical derivation.

Contents

Chapter 1

Introduction

This paper is dedicated towards formal correctness in hardware design at the RT (register transfer) and gate level. During RT and gate level synthesis the circuit description is altered step by step using specific well known transformations such as state encoding, state minimization, boolean optimization, etc. Although these basic synthesis steps conform to simple logical derivation steps, post-synthesis-verification is exacting. Post-synthesis-verification techniques only have access to a specification and an implementation, i.e. the input and the output of the synthesis process. Usually, there is a big gap between specification and implementation: the state representation, the originally given partitioning and the naming of the subcomponents and interconnections may have changed completely. As a major drawback, the information on *how* the implementation was derived from the specification is lost. Much of this information is essential for verification: How were the control states encoded? Where is which data stored? Is a redundant data representation used (one-hot-encoding, signed-digit-encoding etc.)? Which control states were eliminated because of unreachability, or have some (unreachable) control states been added in order to get a more efficient/better testable implementation? Which parts of the gate level implementation belong to the control path/data path of the RT-level description? etc.

This paper is part of our ongoing work for developing techniques to perform formally correct synthesis of synchronous circuit descriptions. The automata theory is intended to be used for simple synchronous circuit descriptions at the gate level and RT level [EiSK93]. The theory provides theorems describing the above mentioned elementary RT- and gate level transformations (data encoding, state minimization etc.) in a logical manner. The automata theory builds a basis for formal synthesis programs where the entire process is described by a sequence of refinement steps within logic. As a result of the formal synthesis process, there is not only the implementation of a given specification but also the proof of its correctness. In contrast to other approaches towards formal synthesis, this approach is very close to conventional synthesis techniques. It is not intended to invent new synthesis algorithms but implement conventional

ones in a formal manner.

Similar to the approach taken in this paper, [Loew92, Day92] also gives a formalization of automata descriptions. This allows reasoning about automata. However, they allow more expressive specifications such as non-deterministic automata and do not provide transformations corresponding to circuit synthesis. In our work we consider only deterministic translating automata (transductors) that directly correspond to sequential circuits. Their formalization is purely functional in nature.

The outline of this paper is as follows: In the next chapter, the formalism for representing automata is introduced along with its semantics. The definitions within the first section build the basis for the entire paper. The third chapter describes two specialized forms of automata: pure combinatorial blocks and pure memory blocks. The fourth chapter is dedicated towards reachability of states and state traversation. The fifth chapter describes synthesis related transformations on automata. There is a set of theorems supporting state encoding, state minimization, retiming and the elimination of redundant parts. State encoding requires proper encoding functions. Chapter six presents techniques for deriving state encodings in a systematic manner. Chapter seven is dedicated towards structures of automata. The theorems described in this chapter bridge the gap between functional circuit descriptions with automata and the relational circuit description style. Furthermore, they provide a means for applying the above mentioned automata transformations also to parts of automata.

Chapter 2

Automata Representation

Usually an automaton is represented by a 6-tuple consisting of input alphabet, output alphabet, set of states, output function, transition function and initial state. In this approach, an automaton will be represented by a pair (f, q), where f is a compound output and transition function and q is the initial state. The way automata are represented in this approach differs in two aspects: First the input alphabet, the output alphabet and the set of states are not given explicitely and second the output function and the transition function are combined to a single output and transition function.

Throughout this paper typed higher order logic expressions will be used. A type is assigned to each expression and only well typed expressions are allowed. Let ι, ω and σ be the types corresponding to the input values, output values and state values, respectively. ι, ω and σ indicate type variables, i.e. arbitrary types. It is to be noted here, that the types ι, o and σ may also be compound types such as tuples of basic data types or even tuples of compound types.

In circuit design, the output function and the transition function correspond to the combinatorial units. However, it is not possible to unambiguously assign combinatorial units to either the output function or the transition function. In general, combinatorial units may be part of the output function as well as part of the transition function. In other words: Their output signal is lead to the output or to combinatorial units that produce the output, and it is also lead to the registers or to combinatorial units that produce the new register values. In circuit synthesis combinatorial transformations are not restricted to the transformational part or the output part of the combinatorial units. Therefore, in this approach, the output function and the transition function are combined to a single output and transition function f whose type is $\iota \times \sigma \rightarrow \omega \times \sigma$. The type of the initial state q is σ. The entire automaton is represented by a pair (f, q) whose type is

$$(\iota \times \sigma \rightarrow \omega \times \sigma) \times \sigma$$

An automaton (transductor) describes a mapping from a time dependent input signal to a time dependent output signal. The type num stands for natural

numbers and will be used to indicate time. Time dependent signals are mappings from num to some data type. $\mathsf{num} \to \iota$ indicates the type of the input signal and $\mathsf{num} \to \omega$ indicates the type of the output signal. Therefore an automaton defines a mapping of the following type

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

f and q unambiguously determine, how the automaton maps a time dependent input signal *input* of type $\mathsf{num} \to \iota$ to a time dependent output signal *output* of type $\mathsf{num} \to \omega$. The higher order logic function automaton describes the semantics of an automaton given as (f, q). The expression $\mathsf{automaton}(f, q)$ indicates a function that maps *input* to *output*. Therefore, automaton is a function of the following type:

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

Figure 2.1 sketches, how some $\mathsf{automaton}(f, q)$ could be "implemented" using a combinatorial component realizing f and a memory unit $\mathcal{D}[q]$. Throughout the paper, the symbol $\mathcal{D}[q]$ will be used to indicate a simple memory unit, where the signal is delayed by one clock cycle and where the initial output is q.

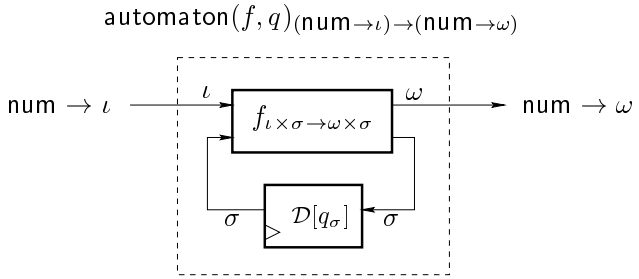


Figure 2.1: automaton

2.1 Definition

Before defining the automaton constant itselft, the constant state is introduced. state is introduced via primitive recursion over time and automaton is defined as an abbreviation for a compound expression, i.e. a simple equation with automaton on the left hand side.

Remark: FST, SND and SUC are predefined functions of the HOL theorem prover. FST maps a pair to its first component, SND maps a pair to its second component and SUC increments a natural number.

`Definition:` (2.1)

$$\begin{aligned}\vdash\quad & \big(\,\mathsf{state}\,(f,q)\;i\;0\;=\;q\,\big)\;\wedge\\ & \big(\,\mathsf{state}\,(f,q)\;i\;(\mathsf{SUC}\,t)\;=\mathsf{SND}(\,f(i(\mathsf{SUC}\,t),\mathsf{state}\,(f,q)\;i\;t)\,)\,\big)\end{aligned}$$

For a given automton representation (f, q), $\mathsf{state}(f, q)$ describes the state of an automaton as a sequence, i.e. a mapping from a time dependent input signal to a state signal, i.e. a mapping from num to σ. state has the following type:

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

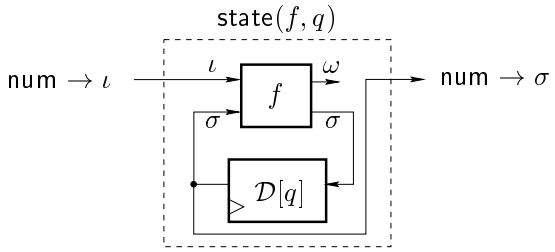


Figure 2.2: state

state is defined by means of primitive recursion over natural numbers, which represent time. For a given *input*, the expression (state (f, q) *input*) denotes a state in terms of a function mapping time represented as num to data values of type ι. The expression (state (f, q) *input* t) denotes the state at some time t.

Based on state, $\mathsf{automaton}$ is defined as follows:

`Definition:` (2.2)

$$\vdash \quad \mathsf{automaton}\ (f, q)\ i\ t \;=\; \mathsf{FST}(f(i(\mathsf{SUC}\ t), \mathsf{state}\ (f, q)\ i\ t))$$

Figure 2.3 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 pre-given. For some input signal, the states for $t > 0$ and the outputs are computed by iteratively applying f.

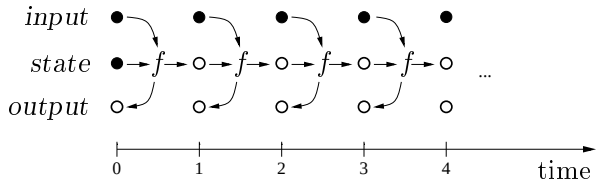


Figure 2.3: Input, State and Output of an Automaton

2.2 Equivalence of Automata

Let (f^1, q^1) and (f^2, q^2) be two automata with types ι^1, ω^1, σ^1 and ι^2, ω^2, σ^2, respectively. The two automata are equivalent 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 puts a stress on transforming automata to equivalent ones. The constant automaton is the corresponding characteristic function for the equivalence relation between pairs (f, q).

Formulae are only considered to be wellformed when they are well typed. The above equation is well typed only if the function of the left hand side and the function of the right hand side have the same type of input values and output values ι and ω. Therefore having the same input and output type is a preliminary for equivalence of automata. However, automata may be equivalent although the type for representing the states σ differs.

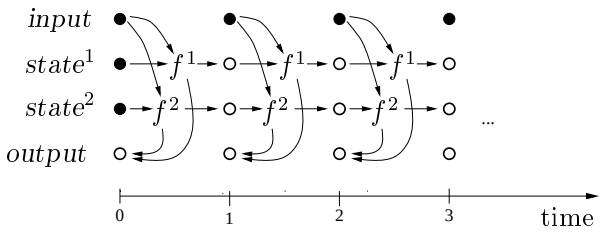


Figure 2.4: Input, State and Output of two equal Automata

There are two cases for equivalence 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, σ^1 and σ^2 must be equal. When performing bisimulation (figure 2.4), 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" $f^1 = f^2$ and $q^1 = q^2$ are not even wellformed, 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. Encoding symbolic states is one step to transform RT-level descriptions down to the gate level. Retiming and reencoding steps can be used to achieve optimizations that go beyond of what is possible with trivial transformations. This paper is dedicated to nontrivial circuit transformations.

2.3 Compound Output/Transition-Functions

Compound functions can be described in the λ-notation supported by HOL. The following term represents the output and transition function f of the automaton displayed in figure 2.5. f is described as a mapping from some pair $((a, b), (c, d))$ to some pair $(x, (a, y))$ via a composition of the functions g, h and j. (a, b) are inputs, (c, d) represent the old state, x is the output and (a, y) indicate the next state.

$\lambda((a, b), (c, d)).$
  let $(v, w) = g(b, c)$ in
  let $y = h(w, d)$ in
  let $x = j(v, y)$ in
    $(x, (a, y))$

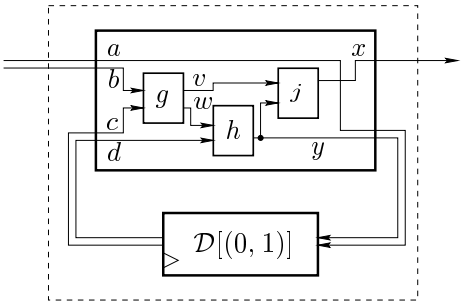


Figure 2.5: Compound Output/Transition-Functions

Using the above output and transition function f, the automaton of figure 2.5 can be represented by:

automaton(
  $\lambda((a, b), (c, d)).$
    let $(v, w) = g(b, c)$ in
    let $y = h(w, d)$ in
    let $x = j(v, y)$ in
      $(x, (a, y))$
  ,
  $(0, 1)$
)

2.4 Product Automata

The equivalence of two automata can be proven by proving that the corresponding product automaton constantly produces the T signal. Let there be

two automata $\mathsf{automaton}(f^1, q^1)$ and $\mathsf{automaton}(f^2, q^2)$. Figure 2.6 displays, how the product automaton is built. A product automaton performs bisimulation: The input is connected to both functions f^1 and f^2. The output value produced by f^1 and f^2 are compared within a equality unit $=$ and the boolean result is the output of the product automaton.

$$
\begin{array}{ll}
\vdash & (\mathsf{automaton}\ (f^1, q^1) = \mathsf{automaton}\ (f^2, q^2)) \\
& = \\
& \forall \mathit{input}, t. \\
& \quad \mathsf{automaton}(\\
& \quad\quad \lambda(i, (s^1, s^2)). \\
& \quad\quad\quad \mathsf{let}\ (x^1, y^1) = f^1(i, s^1)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ (x^2, y^2) = f^2(i, s^2)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ z = (x^1 = x^2)\ \mathsf{in} \\
& \quad\quad\quad\quad (z,\ (y^1, y^2)) \\
& \quad\quad , \\
& \quad\quad (q^1, q^2) \\
& \quad\quad) \\
& \quad\quad \mathit{input}\ t \\
& \quad = \mathsf{T}
\end{array}
\tag{2.3}
$$

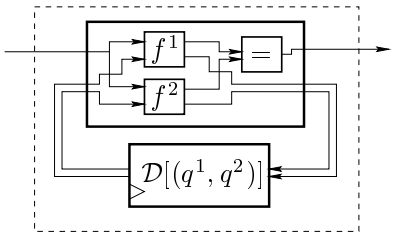


Figure 2.6: Product Automaton

2.5 Causality of Automata

The next theorem states, that the behavior of an automaton only depends on its past history. In other terms: $\mathit{output}(t)$ only depends on $\mathit{input}(x), x = 0, 1, \ldots t$. Given that two input signals are equal until some time t, than an automaton produces the same result at time t for both input signals.

$$
\begin{array}{ll}
\vdash & (\forall x.\ x \leq t \Rightarrow \mathit{input}^1(x) = \mathit{input}^2(x)) \\
& \Rightarrow\ \ (\mathsf{automaton}\ (f, q)\ \mathit{input}^1\ t = \mathsf{automaton}\ (f, q)\ \mathit{input}^2\ t)
\end{array}
\tag{2.4}
$$

Causality is a preliminary for realizability. However, this theorem is sort of academic. It is good to know, that this property holds, but the theorem is not to be used during synthesis.

Chapter 3

Combinatorial Blocks and Memory Blocks

As already mentioned, it is intended to use automaton to describe arbitrary synchronous circuits. There are two special cases of synchronous circuits: pure combinatorial circuits and pure memory blocks. In this chapter, such circuit are to be characterized by the definitions of combinatorial_block and memory_block, respectively. Afterwards the relation between such circuits and general automata is will be described.

3.1 Definitions

A *combinatorial block* can unambiguously be defined by a function $e_{\iota\to\omega}$ mapping the current input to the current output. The constant combinatorial_block maps e to a function mapping some time dependent input $i_{\mathsf{num}\to\iota}$ to some time dependent output $o_{\mathsf{num}\to\omega}$ with $o(t) = e(i(t))$. See figure 3.1.

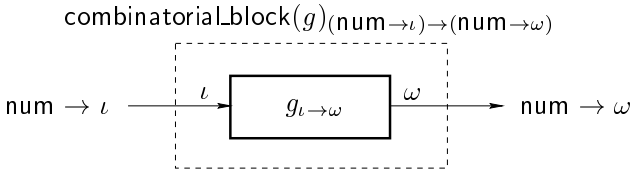


Figure 3.1: Combinatorial Block

The combinatorial block is defined as follows:

```
Definition:                                                    (3.1)
```

$\vdash$ combinatorial_block $e\ i\ t\ =\ e(i(t))$

Memory blocks delay the input by one clock cycle. The initial state is given as a parameter to the memory_block constant.

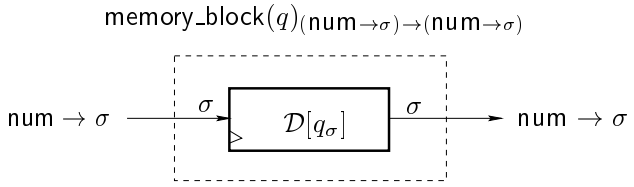


Figure 3.2: Memory Block

Formal definition by means of primitive recursion over time:

```
Definition:                                                    (3.2)
```

$$\begin{array}{lll} \vdash \ \mathsf{memory\_block}\ \mathit{init}\ i\ 0 & = & \mathit{init} \ \wedge \\ \quad \mathsf{memory\_block}\ \mathit{init}\ i\ (\mathsf{SUC}t) & = & i(t) \end{array}$$

3.2 Corresponding Automata Representations

Figure 3.3 sketches, how an automaton can be used to represent combinatorial blocks with some function g. g maps the input to the output. The state is initialized with one. one is a HOL standard data type with only one element, and the constant $\mathsf{one}_{\mathsf{one}}$ represents this unique element. The output and transition function in figure 3.3 ignores the old value of the state and always produces a one as the next state.

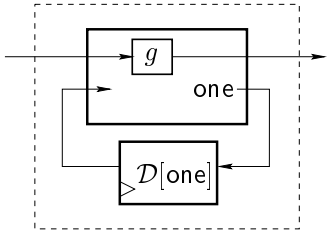


Figure 3.3: Combinatorial Block Represented by an Automaton

The following theorem states, that the automaton sketched in figure 3.3 equals the corresponding combinatorial block (see figure 3.1).

$$\vdash \ \mathsf{combinatorial\_block}\ e \ = \ \mathsf{automaton}\ ((\lambda(x, y_{\mathsf{one}}).\ (e(x), \mathsf{one})),\ \mathsf{one}) \quad (3.3)$$

Memory parts can be represented by automata, where the input is directly connected with the input of the internal memory and the output of internal memory is connected with the output of the automaton (see figure 3.4).

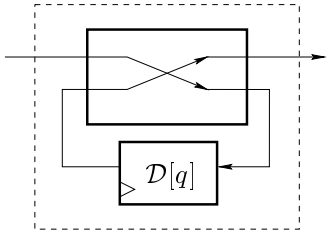


Figure 3.4: Memory Block Represented by an Automaton

The following theorem states, that the automaton sketched in figure 3.4 equals the corresponding memory block (see figure 3.2).

$$\vdash \mathsf{memory\_block}\ \mathit{init}\ =\ \mathsf{automaton}\ ((\lambda(x, y).\ (y, x)),\ \mathit{init}) \tag{3.4}$$

Chapter 4

Reachability of States

4.1 Definition

Some state is called reachable with respect to some given automaton (f, q) iff there exists some input sequence i and some time t such that the automatons state $(\mathsf{state}\ (f, q)\ i\ t)$ equals s: reachable

`Definition` (4.1)

$$\vdash\ \mathsf{reachable}\ (f, q)\ s\ =\ (\ \exists i, t.\ \mathsf{state}\ (f, q)\ i\ t\ =\ s\ )$$

The constant reachable maps an automaton (f, q) to a predicate is a characteristic function for the set of reachable states. reachable has the following type:

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

Figure 4.1 gives an example. The circles represent the states of some automaton. The initial state q is indicated by a filled circle. An arrow from state a to state b indicates, that the automaton may switch from a to b in one step. More formally: There exists an input i such that $\mathsf{SND}(f(i, b)) = a$. The dashed line describes the set of reachable states.

The following theorems ((4.2) and (4.3)) are directly derived from the definition of reachability. They state, that the initial state is reachable and that if some s is reachable then so is any successor state $\mathsf{SND}(f(x, s))$ for arbitrary input x.

$$\vdash\ \mathsf{reachable}\ (f, q)\ q \tag{4.2}$$

$$\vdash\ (\ \mathsf{reachable}\ (f, q)\ s\ )\ \Rightarrow\ (\ \forall x.\ \mathsf{reachable}\ (f, q)\ (\mathsf{SND}(f(x, s)))\ ) \tag{4.3}$$

4.2 State Traversation

This section is dedicated towards determining the set of reachable states of an automaton. It provides a set of theorems for performing state traversation within HOL.

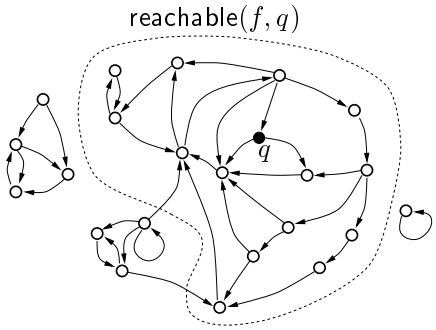


Figure 4.1: Reachability in an Automaton (f, q)

State Traversation techniques start with a set only consisting of the initial state. Within each step the set of states is extended by the states that can directly (in one clock cycle) be reached from one of the states in the current set. Finally one may reach a set of states that cannot be extended any more, i.e. there are no extra states, that can be reached from one of the states. Then the set covers all reachable states (see figure 4.2).

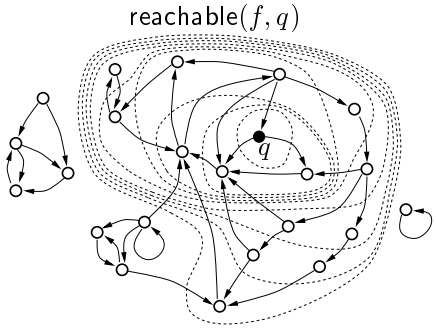


Figure 4.2: State traversation in an Automaton (f, q)

For sake of memory consumption, the elements in the sets of states are not enumerated explicitely but the set of states is represented by a characteristic functions. In conventional state traversation techniques, states have to be represented by tuples of booleans, and BDDs are used for representing the characteristic functions of the state sets.

In HOL, the set of states will be represented by characteristic functions on states of arbitrary type σ, i.e. a function of type $\sigma \to \mathsf{bool}$. In each state of the state traversation, the current set of state fulfills two properties:

1. all states of the set are reachable
2. the set contains the initial state

Definitions

The following definition characterizes these property for a set of states represented by its characteristic function P with respect to some automaton represented by f and q:

```
Definition                                                    (4.4)
```

$$\begin{array}{ll} \vdash & \mathsf{are\_reachable\_and\_contain\_initial\_state}\ (f,q)\ P = \\ & \quad (\forall x.P(x) \Rightarrow \mathsf{reachable}\ (f,q)\ x) \wedge \\ & \quad P(q) \end{array}$$

The function add_direct_successors describes the extension step. It maps some state set P to another state set $P' = \mathsf{add\_direct\_successors}(P)$ where P' covers all states of P plus the states that can directly be reached by applying the f to one of the states x of P and some arbitrary input i.

```
Definition                                                    (4.5)
```

$$\begin{array}{ll} \vdash & \mathsf{add\_direct\_successors}\ f\ P\ s = \\ & \quad P(s) \vee \\ & \quad (\exists x, i. \\ & \quad\ P(x) \wedge \\ & \quad\ s = \mathsf{SND}(f(i,x)) \\ & \quad) \end{array}$$

Theorems for Performing State Traversation

The following three theorems build the basis for the formal state traversation in HOL. They all deal with the set of states that has been determined so far, represented by its characteristic function P.

The first theorem states, that the set $\{q\}$ is a proper set, i.e. all states are reachable and the set contains the initial state. $\{q\}$ is represented by its characteristic function $P = (\lambda x.\ x = q)$.

$$\vdash \mathsf{are\_reachable\_and\_contain\_initial\_state}\ (f,q)\ (\lambda x.\ x = q) \tag{4.6}$$

The second theorem states, that given a proper set represented by P, than adding the direct successors again leads to another proper set.

$$\begin{array}{ll} \vdash & \mathsf{are\_reachable\_and\_contain\_initial\_state}\ (f,q)\ P \\ & \quad \Rightarrow\ \mathsf{are\_reachable\_and\_contain\_initial\_state}\ (f,q) \\ & \quad\ (\mathsf{add\_direct\_successors}\ f\ P) \end{array} \tag{4.7}$$

The third theorem states, that given a proper set where adding direct successors does not extend the set, than the current set equals the set of reachable states.

$$\begin{array}{ll} \vdash & \mathsf{are\_reachable\_and\_contain\_initial\_state}\ (f,q)\ P \wedge \\ & (\forall x.\ \mathsf{add\_direct\_successors}\ f\ P\ x \Rightarrow P(x)\ ) \\ & \quad \Rightarrow\ \mathsf{reachable}\ (f,q) = P \end{array} \tag{4.8}$$

Scheme for State Traversation Algorithms

The following algorithm maps some terms f and q representing the term to a theorem stating that the set of reachable states equals some P. Each state of the algorithm is represented by a theorem `th` of the following form:

$$\vdash \mathsf{are\_reachable\_and\_contain\_initial\_state}\ (f, q)\ P \tag{4.9}$$

Throughout the algorithm (f, q) will not change but P.

1. Let `th` be theorem (4.6). Instantiate f and q with the automaton representation in question.
2. Normalize the characteristic function P within theorem `th`
3. Match the left hand side of (4.7) with the current theorem `th` and apply modus ponens. Store the result in theorem `th'`.
4. Normalize the characteristic function P within theorem `th'`
5. If the characteristic functions in `th` and `th'` are equal, then P the second assumption in theorem (4.8) is fulfilled. Match the first assumption with `th` and derive the conclusion

   $$\vdash \mathsf{reachable}\ (f, q) = P$$

   via modus ponens. The algorithm terminates returning the above theorem.

   If the characteristic functions in `th` and `th'` are unequal, then assign `th` to `th'` and continue with step 3.

The algorithm assumes that there exists some normalization procedure for step 2 and step 4.

Remark: The algorithm assumes, that there exists some normalization function for the characteristic functions for state sets. It always terminates for finite state σ and finite input ι.

4.3 Automata Equality Proof via State Traversation

According to theorem (2.3), the equivalence of two automata can be turned into an equivalent proof goal, stating that the corresponding product automaton always produces the T signal at the output. The following theorem states that, an automaton always produces a T signal at the output iff f produces the T signal as output for all reachable states and all inputs.

$$\begin{aligned} \vdash\ & (\forall \mathit{input}, t.\ \mathsf{automaton}\ (f, q)\ \mathit{input}\ t = \mathsf{T}) \\ & = \\ & (\forall s, i.\ \mathsf{reachable}\ (f, q)\ s \Rightarrow \mathsf{FST}(f(i, s))) \end{aligned} \tag{4.10}$$

This allows one to proof the equivalence of automata by first performing state traversation according to the previous section and then checking whether the set characterized by $((f, q)\ s)$ is a superset of $(\lambda s.\ \exists i.\ \mathsf{FST}(f(i, s)))$.

4.4 Modified State Traversation

The state traversation algorithm described in 4.2 is intended for determining the exact set of reachable states. It may also be useful to determine a superset of the set of reachable states. State minimization as described in section 5.1, for example, eliminates some unreachable states. It therefore requires a superset of the reachable states as the new state set of the optimized automaton. Proving the equality of two automata according to theorem 4.10 can as well be done by proving that $\mathsf{FST}(f(i, s))$ holds for all s of some superset of the reachable states.

To achieve some superset of the reachable states, one can modify the algorithm of section 4.2. Other than in the standard, one can also add arbitrary unreachable states to the state set. The only thing that need be guaranteed in the end is that one has achieved a set that contains q and adding the direct successors does not extend the set. These properties can rather easily be checked and, according to the following theorem, are an adequate criterion for ensuring that the current set is a superset of the reachable states.

$$
\begin{array}{lll}
\vdash & (& \\
 & \quad P(q) \wedge & \\
 & \quad (\forall i, s.\ \mathsf{add\_direct\_successors}\ (f, q)\ P\ s \Rightarrow P(s)) & \\
 &) & \\
 & = & \\
 & (\forall s.\ \mathsf{reachable}\ (f, q)\ s \Rightarrow P(s)\ ) &
\end{array}
\tag{4.11}
$$

Chapter 5

Synthesis Related Transformations

Equivalence of automata means that for a given input, the automata produce the same output. An automaton (f, q) can be trivially turned into an equivalent automaton by substituting f and q by equivalent terms $\tilde{f} = f$ and $\tilde{q} = q$. All automata achievable by such transformations have one thing in common: the states are represented in the same way. This section presents automata transformations which go beyond this: state encoding, retiming and the elimination of redundant parts.

5.1 State Encoding

This section introduces a set of four theorems for changing the state encoding of states. All four theorems have one thing in common: they change an automaton according to figure 5.2. An automaton given as (f, q) is modified by inserting two functions g and h such that state values are encoded via g before they are led to the memory unit and decoded via h on their way from the memory unit back to f. The initial value q is substituted by $g(q)$.

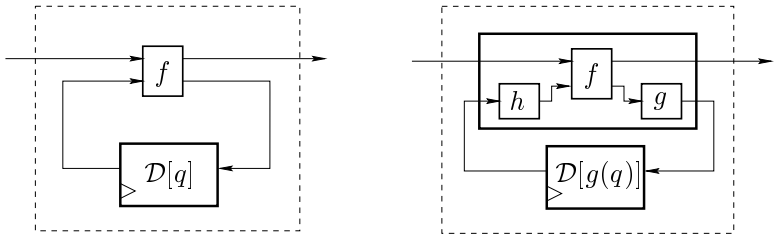


Figure 5.1: State Encoding

Performing state encoding changes the type of states. In the original automaton it is σ, and it becomes σ' after the encoding. The encoding function g is a mapping from σ to σ' and the decoding function h is a mapping in the inverse direction.

The following equation describes this relation in a formal manner. It has to be noted, that this equation is not a tautology. The four theorems to be presented have different assumptions under which this equation is fulfilled.

```
automaton (f, q)
=
automaton(
  (λ(i, s).
    let a = h(s) in
    let (b, c) = f(i, a) in
    let d = g(c) in
      (b, d)
  ) ,
  g(q)
)
```

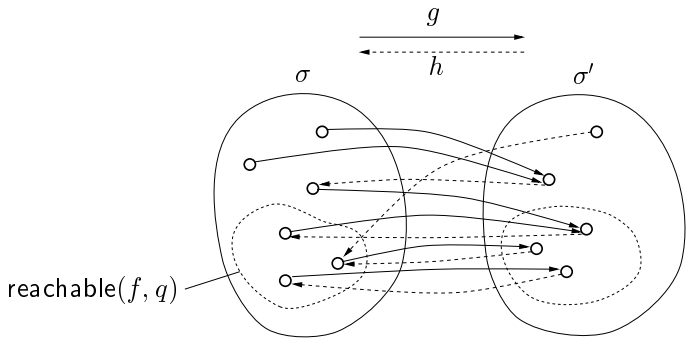


Figure 5.2: Encoding from σ to σ'

The first theorem performs the encoding of reachable states. The second and the third theorem are corollaries: they perform state encoding of all states and the pure elimination of unreachable states, respectively. The last theorem encodes classes of states with the same behavior by single states.

Encoding of Reachable States

The following theorem performs the encoding of reachable states. The left hand side of the implication in theorem (5.1) states, that there the encoding/decoding functions g and h are "inverse" for all reachable states. Here, inverse means $h(g(s)) = s$ but not necessarily $g(h(s)) = s$. Figure 5.1 illustrates the relation

between g and h.

$$
\begin{array}{ll}
\vdash & (\forall s.\ (\mathsf{reachable}\ (f,q)\ s)\ \Rightarrow\ h(g(s)) = s) \\
& \Rightarrow \\
& (\\
& \quad \mathsf{automaton}\ (f,q) \\
& \quad = \\
& \quad \mathsf{automaton}(\\
& \quad\quad (\lambda(i,s). \\
& \quad\quad\quad \mathsf{let}\ a\ =\ h(s)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ (b,c)\ =\ f(i,a)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ d\ =\ g(c)\ \mathsf{in} \\
& \quad\quad\quad\quad (b,d) \\
& \quad\quad)\ , \\
& \quad\quad g(q) \\
& \quad) \\
&)
\end{array}
\tag{5.1}
$$

Corollary A: Pure State Encoding

The previous theorem has two effects: it eliminates some unreachable states and it changes the encoding of the remaining states. Determining reachability can only be performed for small sized automata. The following is restricted to pure state encoding. In contrast to theorem (5.1), theorem (5.2) performs the state encoding for the entire set of states — reachability need not be considered.

$$
\begin{array}{ll}
\vdash & (\forall s.\ h(g(s)) = s) \\
& \Rightarrow \\
& (\\
& \quad \mathsf{automaton}\ (f,q) \\
& \quad = \\
& \quad \mathsf{automaton}(\\
& \quad\quad (\lambda(i,s). \\
& \quad\quad\quad \mathsf{let}\ a\ =\ h(s)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ (b,c)\ =\ f(i,a)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ d\ =\ g(c)\ \mathsf{in} \\
& \quad\quad\quad\quad (b,d) \\
& \quad\quad)\ , \\
& \quad\quad g(q) \\
& \quad) \\
&)
\end{array}
\tag{5.2}
$$

Before this corollary can be applied, an appropriate encoding in terms of g and h has to be found and it has to be proven, that the encoding is correct, i.e. $\forall s.\ h(g(s))$ holds (see figure 5.3). The quality of the synthesis result (size of combinatorial logic, size of memory, etc.) very much depends on the encoding chosen. Usually there are lots of different encodings, and there already exist

different techniques for determining good encodings according to different optimization criteria. For types with a huge cardinality, proving $\forall s.h(g(s)) = s$ may become exacting. Besides explicitly proving the correctness of a given encoding, it is also possible to derive a correct encoding in a systematic manner. Different ways for deriving encoding/decoding function pairs will be described in chapter 6.

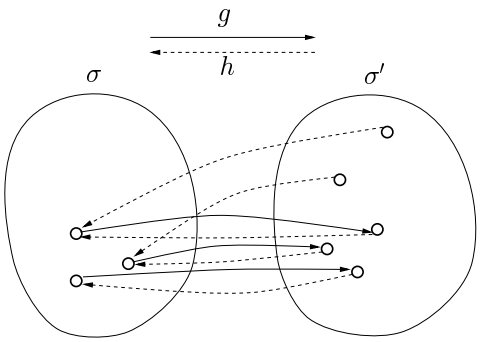


Figure 5.3: Encoding all states

Corollary B: State Minimization

Corollary B to theorem (5.1) is dedicated to reducing the state set by eliminating unreachable states. It is assumed, that one has divided σ into $\sigma^1 + \sigma^2$, where all the reachable states are in σ^1. There may also be some unreachable states in σ^1, but there is no reachable state in σ^2. In this situation, the state representation can be cut down to σ^1 using a pair of encoding/decoding functions g and h where h maps x to $\mathsf{INL}(x)$ and g maps every expression $\mathsf{INL}(x)$ back to x. It is to be noted here, that g is only partially specified. The result of applying g to

$\mathsf{INR}(x)$ is unspecified.

$$
\begin{array}{ll}
\vdash & (\\
& \quad (\forall s.\ (\mathsf{reachable}\ (f,q)\ s))\ \Rightarrow\ \mathsf{ISL}(s)\ ) \wedge \\
& \quad (\forall x.\ h(x) = \mathsf{INL}(x)\ ) \wedge \\
& \quad (\forall x.\ g(\mathsf{INL}\ x) = x\ ) \\
&) \\
& \Rightarrow \\
& (\\
& \quad \mathsf{automaton}\ (f,q) \\
& \quad = \\
& \quad \mathsf{automaton}(\\
& \quad\quad (\lambda(i,s). \\
& \quad\quad\quad \mathsf{let}\ a\ =\ h(s)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ (b,c)\ =\ f(i,a)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ d\ =\ g(c)\ \mathsf{in} \\
& \quad\quad\quad\quad (b,d) \\
& \quad\quad)\ , \\
& \quad\quad g(q) \\
& \quad) \\
&)
\end{array}
\tag{5.3}
$$

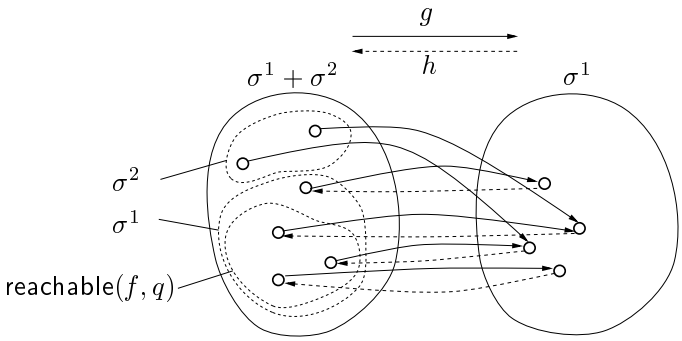


Figure 5.4: Elimination of Unreachable States

Annotation: Usually σ does not have the form $\sigma^1 + \sigma^2$ with all reachable states being on the left hand side. Conversions based on corollary A can be used to reach such a representation.

Classifying States

The previous state encoding theorems can be used for eliminating unreachable states and changing the encoding states in a bijective manner. The encoding function g has always been injective with respect to the set of reachable states. Other than these theorems, the theorem to be presented in this section may also reduce the number of states among the reachable states.

Theorem 5.4 reduces the number of states by mapping a set of "equivalent" states to a single state. Let there be some classification for the set of states such that the output and transition function f always produces equivalent results for states being in the same class. Equivalent means: the output is the same and the successor states are in the same class.

The theorem states, that there exists a function h that maps each value to the representative of its class and an "inverse" function g that maps each representative of some class to some value of the class (see figure 5.5). g is the characteristic function of the classification. During state encoding, g is used as the encoding function and h is used as the decoding function according to figure 5.1.

$$
\begin{array}{ll}
\vdash & (\,\forall s.\, g(h(s)) = s\,) \wedge \\
& (\,\forall s^1, s^2.\, g(s^1) = g(s^2) \\
& \quad \Rightarrow (\forall i. \\
& \quad\; \mathsf{FST}(f(i, s^1)) = \mathsf{FST}(f(i, s^2)) \wedge \\
& \quad\; g(\mathsf{SND}(f(i, s^1))) = g(\mathsf{SND}(f(i, s^2))) \\
& \quad)) \\
& \Rightarrow \\
& (\\
& \quad \mathsf{automaton}\ (f, q) \\
& \quad = \\
& \quad \mathsf{automaton}(\\
& \quad\quad (\lambda(i, s). \\
& \quad\quad\quad \mathsf{let}\ a\ =\ h(s)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ (b, c)\ =\ f(i, a)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ d\ =\ g(c)\ \mathsf{in} \\
& \quad\quad\quad\quad (b, d) \\
& \quad\quad)\,, \\
& \quad\quad g(q) \\
& \quad) \\
&)
\end{array}
\tag{5.4}
$$

5.2 Retiming

In simplified terms, retiming (more precisely: forward retiming) moves the memory part over some combinatorial part g. In order to guarantee correctness, the initial state q has to be transformed from q to $g(g)$ (see figure 5.6). Retiming can significantly change the delay of the combinatorial part of the circuit and therefore increase the clock frequence. 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.

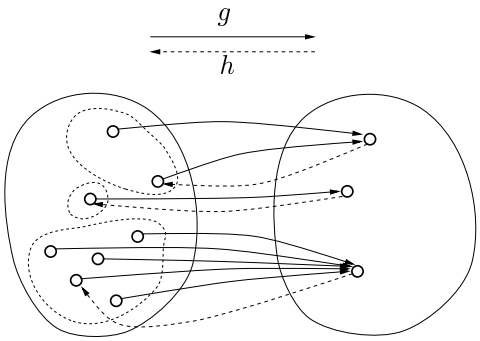


Figure 5.5: Classifying States

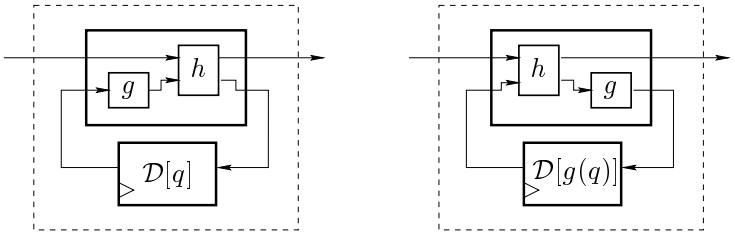


Figure 5.6: Retiming

The Retiming Theorem

The following theorem describes retiming in a very general manner. It states that the two automata descriptions in figure 5.6 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 5.5 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.

$$\begin{aligned} \vdash\ & \mathsf{automaton}(\ (\lambda(i,s).\, h(i,g(s)))\ ,\ q\ ) \\ & = \mathsf{automaton}(\ (\lambda(i,s).\, \mathsf{let}\ (x,y) = h(i,s)\ \mathsf{in}\ (x,g(s)))\ ,\ g(q)\ ) \end{aligned} \tag{5.5}$$

Applying the Retiming Theorem

Retiming can be performed in both directions. The synthesis step from left to right (figure 5.6) 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 5.7 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.

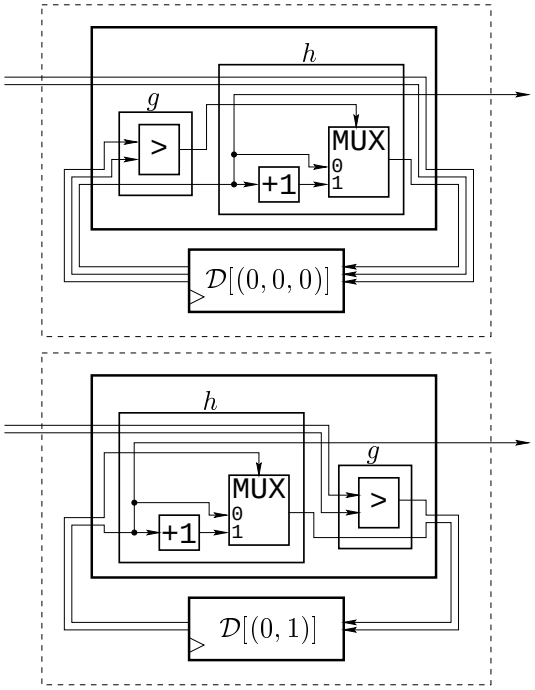


Figure 5.7: Example for Applying the Retiming Scheme

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 omitted 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 5.6. 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 5.8 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).

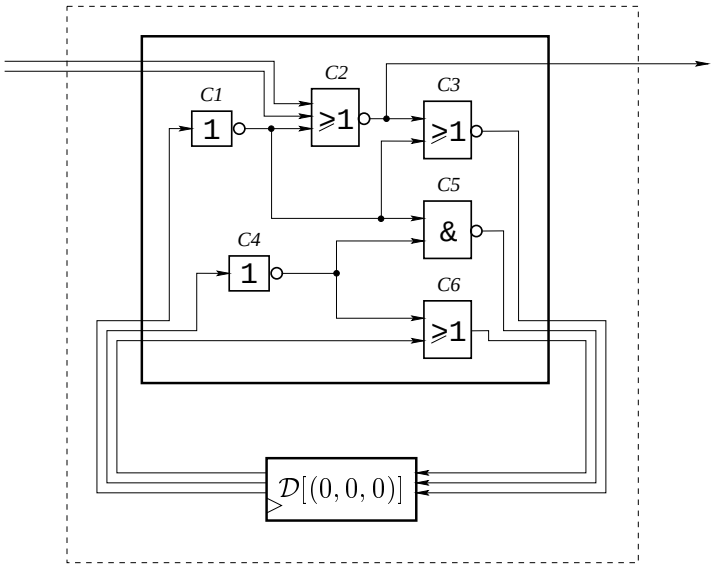


Figure 5.8: Example for Retiming

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 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.

5.3 Elimination of Redundant Parts

The following theorem can be applied to eliminate parts of the circuits, that do not affect the input/output behaviour. There are two preliminaries for eliminating a combinatorial block: it does not effect the output and the values it writes to the memory component are only read by itself.

It is assumed that the combinatorial part is divided in two parts g and h

(see figure 5.9) where h is the part that is eliminated during the transformation step.

$$
\begin{array}{ll}
\vdash & \mathsf{automaton}(\\
& \quad (\lambda(i,(s^1,s^2)).\ \mathsf{let}\ (x,y) = g(i,s^1)\ \mathsf{in}\ (x,(y,h(i,s^1,s^2)))) \ , \\
& \quad (q^1,q^2) \\
& \quad) \\
& = \mathsf{automaton}(\ g,\ q^1\ )
\end{array}
\tag{5.6}
$$

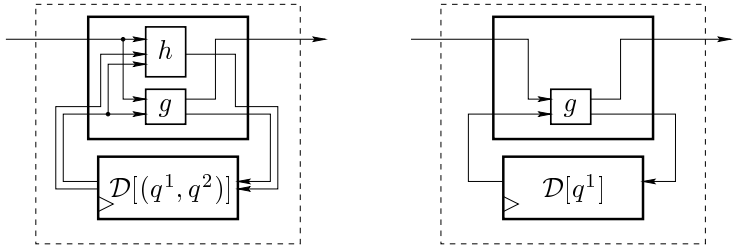


Figure 5.9: Elimination of Redundant Parts

Synthesis of synchronous VHDL descriptions may lead to such redundant parts [EiKu95c, EiKu96]. VHDL variables in general have to be implemented by memory units since in VHDL the result of variables is also accessible in the next clock tick. However, variables may also be used to only hold variable values within one clock tick. However, omitting the register is allowed only if the automaton derived from the VHDL process matches the pattern in figure 5.9.

Chapter 6

Systematic Derivation of State Encodings

The automata theory provides several pairs of encoding/decoding functions for a set of basic data types. This chapter is dedicated towards formally deriving encoding/decoding-pairs according to section 5.1. The theorems to be produced are to be used as assumptions in theorem (5.3).

The chapter is structured as follows: First, the data types that are to be considered will be introduced. The next section lists a set of theorems supporting encoding. The succeeding section will illustrate, how this set of encoding theorems can be applied to formally derive pairs of encoding/decoding-functions in a systematic manner.

6.1 A Set of HOL Data Types

The encoding theorems to be described will be restricted to a set of 6 data types (and type operators). Their semantics is described in a ML-fashion. Greek letters are used to indicate type variables. HOL provides a type definition mechanism based on such descriptions [Melh88, GoMe93, Melh93].

In simple words, the semantics is as follows: Each data type t has a set of constructors that is either a constant of type t or a function mapping some parameters of some given type to t. The data type holds all values that can be produced by its constructors. Values produced by different constructors are unequal.

one	=	one
bool	=	T | F
num	=	0 | SUC of num
(α)option	=	none | any of α
$\alpha \times \beta$	=	, of $\alpha \Rightarrow \beta$
$\alpha + \beta$	=	INL of α | INR of β

one, bool and num are simple data types whereas option, $\times$ and $+$ are type constructors, i.e. mappings from some type to another. The unary type constructor option, for example, maps some arbitrary type α to (α)option. The type constructors $\times$ and $+$ are binary type constructors mapping arbitrary data types α and β to $\alpha \times \beta$ and $\alpha + \beta$, respectively.

The data type one has one single constructor which is also named one. The constructor one is a constant of type one. Therefore the set represented by the data type one is {one}. The data type bool represents the boolean values T and F.

The data type num represents the set of nonnegative natural numbers. It is defined in a recursive manner. There is a constructor named 0, which is a constant of type num and there is a constructor function SUC mapping some value of type num to its successors value.

option is a unary type operator that is used in postfix notation. In simplified terms, option adds one element to some data type. It is defined via the two constructors none and any. Let α be some arbitrary type. (α)option represents the set consisting of the term none plus all the elements any(x), where x is of type α. In mathematical notation, (α)option represents the following set:

$$\{\mathsf{none}\} \cup \{\mathsf{some}(x) \mid x \in \alpha\}$$

$\times$ represents the scalar product. It is a binary type operator that is used in infix notation. The ,-operator maps two expressions of type α and β to an expression of type $\alpha \times \beta$. The expression x, y denotes a logical term of type $\alpha \times \beta$. In mathematical notation, $\alpha \times \beta$ represents the following set:

$$\{(x, y) \mid x \in \alpha \wedge x \in \beta\}$$

$+$ represents the "sum" of two types. $+$ is a binary type operator that is used in infix notation. There are two constructors INL and INR. The constructor INL maps an expressions of type α to an expression of type $\alpha + \beta$, and the constructor INL maps an expressions of type β to an expression of type $\alpha + \beta$. In mathematical notation, $\alpha + \beta$ represents the following set:

$$\{\mathsf{INL}(x) \mid x \in \alpha\} \cup \{\mathsf{INR}(x) \mid x \in \alpha\}$$

6.2 Encoding Theorems

The automata theory provides some theorems with pairs of correct encoding/decoding functions for the data types mentioned above. They support conversions from RT level data type descriptions down to gate level data types. We will explain, which are the types these conversions come from and go to, rather then, explain them in detail.

We will use $\alpha \rightharpoonup \beta$ to indicate, that there is some encoding from type α to type β and we will use $\alpha \rightleftharpoons \beta$ to indicate, that there are bijective encodings, i.e. encodings from α to β and vice versa. Table 6.1 lists some useful encoding theorems and the corresponding data types.

Theorem Names	Encoding/Decoding		
NUM_BOOL*	num	$\rightleftharpoons$	bool
NUM_PROD	num	$\rightleftharpoons$	num $\times$ bool
OPTION_SUM	(α)option	$\rightleftharpoons$	one $+ \alpha$
OPTION_TRANS†	(α)option	$\rightleftharpoons$	(α')option
OPTION_EXTEND	α	$\rightharpoonup$	(α)option
SUM_ASSOC	$(\alpha + \beta) + \gamma$	$\rightleftharpoons$	$\alpha + (\beta + \gamma)$
SUM_COM	$\alpha + \beta$	$\rightleftharpoons$	$\beta + \alpha$
SUM_TRANS†	$\alpha + \beta$	$\rightleftharpoons$	$\alpha' + \beta'$
SUM_EXTEND	α	$\rightharpoonup$	$\alpha + \beta$
SUM_PROD	$\alpha + \alpha$	$\rightleftharpoons$	bool $\times \alpha$
PROD_ASSOC	$(\alpha \times \beta) \times \gamma$	$\rightleftharpoons$	$\alpha \times (\beta \times \gamma)$
PROD_COM	$\alpha \times \beta$	$\rightleftharpoons$	$\beta \times \alpha$
PROD_NEUTRAL	$\alpha \times$ one	$\rightleftharpoons$	α
PROD_TRANS†	$\alpha \times \beta$	$\rightleftharpoons$	$\alpha' \times \beta'$
PROD_EXTEND	α	$\rightharpoonup$	$\alpha \times \beta$
BOOL_NEG	bool	$\rightleftharpoons$	bool

Table 6.1: Encodings For Simple Data Types

The theorems NUM_BOOL and NUM_PROD can be used to convert natural numbers with a limited range to tuples of booleans. NUM_PROD is used to split a boolean from a natural number and to halve the size of the number, and NUM_BOOL is used for encoding nonnegative numbers that are less than 2.

Theorem OPTION_SUM states, that (α)option can be encoded by means of $+$ and one. Theorem BOOL_NEG states, that there is an encoding from booleans to booleans (turning T to F and vice versa).

option, $+$ and $\times$ are type operators. The theorems OPTION_TRANS, SUM_TRANS and PROD_TRANS derive encodings for these type operators. Provided that there are encodings for the subtypes $\alpha \rightleftharpoons \alpha'$ and $\beta \rightleftharpoons \beta'$, then the encoding for the entire type expressions (α)option $\rightleftharpoons (\alpha')$option, $\alpha + \beta \rightleftharpoons \alpha' + \beta'$ and $\alpha \times \beta \rightleftharpoons \alpha' \times \beta'$ are derived.

The binary type operators $+$ and $\times$ are commutative and associative in the sense that there are bijective encodings between such type expressions (see theorems SUM_ASSOC, SUM_COM, PROD_ASSOC and PROD_COM).

All the encodings described until now, are bijective encodings. The theorems OPTION_EXTEND, SUM_EXTEND and PROD_EXTEND, however, describe encodings that are applicable only in one direction. They all lead to "bigger"

*$\rightharpoonup$ only for natural numbers < 2

†under the assumption that $\alpha \rightleftharpoons \alpha'$ and $\beta \rightleftharpoons \beta'$

types in the sense that the new type contains some extra elements.

6.3 Algorithms for Deriving Correct Encodings

This section gives a gist of how encoding algorithms may look like. The algorithms are motivated by our recent work about the embedding a synchronous subset of VHDL in HOL [EiKu95c, EiKM96, EiKu96]. See also [KlBr95] for various other approaches for embedding VHDL. Our approach is based on the automata theory. For a given behavioural description in VHDL, the corresponding automata description in terms of its initial state q and the output and transition function f are extracted. In such automata the state $\sigma = \sigma^c \times \sigma^d$ consists of two parts: control state σ^c and data state σ^d. This section addresses the encoding of the control state part using the encodings given in the previous section.

The set of controller states is finite. To represent them, type expressions built with one, option and + are used. To derive a representation on the gate level, these types have to be mapped to tuples of booleans, i.e. data types constructed via bool and $\times$.Each control state represents either the starting point or one of the `wait`-statement positions in the VHDL program. It is not intended to go into the detail of how these type expressions have resulted. Here is just a brief hint on their meaning:

- one is used to represent single wait statement positions,
- $\alpha + \beta$ is used to represent the control states of a compound statement consisting of two parts (sequence, if-then-else) where α represents the set of wait-statement positions in the first part and β is used to represent the wait-statement positions of the second part.
- (α)option is used for expressing positions before or after (compound) statements. While any(s) is used to represent wait-statement positions within a statement, none is used to indicate either the position before the statement or (in another context) the position immediately after the statement.

Derivation of a Minimal Bit Encoding

There usually is a broad range of correct encodings. The algorithm to be presented in this section produces an encoding with a minimal number of bits. The algorithm is illustrated by the following example:

$$(\mathsf{one} + (\mathsf{one})\mathsf{option})\mathsf{option} + (\mathsf{one} + \mathsf{one})$$

In the first step all occurrences of (α)option are replaced by one $+ \alpha$. Theorem OPTION_SUM is used to perform this encoding step. The type reached after the encoding:

$$(\mathsf{one} + (\mathsf{one} + (\mathsf{one} + \mathsf{one}))) + (\mathsf{one} + \mathsf{one})$$

Now the type expression consists of the type constant one and the binary type operator + only. The cardinality of a set represented by such a type expression equals the number of one occurrences. Such type expressions can be seen as binary trees, whose depth corresponds to the number of bits needed for encoding.

In this step, the depth of the tree is reduced by applying SUM_ASSOC. The algorithm balances the tree in a bottom up fashion. Let $\alpha + \beta$ be some node where the cardinalities of α and β are $|\alpha|$ and $|\beta|$, respectively. If $|\alpha| > 2 * |\beta|$ holds, then SUM_ASSOC is applied and if $|\beta| > 2 * |\alpha|$ holds, then SUM_ASSOC is applied in the inverse direction.

In our example, there is only one position, where the tree has to be balanced: the subexpression (one + (one + (one + one))). Here the cardinality of the left hand side is 1 and the cardinality of the right hand side is 3. So SUM_ASSOC is applied in the inverse direction. We obtain:

((one + one) + (one + one)) + (one + one)

Until now, the cardinality of the entire type has been left unchanged. In order to reach a symmetric tree and to be able to encode the type by scalar products of booleans, we will now add some redundant states. Theorem SUM_EXTEND is applied to encode one by one + one whenever one is a leaf with a depth less than the maximum depth of the tree.

In our example, there were 6 states. After the extension, there are 8. In the automaton the two extra states which have been added during the extension are unreachable.

((one + one) + (one + one)) + ((one + one) + (one + one))

Now the type expression tree is symmetric, i.e. in every node the left hand side equals the right hand side. Theorem SUM_PROD is now applied repeatedly applied in a top down fashion.

bool × (bool × (bool × one))

Finally SUM_NEUTRAL is applied to encode bool × one by bool.

bool × (bool × bool)

Derivation of a One Hot Encoding

The algorithm to be described in this section derives boolean encodings that use one bit per state. Each bit corresponds to one state. When the automaton is in some state, the corresponding bit becomes T and all other bits become F. The algorithm will be described with the same example as for the previous algorithm.

(one + (one)option)option + (one + one)

As in the previous example, the option type operator is eliminated using OPTION_SUM:

(one + (one + (one + one))) + (one + one)

Applying SUM_ASSOC repeatedly leads to:

$$\mathsf{one} + (\mathsf{one} + (\mathsf{one} + (\mathsf{one} + (\mathsf{one} + \mathsf{one}))))$$

Combining the encodings SUM_TRANS (in forward direction), ONE_EXTEND and SUM_PROD leads to the following compound encoding:

$$\alpha + \mathsf{one} \rightharpoonup \mathsf{bool} \times \alpha$$

Applying this compound encoding repeatedly produces:

$$\mathsf{bool} \times (\mathsf{bool} \times (\mathsf{bool} \times (\mathsf{bool} \times (\mathsf{bool} \times \mathsf{bool}))))$$

Chapter 7

Structures consisting of Automata

Until now, circuits have always been described by single automata. Automata are an appropriate means for describing arbitrary circuits in a functional style. This chapter is dedicated towards describing circuits by structures of where the components are automata. Structures of automata will formally be described in a relational style.

Describing circuits by complex single automata as well as using structures each consisting of several small automata has its pros and cons. The automata theory provides means for switching between the two representations styles.

7.1 The Relational Circuit Description Style

Up to now, circuits where formalized as mapping from time dependent input signals to time dependent output signals. In this section, circuits are to be described by means of relations between time dependent signals. Relations are more general than functions: every function is a relation, but it is not the other way round.

In the relational approach, input and output signals need not necessarily be distinguished. An expression of the form

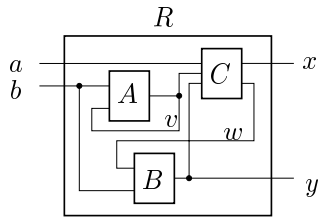
$$P(a, b, c)$$

states, that there are some signals a, b and c fulfilling some relation P corresponding to some circuit. Representing structures in higher order logic is straightforward [HaDa86, Melh93]. The general scheme is as follows:

$$\begin{array}{l} \forall i^1, i^2, \ldots i^{n_i}, o^1, o^2, \ldots o^{n_o}. \\ \quad \exists y^1, x^2, \ldots y^m. \\ \qquad R(x^1, x^2, \ldots x^n) = R^1(\ldots) \wedge R^2(\ldots) \wedge \ldots \wedge R^k(\ldots) \end{array}$$

In this formula, a compound circuit R is defined as a composition of its parts $R^1, R^2, \ldots R^k$. The external signals $i^1, i^2, \ldots i^{n_i}, o^1, o^2, \ldots o^{n_o}$ are all-quantified and the internal signals $y^1, y^2, \ldots y^m$ are existentially quantified. The interface of the compound circuit is connected with all external signals, the interfaces of its components may be connected to arbitrary internal or external signals according to the given net list. In such net list descriptions, circuits are represented by relations. Input and output signals are not distinguished, and there may be several input and output signals.

Figure 7.1 gives an example for a formalization of a circuit structure according to this scheme. The compound circuit is named R, its parts are named A, B and C. The components A, B, C and R are arbitrary synchronous circuits. The inputs of R are a and b, its outputs are x and y and there are two internal lines named v and w.



$$\forall a, b, x, y.$$
$$\exists v, w.$$
$$R(a, b, x, y) = A(a, v, v) \wedge B(w, b, y) \wedge C(a, v, x, w, y)$$

Figure 7.1: Formalization of a Structure

7.2 Relational versus Functional Circuit Descriptions

The major advantage of the relational circuit description style is, that circuits may be composed easily. Relations are powerful as to expressiveness. They also allow partial specifications that do not represent single circuits but characterize sets of real circuits. However, such a set may also be empty. Such contradictory circuit descriptions are crucial. On the one hand side, there is no real circuit they stand for, and this means they are unsynthesizable. For large sized circuit descriptions detecting contradictions may become difficult. On the other hand side, contradictory circuits fulfill any property (ex falso quodlibet) which may lead to false hopes after having proven specific properties for contradictory circuits.

When building sequential circuit structures, two restrictions have to be considered: shortcircuits and zero delay cycles. A shortcircuit is an interconnection of two or more output signals. In the technical practice this leads to an extreme power consumption in case that the signal values differ. The resulting signal

value is uncertain, and it may even come to a destruction of the circuit due to the heat.

In the sequential timing abstraction, combinatorial components produce the output signals without delay whereas memory units delay signals by one time unit. Zero delay cycles are rings of combinatorial components where each component produces a signal to an input of its successors. To detect such cycles in hierarchical structures, one has to analyze zero delay dependencies between outputs and inputs bottom up. In figure 7.1, for example, the signal v builds a loop around A. Whether or not this is a zero delay cycle depends on the delay on the delay between the second input and the output of A. In figure 7.1, there may also be a second zero delay cycle with the components B and C and the signals w and y. It depends on the delay between the first input and the output of B and on the delay between the second input and the second output of C.

Both shortcircuits and zero delay cycles may lead to contradictory circuit descriptions. In the functional circuit description approach shortcircuits and zero delay cycles cannot be described, but with relational circuit structures one may. One way to avoid such bugs is explicitely check the absence of shortcircuits and zero-delay cycles [AHL92].

Automata descriptions also have the advantage of being simulatable. For a given automaton description represented via initial state and output- and transition function, there is a simple algorithm for mapping the input signal to the output signal (see section **??**). Relations on the other hand are less constructive. They only state whether or not some signals correspond to some simulation run of a circuit. However, they do not give a hint on how to perform simulation, and it may even be that this computation is not computable at all.

7.3 Representing Structures that Consist of Automata

An automaton can easily be used to describe a relation between an input and an output

$$output = \mathsf{automaton}(f, q)\ input$$

However, both the input and the output may be bundles of signals rather than single signals. To allow arbitrary interconnections between single signals, one has to split the signal. The expression

$$(\lambda t.\ (output^1(t), \ldots, outputs^{n_o}(t))) = \\ \mathsf{automaton}(f, q)\ (\lambda t.\ (input^1(t), \ldots, inputs^{n_i}(t)))$$

is similar to the previous one except that input and output are represented by functions mapping time t to a tuple, where each component corresponds to the value of one signal within the bundle. Using this pattern, the above mentioned scheme for representing structures in a relational manner can be used to also represent structures consistion of several automata.

7.4 Splitting an Automaton into Combinatorial Block and Memory Block

The following theorem describes a relation between automata, combinatorial blocks and memory blocks. It states that an automaton is equivalent to a structure consisting of a combinatorial block and a memory block.

Applying this theorem can be considered to be a first step towards switching from an automaton description towards a structural description using the relational description style. The next step could be splitting the memory block and the combinatorial block into smaller memory parts and combinatorial units, respectively.

$$
\begin{array}{ll}
\vdash & (output = \mathsf{automaton}\ (f, q)\ input) \\
& = \\
& (\exists x, y. \\
& \quad (\lambda t.\ (output(t), x(t))) = \\
& \quad\quad \mathsf{combinatorial\_block}\ f\ (\lambda t.\ (input(t), y(t)))\ ) \\
& \quad \wedge \\
& \quad (y = \mathsf{memory\_block}\ q\ x) \\
&)
\end{array}
\tag{7.1}
$$

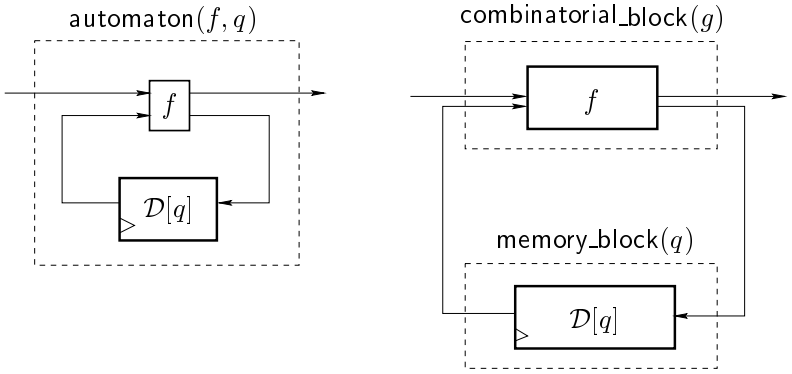


Figure 7.2: Splitting an Automaton into Combinatorial Block and Memory Block

7.5 Switching from Relational to Functional Circuit Descriptions

The next two theorems provide a mechanism for switching from relational circuit descriptions to automata representations. As described in the previous sections,

there are several reasons why this makes sense. The most important reasons are, that they are unambiguous and simulatable.

Not all relational circuit descriptions are free from contradictions due to zero delay cycles and shortcircuits. Being free from zero delay cycles, however, is a preliminary for transforming them them to automata representations. Therefore having performed this transformations implicitly proves the absence of such bugs.

The theorems (7.2) and (7.3) provide means for switching between relational circuit descriptions and single automaton representations. The transformation is achieved in two steps. In the first step, theorem (7.2) is applied as often as possible. Theorem (7.2) combines two parallel circuits. All circuits of some structure can be considered to be switched in parallel although this may lead to circuits with connections from its outputs towards its inputs. The result of the first step is a structure only consisting of a single component. However, this still is a structural description since there still may be some connections from the outputs of this component towards its inputs.

$$
\begin{array}{ll}
\vdash & (\exists x. \\
& \quad (\lambda t.\,(output(t), x(t))) = \\
& \qquad \mathsf{automaton}(\\
& \qquad\quad (\lambda((i^1, i^2), s).\,(g(i^1, i^2, s),\ h(i^1, s),\ j(i^1, i^2, s)))\ , \\
& \qquad\quad q \\
& \qquad) \\
& \qquad\quad (\lambda t.\,(input(t), x(t)))\quad) \\
& = \\
& (output = \\
& \quad \mathsf{automaton}(\ (\lambda(i, s).\ \mathsf{let}\ z = h(i, s)\ \mathsf{in}\ (g(i, z, s),\ j(i, z, s)))\ ,\ q\ ) \\
& \qquad input\quad)
\end{array}
\tag{7.2}
$$

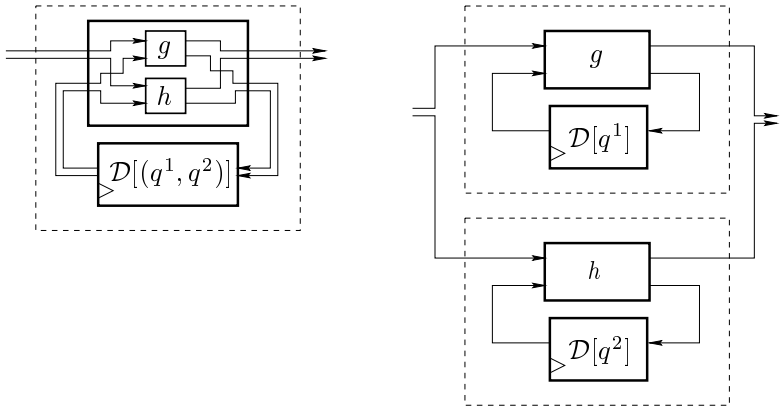


Figure 7.3: Parallel Connection of Two Automata

In the second step, cycles are eliminated by applying theorem (7.2) (see figure 7.3). Other than the first step, the second step may fail. The second step fails for zero-delay cycles. Figure 7.3 illustrates the theorem. There is a feedback signal that is produced by some component h. To apply the theorem, it must be provided, that h itself does not depend on the signal, i.e. the feedback signal must not be connected with an input of h.

The result of step two should be a single automaton component without any connections between its interface ports. Besides connections from outputs to inputs, some relational circuit descriptions also lead to interconnections between outputs due to shortcircuits. Such interconnections cannot be eliminated, the overall transformation fails.

$$
\begin{aligned}
\vdash \quad & (\exists x. \\
& \quad (\lambda t.\,(output(t), x(t))) = \\
& \quad\quad \mathsf{automaton}(\\
& \quad\quad\quad (\lambda((i^1, i^2), s).\,(g(i^1, i^2, s),\ h(i^1, s),\ j(i^1, i^2, s)))\ , \\
& \quad\quad\quad q \\
& \quad\quad) \\
& \quad\quad\quad (\lambda t.\,(input(t), x(t)))\quad) \\
& = \\
& (output = \\
& \quad \mathsf{automaton}(\ (\lambda(i, s).\,\mathsf{let}\ z = h(i, s)\ \mathsf{in}\ (g(i, z, s),\ j(i, z, s)))\ ,\ q\ ) \\
& \quad\quad input\quad)
\end{aligned}
\tag{7.3}
$$

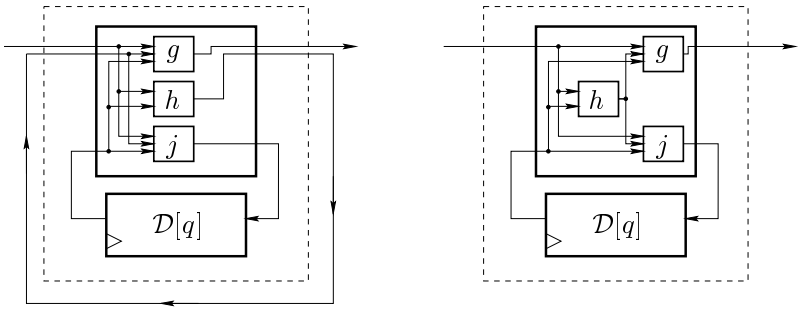


Figure 7.4: Cycle

Theorem (7.4) offers an alternative to theorem (7.2). As with theorem (7.2), it describes the combination of two components. But unlike theorem (7.2), there is also a connection from the first to the second component. Applying theorem (7.2) would lead to a loop, which one would have to eliminate via theorem (7.3). The application of theorem (7.4) allows performing this in single step rather than in two steps. It has to be noted, that theorem (7.4) is only applicable if the two

components are interconnected in exactly one direction.

$$
\begin{array}{ll}
\vdash & (\exists x. \\
& \quad (\ (\lambda t.\ (output^1(t), x(t))) = \mathsf{automaton}\ (f^1, q^1)\ input\ ) \wedge \\
& \quad (\ output^2 = \mathsf{automaton}\ (f^2, q^2)\ (\lambda t.\ (input(t), x(t)))\ ) \\
&) \\
& = \\
& (\\
& \quad (\lambda t.\ (output^1(t), output^2(t))) = \\
& \quad\ \mathsf{automaton}(\\
& \quad\quad (\lambda (i, (s^1, s^2)). \\
& \quad\quad\quad \mathsf{let}\ ((a, b), c) = f^1(i, s^1)\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ (d, e) = f^2((i, b), s^2)\ \mathsf{in} \\
& \quad\quad\quad\quad ((a, d), (c, e)) \\
& \quad\quad)\ , \\
& \quad\quad q \\
& \quad\quad) \\
& \quad\quad\quad input \\
& \quad) \\
&)
\end{array}
\tag{7.4}
$$

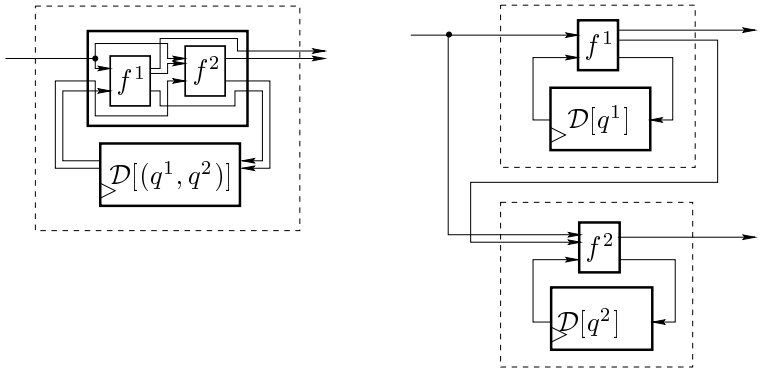


Figure 7.5: Concatenation of Two Automata

7.6 Subautomata

The next three theorems are designed for systematically extracting subautomata, i.e. parts of automata that are themselves automata. The idea behind this is to allow circuit transformations also on parts of automata, rather than on the entire automaton. Applying circuit transformation on a subautomaton is performed in three steps. First the subautomaton is extracted by applying the

following theorems in forward direction. This leads to a structure, where one of the components is the subautomaton in question. Then the circuit transformation is applied to this subautomaton, and finally the theorems, that were applied to extract the subautomaton are applied in inverse direction to reconstruct a single automaton.

Theorem (7.5) assumes, that the memory consists of two parts (scalar product) and splits apart one the two. The resulting structure consists of a subautomaton and a memory component (see figure 7.6).

$$
\begin{array}{ll}
\vdash & (output = \mathsf{automaton}\ (f, (q^1, q^2))\ input) \\
& = \\
& (\ \exists x, y. \\
& \quad (\ (\lambda t.\ (output(t), y(t))) = \\
& \quad\quad \mathsf{automaton}\ (\\
& \quad\quad\quad \lambda((a,b),c).\ \mathsf{let}\ (r,(s,t)) = f(a,(b,c))\ \mathsf{in}\ ((r,s),t)), \\
& \quad\quad\quad q^2 \\
& \quad\quad) \\
& \quad\quad\quad (\lambda t.\ (input(t), x(t))) \\
& \quad) \\
& \quad \wedge \\
& \quad (\ x = \mathsf{memory\_block}\ q^1\ y\ ) \\
&)
\end{array}
\tag{7.5}
$$

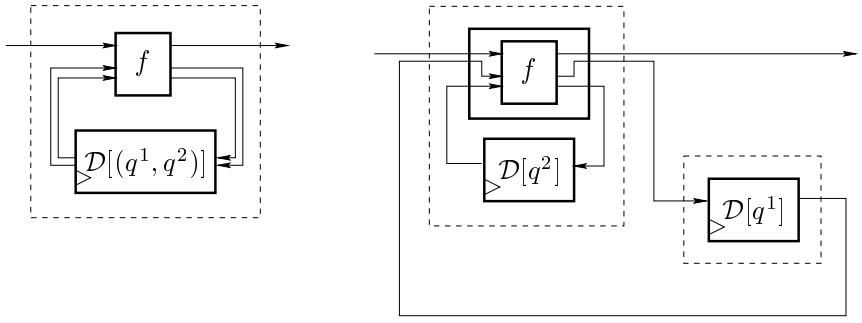


Figure 7.6: Splitting off a Memory Block

Theorem (7.6) and (7.7) split off combinatorial components. Theorem (7.6) splits off two combinatorial parts that are directly connected to the automaton's inputs and outputs, respectively (figure 7.7). Theorem (7.7) splits off an inner

combinatorial part.

$$
\begin{array}{ll}
\vdash & (\\
& \quad output = \\
& \quad\quad \mathsf{automaton}\ (\\
& \quad\quad\quad \lambda(i,s). \\
& \quad\quad\quad\quad \mathsf{let}\ (a,c)\ = f(i,s)\ \mathsf{in} \\
& \quad\quad\quad\quad \mathsf{let}\ b\ = g(a)\ \mathsf{in} \\
& \quad\quad\quad\quad \mathsf{let}\ (d,e)\ = h(b,c)\ \mathsf{in} \\
& \quad\quad\quad\quad\quad (d,e) \\
& \quad\quad\quad , \\
& \quad\quad\quad q \\
& \quad\quad) \\
& \quad\quad\quad input \\
&) \\
& = \\
& (\, \exists x, y. \\
& \quad (\, (\lambda t.\, (output(t), y(t))) = \\
& \quad\quad \mathsf{automaton}\ (\\
& \quad\quad\quad \lambda((i,b),s). \\
& \quad\quad\quad\quad \mathsf{let}\ (a,c)\ = f(i,s)\ \mathsf{in} \\
& \quad\quad\quad\quad \mathsf{let}\ (d,e)\ = h(b,c)\ \mathsf{in} \\
& \quad\quad\quad\quad\quad ((d,a),e) \\
& \quad\quad\quad , \\
& \quad\quad\quad q \\
& \quad\quad) \\
& \quad\quad\quad (\lambda t.\, (input(t), \mathrm{x}(t))) \\
& \quad) \\
& \quad \wedge \\
& \quad (\, x = \mathsf{combinatorial\_block}\ g\ y\,) \\
&)
\end{array}
\tag{7.6}
$$

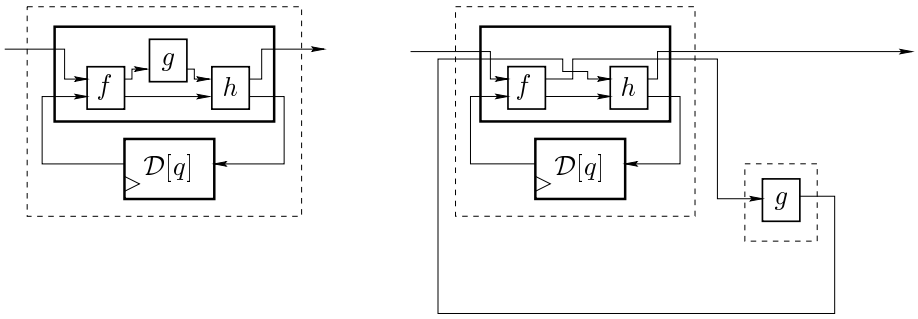


Figure 7.7: Splitting off a Combinatorial Block / I

$$
\begin{array}{ll}
\vdash & (\qquad (7.7)\\
& \quad output = \\
& \quad\quad \mathsf{automaton}\ (\\
& \quad\quad\quad \lambda(i,s). \\
& \quad\quad\quad\quad \mathsf{let}\ (a,b)\ = f(i)\ \mathsf{in} \\
& \quad\quad\quad\quad \mathsf{let}\ (c,d)\ = g(b,s)\ \mathsf{in} \\
& \quad\quad\quad\quad \mathsf{let}\ e\ = h(a,c)\ \mathsf{in} \\
& \quad\quad\quad\quad\quad (e,d) \\
& \quad\quad\quad , \\
& \quad\quad\quad q \\
& \quad\quad) \\
& \quad\quad\quad input \\
&) \\
& = \\
& (\, \exists x, y. \\
& \quad (\, (\lambda t.\, (output(t), y(t))) = \\
& \quad\quad \mathsf{combinatorial\_block}\ (\\
& \quad\quad\quad \lambda(i,c). \\
& \quad\quad\quad\quad \mathsf{let}\ (a,b)\ = f(i,s)\ \mathsf{in} \\
& \quad\quad\quad\quad \mathsf{let}\ e\ = h(a,c)\ \mathsf{in} \\
& \quad\quad\quad\quad\quad (e,b) \\
& \quad\quad) \\
& \quad\quad\quad (\lambda t.\, (input(t), x(t))) \\
& \quad) \\
& \quad \wedge \\
& \quad (\, x = \mathsf{automaton}\ (g,q)\ y\,) \\
&)
\end{array}
$$

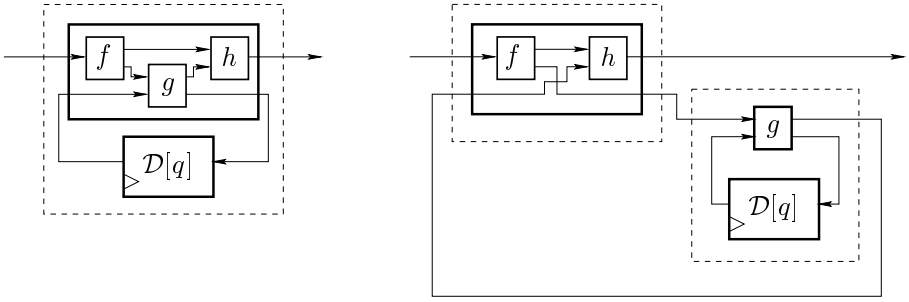


Figure 7.8: Splitting off a Combinatorial Block / II

Bibliography

[AHL92] AHL. *Lambda Reference Manual*, 1989.

[Day92] Nancy Day. A comparison between statecharts and state transition assertions. In [HOL92], pages 247–262.

[EiKM96] D. Eisenbiegler, R. Kumar, and J. Müller. A formal model for a VHDL subset of synchronous circuits. In *APCHDL '96*, Bangalore, India, 1996.

[EiKu95c] D. Eisenbiegler and R. Kumar. Formalizing the semantics for a synchronous subset of VHDL. Technical Report FZI-Report 8/95, Forschungszentrum Informatik (FZI), 1995.

[EiKu96] D. Eisenbiegler and R. Kumar. Synthese von verhaltensbeschreibungen in VHDL mittels logischer transformationen. In *Workshop Methoden des Entwurfs und der Verifikation digitaler Schaltungen*, Kreischa, Germany, March 1996. GI/ITG/GME.

[EiSK93] D. Eisenbiegler, K. Schneider, and R. Kumar. A functional approach for formalizing regular hardware structures. In [HOL93], pages 101–114.

[GoMe93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[HaDa86] F.K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEE Proc. Pt. E*, 133(3):242–254, 1986.

[HOL92] L. J. M. Claesen and M. J. C. Gordon, editors. *Higher Order Logic Theorem Proving and its Applications*, volume A-20, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland.

[HOL93] Jeffrey J. Joyce and Carl-Johan H. Seger, editors. *Higher Order Logic Theorem Proving and its Applications*, number 780, Vancouver, B.C., Canada, August 1993. Springer-Verlag.

[KlBr95] C.D. Kloos and P.T. Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *The Kluwer international series in engineering and computer science*. Kluwer, Madrid, Spain, March 1995.

[Loew92] Paul Loewenstein. A formal theory of simulations between infinite automata. In [HOL92], pages 227–246.

[Melh88] F. Melham. Automating recursive type definitions in higher order logic. Technical Report 140, University of Cambridge Computer Laboratory, 1988.

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

[ScKK93] K. Schneider, R. Kumar, and Thomas Kropf. Alternative proof procedures for finite-state machines in higher-order logic. In [HOL93], pages 213–226.

Index