CORE
Metadata, citation and similar papers at core.ac.uk
Provided by KITopen

An Automata Theory Dedicated towards Formal Circuit Synthesis

Dirk Eisenbiegler and Ramayya Kumar

Forschungszentrum Informatik
(Prof. Dr.-Ing. D. Schmid)
Haid–und–Neu–Straße 10-14, 76131 Karlsruhe, Germany
e–mail: {eisen,kumar}@fzi.de

Abstract We present a theory for automata in HOL, which is dedicated towards formal hardware synthesis. The theory contains definitions for formally representing and transforming automata. In this approach hardware is represented by automata descriptions and formal synthesis is performed by applying formally proven theorems. The approach presented is constructive — i.e. starting from specifications at higher levels of abstractions, synthesis can be performed by repeated applications of these transformations. Specialized refinements and optimizations at the RT and gate levels are discussed.

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 and the originally given partitioning 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/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 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. We do not intend to invent new synthesis algorithms but implement conventional ones in a formal manner.

The current state of the art about embedding automata in HOL is as follows: In [ScKK93] a specific set of formulae named hardware formulae is used for describing specifications and implementations of automata and appropriate proof procedures are defined. Although such descriptions are very useful for post-synthesis verification, they do not allow a constructive approach for performing formal synthesis. Similar to the approach taken in this paper, [Loew92, Day92] describe automata explicitly by means of expressions. This allows definitions and derivations of general theorems about automata. However, they allow more complex specifications such as non-deterministic automata and do not give constructive transformations which could lead to circuit implementations. In our work we consider only deterministic automata whose formalization is purely functional in nature and give transformations which can be used to perform refinements and optimizations, especially at the RT and gate levels. The overall theory can be regarded as a simple toolbox for formal synthesis algorithms at the RT and gate levels.

The outline of this paper is as follows: starting from the functional input/output definitions of the automata, we go on to describe the property of reachability. In section 5, we define the transformations which correspond to simple synthesis steps: state encoding, removal of unreachable states and the elimination of redundant memory parts. In section 6, we provide some encoding theorems for a small set of data types which is followed by an example in section 7.

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 our approach, we use the concept of typed functions, available in HOL, for representing automata. Given that ι, ω and σ are the types corresponding to the inputs, outputs and states, respectively, the output and the transition function have been combined to a single function f. It is to be noted here, that the types ι, o and σ can be compound — such as, tuples of basic data types.

The entire automata is represented by a pair (f, q), where f has the type $\iota \times \sigma \to \omega \times \sigma$ and q represents the initial state and has the type σ. The various manipulations that can be performed using such a representation is the chief concern of this paper.

f and q unambiguously determine, how the automaton maps a time dependent input signal $i_{\mathsf{num}\to\iota}$ to a time dependent output signal $o_{\mathsf{num}\to\omega}$. The constant automaton maps a pair (f, q) to a function mapping i to o. the constant automaton has the following type

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

Figure (1) sketches, how some automaton(f, q) could be "implemented" using a combinatorial component realizing f and a memory unit $\mathcal{D}^q$, which stores data of type σ and its initial value is q.

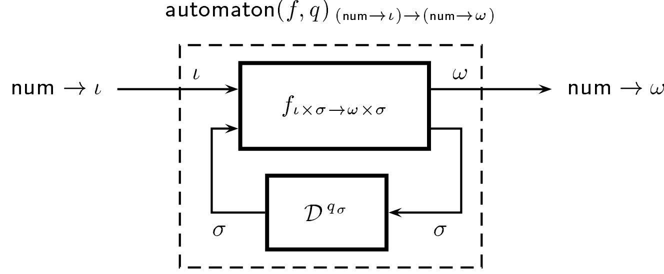

Figure1. Automaton

The constant automaton will formally be defined by means of another constant named automaton'. automaton' is similar to automaton except that the set of states are also visible (see figure 2). Hence the constant automaton' has the type:

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

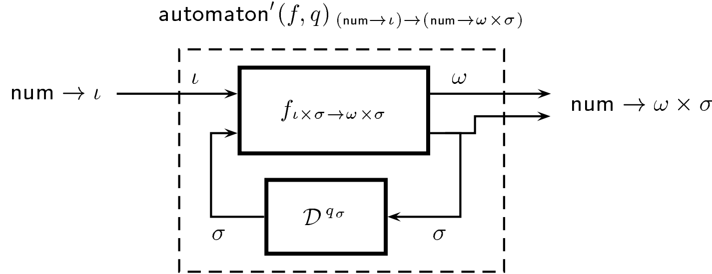

Figure2. Automaton'

automaton' is defined by means of primitive recursion over natural numbers, which represent time. For a given $i_{\mathsf{num}\to\iota}$, the expression (automaton' (f, q) i)

denotes the output and the present state and $(\mathsf{automaton'}\,(f,q)\; i\; t)$ 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 $i(t)$ 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 $i(\mathsf{SUC}\; t')$ and the current state s. Since $(\mathsf{automaton'}\,(f,q)\, i\, t')$ produces a pair corresponding to the output and the state, the function SND is applied in order to extract the state from this result.

$$\begin{array}{l} \vdash \big(\mathsf{automaton'}\,(f,q)\; i\; 0 \;=\; f(i(0),q)\,\big) \;\wedge \\ \quad \big(\mathsf{automaton'}\,(f,q)\; i\; (\mathsf{SUC}\; t') \;= \\ \quad\; \mathsf{let} \\ \quad\;\; s = \mathsf{SND}(\mathsf{automaton'}\,(f,q)\; i\; t') \\ \quad\; \mathsf{in} \\ \quad\;\; f(i(\mathsf{SUC}\; t'), s)\,\big) \end{array} \tag{1}$$

Now $\mathsf{automaton}$ can be defined as

$$\vdash \mathsf{automaton}\,(f,q)\; i\; t \;=\; \mathsf{FST}(\mathsf{automaton'}\,(f,q)\; i\; t) \tag{2}$$

Example

A simple traffic light controller is to be described based on the constant $\mathsf{automaton}$. The controller has two boolean inputs *reset* and *up*. So ι becomes $\mathsf{bool} \times \mathsf{bool}$. There are three outputs named *ron*, *yon* and *gon*. Each corresponds to one single light and determines whether this light is on or off. All outputs are of type bool and so ω becomes $\mathsf{bool} \times \mathsf{bool} \times \mathsf{bool}$.

To represent the state s of the traffic light controller, a simple enumeration type named ryg with values red, yellow and green is used. The type of the output and transition function f is as follows:

$$\big(\underbrace{(\underbrace{\mathsf{bool}}_{reset} \times \underbrace{\mathsf{bool}}_{up})}_{\text{input}} \times \underbrace{\underbrace{\mathsf{ryg}}_{\text{old } s}}_{\text{old state}}\big) \rightarrow \big(\underbrace{(\underbrace{\mathsf{bool}}_{ron} \times \underbrace{\mathsf{bool}}_{yon} \times \underbrace{\mathsf{bool}}_{gon})}_{\text{output}} \times \underbrace{\underbrace{\mathsf{ryg}}_{\text{new } s}}_{\text{new state}}\big)$$

The definitions of f and q are as follows:

$$\mathsf{q} \;=\; \mathsf{green}$$

$$\begin{array}{ll} \mathsf{f}((\mathsf{F},\mathsf{F}),\mathsf{red}) & = ((\mathsf{T},\mathsf{F},\mathsf{F}),\mathsf{red}) \;\wedge \\ \mathsf{f}((\mathsf{F},\mathsf{F}),\mathsf{yellow}) & = ((\mathsf{F},\mathsf{T},\mathsf{F}),\mathsf{yellow}) \;\wedge \\ \mathsf{f}((\mathsf{F},\mathsf{F}),\mathsf{green}) & = ((\mathsf{F},\mathsf{F},\mathsf{T}),\mathsf{green}) \;\wedge \\ \mathsf{f}((\mathsf{F},\mathsf{T}),\mathsf{red}) & = ((\mathsf{T},\mathsf{F},\mathsf{F}),\mathsf{yellow}) \;\wedge \\ \mathsf{f}((\mathsf{F},\mathsf{T}),\mathsf{yellow}) & = ((\mathsf{F},\mathsf{T},\mathsf{F}),\mathsf{green}) \;\wedge \\ \mathsf{f}((\mathsf{F},\mathsf{T}),\mathsf{green}) & = ((\mathsf{F},\mathsf{F},\mathsf{T}),\mathsf{red}) \;\wedge \\ \mathsf{f}((\mathsf{T},\mathsf{F}),x) & = ((\mathsf{T},\mathsf{F},\mathsf{F}),\mathsf{red}) \;\wedge \\ \mathsf{f}((\mathsf{T},\mathsf{T}),x) & = ((\mathsf{T},\mathsf{F},\mathsf{F}),\mathsf{red}) \end{array}$$

The expression automaton(f, q) has the following type:

$$\underbrace{(\mathsf{num}}_{\text{time}} \to \underbrace{(\underbrace{\mathsf{bool}}_{\textit{reset}} \times \underbrace{\mathsf{bool}}_{\textit{up}}))}_{\text{input}} \;\to\; \underbrace{(\mathsf{num}}_{\text{time}} \to \underbrace{(\underbrace{\mathsf{bool}}_{\textit{ron}} \times \underbrace{\mathsf{bool}}_{\textit{yon}} \times \underbrace{\mathsf{bool}}_{\textit{gon}}))}_{\text{output}}$$

3 Special Cases of Automata

As already mentioned, we intend to use automata to describe both combinatorial and sequential circuits. We will now define two constants named combinatorial_block (for purely combinatorial circuits) and memory_block (for memory parts) and we will explain, how they are related to the previously defined automaton.

A combinatorial cuircuit 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). Definition:

$$\vdash \ \mathsf{combinatorial_block}\ e\ i\ t \ = \ e(i(t)) \tag{3}$$

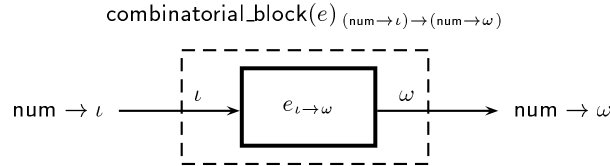

Figure3. Combinatorial Circuits

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

$$\begin{aligned} \vdash \ & \mathsf{memory_block}\ \mathit{init}\ i\ 0 &&= \mathit{init}\ \wedge \\ & \mathsf{memory_block}\ \mathit{init}\ i\ (\mathsf{SUC}t) &&= i(t) \end{aligned} \tag{4}$$

One can represent a combinatorial circuit by an ordinary automaton, where the type of the state is one. one is a HOL standard data type with only one element. The constant $\mathsf{one}_{\mathsf{one}}$ represents its unique element.

$$\vdash \ \mathsf{combinatorial_block}\ e \ = \ \mathsf{automaton}\ ((\lambda(x, y_{\mathsf{one}}).\ (e(x), \mathsf{one})),\ \mathsf{one}) \tag{5}$$

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.

$$\vdash \ \mathsf{memory_block}\ \mathit{init} \ = \ \mathsf{automaton}\ ((\lambda(x, y).\ (y, x)),\ \mathit{init}) \tag{6}$$

4 Reachability of States

Using the definition of an automaton given in section 2, we can define the concept of reachability. The constant reachable maps an automaton given by (f, q) onto a predicate which indicates if some state s may be reached or not. reachable has the following type:

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

reachable is defined by means of a constant definition using automaton'. The definition states that a state s is reachable iff there is some input sequence $i_{\mathsf{num} \rightarrow \iota}$ and some time t such that the current state, i.e. $\mathsf{SND}(\mathsf{automaton'}\,(f, q)\,i\,t)$, becomes s.

$$\vdash \ \mathsf{reachable}\,(f, q)\ s \ = \ (\exists i, t.\ \mathsf{SND}(\mathsf{automaton'}\ (f, q)\ i\ t) \ = s) \tag{7}$$

Theorem (8) states, that the initial state q is reachable. Theorem (9) states, that if some s is reachable then so is any successor state $\mathsf{SND}(f(a, s))$ for arbitrary input x.

$$\vdash \ \mathsf{reachable}\,(f, q)\ q \tag{8}$$

$$\vdash \ (\ \mathsf{reachable}\,(f, q)\ s\ ) \ \Rightarrow \ (\ \forall x.\ \mathsf{reachable}\,(f, q)\ (\mathsf{SND}(f(x, s)))\ ) \tag{9}$$

When encoding states of automata later on in this paper, we will have to find subsets of states, that cover all reachable states. Given a predicate $P_{\sigma \rightarrow \mathsf{bool}}$ indicating the chosen subset, we can prove the theorem, that P covers all reachable states in an inductive manner using theorem (10).

$$\begin{array}{l} \vdash \ \forall P. \\ \quad (\\ \quad\ P(q) \ \wedge \\ \quad\ (\forall s.\ P(s) \ \Rightarrow \ (\forall x.\ P(\mathsf{SND}(f(x, s))))) \\ \quad) \\ \quad \Rightarrow \ \big(\ \forall s.(\mathsf{reachable}\,(f, q)\ s) \ \Rightarrow \ P(s)\ \big) \end{array} \tag{10}$$

Theorem (10) states, that P covers all reachable states if

1. the initial state q is in the subset described by P, and
2. for all states s within this subset any succeeding state $\mathsf{SND}(f(x, s))$ (for arbitrary input x) is also in this subset.

5 Transformations on Automata

Equivalence of automata means that for a given input, they produce the same output. In other words, two automata (f, q) and $(\tilde{f}, \tilde{q})$ are called equivalent iff $\mathsf{automaton}(f, q) = \mathsf{automaton}(\tilde{f}, \tilde{q})$.

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. In this section we will present automata transformations which go beyond this — namely those, where the states are represented in a different manner, the number of states differs, etc. .

In this section, we will first introduce a more general state encoding theorem, then derive two corollaries to this theorem and finally we introduce a theorem for removing redundant memory parts.

5.1 The State Encoding Theorem

The general state encoding theorem has two technical applications: encoding the data types of the state and elimination of unreachable states.

$$
\begin{array}{l}
\vdash\ (\forall s.\ (\mathsf{reachable}\ (f,q)\ s)\ \Rightarrow\ h(g(s)) = s) \\
\quad \Rightarrow \\
\quad (\\
\quad\ \ \mathsf{automaton}\ (f,q)\ = \\
\quad\ \ \ \mathsf{let} \\
\quad\ \ \ \ \ \tilde{f} = (\lambda(v,x).(\lambda(y,z).\ (y,g(z)))(f(v,h(x))))\ \ \mathsf{and} \\
\quad\ \ \ \ \ \tilde{q} = g(q) \\
\quad\ \ \ \mathsf{in} \\
\quad\ \ \ \ \ \mathsf{automaton}(\tilde{f},\tilde{q}) \\
\quad)
\end{array}
\tag{11}
$$

The left hand side of the implication states in theorem (11), that there functions g and h fulfilling $h(g(s)) = s$ for all reachable states. g maps a value of type σ to a value of some type σ' and h maps this value back to the former one (see figure 4).

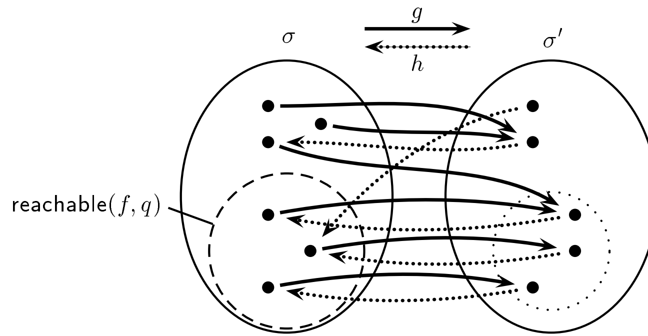

Figure4. Encoding from σ to σ'

The right hand side of theorem (11) states, that the automata $\mathsf{automaton}(f,q)$ and $\mathsf{automaton}(\tilde{f},\tilde{q})$ are equivalent. $\tilde{f}$ and $\tilde{q}$ have been derived from f, q, g and

h. The new initial state $\tilde{q}$ has been obtained by encoding q. The new output and transition function $\tilde{f}$ has been derived from f by encoding every state input and decoding every state output.

Figure 5 illustrates, how the new automaton looks like. Theorem (11) states that provided the above mentioned assumption, the automata in figure 1 and 5 are equivalent.*

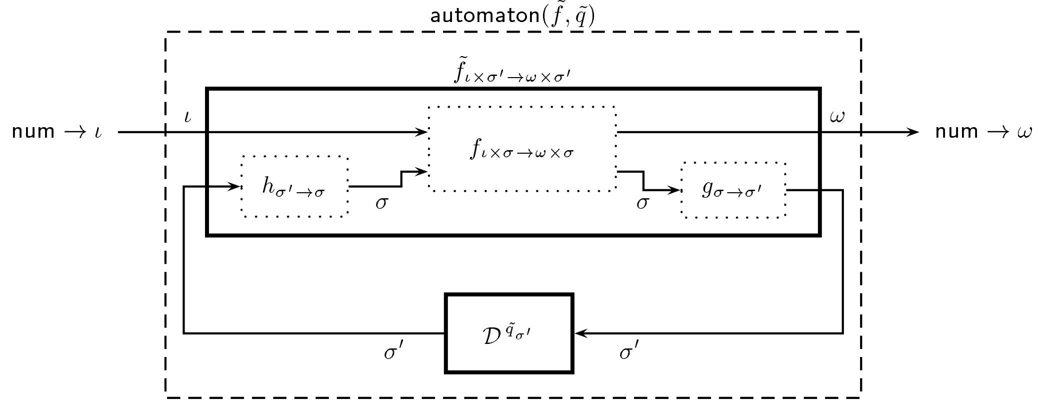

Figure5. State Encoding

Corollary A Determining reachability can only be performed for small sized automata and theorem (11) is applied to pure encoding problems. The following corollary is convenient for this purpose:

$$
\begin{array}{ll}
\vdash & (\forall s.\, h(g(s)) = s) \\
 & \Rightarrow \\
 & (\\
 & \ \mathsf{automaton}\ (f, q)\ = \\
 & \ \ \mathsf{let} \\
 & \ \ \ \tilde{f} = (\lambda(v,x).(\lambda(y,z).\,(y, g(z)))(f(v, h(x))))\ \ \mathsf{and} \\
 & \ \ \ \tilde{q} = g(q) \\
 & \ \ \mathsf{in} \\
 & \ \ \ \mathsf{automaton}(\tilde{f}, \tilde{q}) \\
 &)
\end{array}
\qquad (12)
$$

* This automata encoding transformation with its pair of encoding/decoding functions (g, h) resembles the type definition mechanism of HOL [Melh88]. However, in state encoding of automata, the new type may have some extra elements. Furthermore, the subset of states to be encoded cannot be an arbitrary nonempty set as in type definitions but must cover at least all reachable states of the automaton.

In contrast to theorem 11, theorem 12 performs the state encoding for the entire set of states — reachability need not be considered.

Before this corollary can be applied, an appropriate encoding in terms of h and g has to be found and it has to be proven, that the encoding is correct, i.e. $\forall s.\, h(g(s))$ holds. 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 $\forall s.\, h(g(s))$ 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. We will present an approach in section 7.

Example

In our traffic light example, symbolic values were used to describe the state the controller. To convert this RT-level circuit description into a gate level description, states have to be encoded using boolean values. We will describe to different implementation alternatives: $\mathsf{automaton}(\mathsf{f}', \mathsf{q}')$ and $\mathsf{automaton}(\mathsf{f}'', \mathsf{q}'')$. Both $\mathsf{automaton}(\mathsf{f}', \mathsf{q}')$ and $\mathsf{automaton}(\mathsf{f}'', \mathsf{q}'')$ are equivalent to $\mathsf{automaton}(\mathsf{f}, \mathsf{q})$. They are derived by means of state encoding using the encodings $(\mathsf{g}', \mathsf{h}')$ and $(\mathsf{g}'', \mathsf{h}'')$, respectively.

$(\mathsf{g}', \mathsf{h}')$ is a minimal bit encoding, where only two bits are used:

$$
\begin{array}{ll}
\mathsf{g}'(\mathsf{red}) & = (\mathsf{F}, \mathsf{F}) \;\wedge \\
\mathsf{g}'(\mathsf{yellow}) & = (\mathsf{F}, \mathsf{T}) \;\wedge \\
\mathsf{g}'(\mathsf{green}) & = (\mathsf{T}, \mathsf{F})
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{h}'(\mathsf{F}, \mathsf{F}) & = \mathsf{red} \;\wedge \\
\mathsf{h}'(\mathsf{F}, \mathsf{T}) & = \mathsf{yellow} \;\wedge \\
\mathsf{h}'(\mathsf{T}, \mathsf{F}) & = \mathsf{green} \;\wedge \\
\mathsf{h}'(\mathsf{T}, \mathsf{T}) & = \mathsf{red}
\end{array}
$$

Obviously, the state (T, T) remains unused and $\mathsf{h}'(\mathsf{g}'\, s)$ is fulfilled no matter how the result of h' is defined for (T, T). Besides red, every other value could have been chosen, and it would also be possible to leave this decision open at this moment and instantiate the value later on during boolean optimizations.

Applying the $(\mathsf{g}', \mathsf{h}')$ state encoding leads to

$$\vdash\ \mathsf{automaton}(\mathsf{f}, \mathsf{q}) = \mathsf{automaton}(\mathsf{f}', \mathsf{q}')$$

with:

$$\mathsf{q}' \;=\; (\mathsf{T},\mathsf{F})$$

$$\begin{aligned}
&\mathsf{f}'((\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{F})) = ((\mathsf{T},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{F})) \;\wedge\\
&\mathsf{f}'((\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{T})) = ((\mathsf{F},\mathsf{T},\mathsf{F}),(\mathsf{F},\mathsf{T})) \;\wedge\\
&\mathsf{f}'((\mathsf{F},\mathsf{F}),(\mathsf{T},\mathsf{F})) = ((\mathsf{F},\mathsf{F},\mathsf{T}),(\mathsf{T},\mathsf{F})) \;\wedge\\
&\mathsf{f}'((\mathsf{F},\mathsf{T}),(\mathsf{F},\mathsf{F})) = ((\mathsf{T},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{T})) \;\wedge\\
&\mathsf{f}'((\mathsf{F},\mathsf{T}),(\mathsf{F},\mathsf{T})) = ((\mathsf{F},\mathsf{T},\mathsf{F}),(\mathsf{T},\mathsf{F})) \;\wedge\\
&\mathsf{f}'((\mathsf{F},\mathsf{T}),(\mathsf{T},\mathsf{F})) = ((\mathsf{F},\mathsf{F},\mathsf{T}),(\mathsf{F},\mathsf{F})) \;\wedge\\
&\mathsf{f}'((\mathsf{T},\mathsf{F}),(x,y)) = ((\mathsf{T},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{F})) \;\wedge\\
&\mathsf{f}'((\mathsf{T},\mathsf{T}),(x,y)) = ((\mathsf{T},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{F}))
\end{aligned}$$

$(\mathsf{g}'',\mathsf{h}'')$ is a one hot encoding. For the one hot encoding three bits are required but only the states $(\mathsf{F},\mathsf{F},\mathsf{T})$, $(\mathsf{F},\mathsf{T},\mathsf{F})$ and $(\mathsf{T},\mathsf{F},\mathsf{F})$ are used. Since the outputs also correspond to the control states, this approach helps minimizing the combinatorial logic required for the implementation.

$$\begin{aligned}
&\mathsf{g}''(\mathsf{red}) = (\mathsf{F},\mathsf{F},\mathsf{T}) \;\wedge\\
&\mathsf{g}''(\mathsf{yellow}) = (\mathsf{F},\mathsf{T},\mathsf{F}) \;\wedge\\
&\mathsf{g}''(\mathsf{green}) = (\mathsf{T},\mathsf{F},\mathsf{F})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{h}''(\mathsf{F},\mathsf{F},\mathsf{T}) = \mathsf{red} \;\wedge\\
&\mathsf{h}''(\mathsf{F},\mathsf{T},\mathsf{F}) = \mathsf{yellow} \;\wedge\\
&\mathsf{h}''(\mathsf{T},\mathsf{F},\mathsf{F}) = \mathsf{green} \;\wedge\\
&\mathsf{h}''(\mathsf{F},\mathsf{F},\mathsf{F}) = \mathsf{red} \;\wedge\\
&\mathsf{h}''(\mathsf{F},\mathsf{T},\mathsf{T}) = \mathsf{red} \;\wedge\\
&\mathsf{h}''(\mathsf{T},\mathsf{F},\mathsf{T}) = \mathsf{red} \;\wedge\\
&\mathsf{h}''(\mathsf{T},\mathsf{T},\mathsf{F}) = \mathsf{red} \;\wedge\\
&\mathsf{h}''(\mathsf{T},\mathsf{T},\mathsf{T}) = \mathsf{red}
\end{aligned}$$

Applying the $(\mathsf{g}'',\mathsf{h}'')$ state encoding leads to

$$\vdash \mathsf{automaton}(\mathsf{f},\mathsf{q}) = \mathsf{automaton}(\mathsf{f}'',\mathsf{q}'')$$

with:

$$\mathsf{q}'' \;=\; (\mathsf{T},\mathsf{F})$$

$$\begin{aligned}
&\mathsf{f}''((\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{F},\mathsf{T})) = ((\mathsf{T},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{F},\mathsf{T})) \;\wedge\\
&\mathsf{f}''((\mathsf{F},\mathsf{T}),(\mathsf{F},\mathsf{T},\mathsf{F})) = ((\mathsf{F},\mathsf{T},\mathsf{F}),(\mathsf{F},\mathsf{T},\mathsf{F})) \;\wedge\\
&\mathsf{f}''((\mathsf{F},\mathsf{T}),(\mathsf{T},\mathsf{F},\mathsf{F})) = ((\mathsf{F},\mathsf{F},\mathsf{T}),(\mathsf{T},\mathsf{F},\mathsf{F})) \;\wedge\\
&\mathsf{f}''((\mathsf{F},\mathsf{T}),(\mathsf{F},\mathsf{F},\mathsf{T})) = ((\mathsf{T},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{T},\mathsf{F})) \;\wedge\\
&\mathsf{f}''((\mathsf{F},\mathsf{T}),(\mathsf{F},\mathsf{T},\mathsf{F})) = ((\mathsf{F},\mathsf{T},\mathsf{F}),(\mathsf{T},\mathsf{F},\mathsf{F})) \;\wedge\\
&\mathsf{f}''((\mathsf{F},\mathsf{T}),(\mathsf{T},\mathsf{F},\mathsf{F})) = ((\mathsf{F},\mathsf{F},\mathsf{T}),(\mathsf{F},\mathsf{F},\mathsf{T})) \;\wedge\\
&\mathsf{f}''((\mathsf{T},\mathsf{F}),(x,y,z)) = ((\mathsf{T},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{F},\mathsf{T})) \;\wedge\\
&\mathsf{f}''((\mathsf{T},\mathsf{T}),(x,y,z)) = ((\mathsf{T},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{F},\mathsf{T}))
\end{aligned}$$

Corollary B Corollary B to theorem (11) is dedicated to pure state reduction problems. It is assumed, that one has divided σ into $\sigma^1 + \sigma^2$, where all the reachable states are in σ^1. In this situation, the state representation can be cut

down to σ^1 using the following pair of encoding/decoding functions g^B and h^B. g^B is introduced by means of a constant specification. The variable z may be instantiated in an arbitrary manner to derive some "concrete" g^B.

$$\vdash\ \exists z.\,(\mathsf{g}^B(\mathsf{INL}\,x) = x) \wedge (\mathsf{g}^B(\mathsf{INR}\,y) = z(y))$$
$$\vdash\ h^B = \mathsf{INL}$$

Remark: It is not demanded, that σ^1 represents exactly the set of all reachable states. It must cover all reachable states, but there may also be some unreachable states.

$$\begin{array}{l}\vdash\ (\ (\forall s.\,(\mathsf{reachable}\,(f,q)\;s))\ \Rightarrow\ \mathsf{ISL}(s)\ )\\ \quad\Rightarrow\\ \quad(\\ \quad\ \mathsf{automaton}\,(f,q)\ =\\ \quad\ \ \mathsf{let}\\ \quad\ \ \ \tilde{f} = (\lambda(v,x).(\lambda(y,z).\,(y,\mathsf{g}^B(z)))(f(v,\mathsf{h}^B(x))))\ \mathsf{and}\\ \quad\ \ \ \tilde{q} = \mathsf{g}^B(q)\\ \quad\ \ \mathsf{in}\\ \quad\ \ \ \mathsf{automaton}(\tilde{f},\tilde{q})\\ \quad)\end{array} \tag{13}$$

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.

5.2 Elimination of Redundant Memory Parts

The last theorem to be introduced describes, how parts of the memory can be omitted if these parts are of no importance for the output and transition function f. This theorem can be used for removing flipflops with unconnected outputs from a synchronous circuit description.

Let us assume, that the type of the states σ is a scalar product of two types $\sigma^1 \times \sigma^2$ and that f is $(\lambda(x,(s^1,s^2)).\,f'(x,s^1))$ for some f'. In other words, f depends on the input and on the left hand side of the pair $(s^1,s^2)_{\sigma^1\times\sigma^2}$ representing the state but not on the right hand side. Theorem (14) states, that this automata (f,q) is equivalent to the automaton (f',q^1).

$$\begin{array}{l}\vdash\ \ \mathsf{let}\\ \quad\ \ f = (\lambda(x,(s^1,s^2)).\,f'(x,s^1))\ \mathsf{and}\\ \quad\ \ q = (q^1,q^2)\\ \quad\ \mathsf{in}\\ \quad\ \ \mathsf{automaton}(f,q)\\ \quad=\\ \quad\ \mathsf{automaton}(f',q^1)\end{array} \tag{14}$$

6 Systematic Derivation of State Encodings

The automata theory provides several pairs of encoding/decoding functions for the following set of data types useful for RT and gate level circuit descriptions. These theorems are intended for pure encodings according to corollary B.

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

On the gate level, booleans shall also be used for representing signal values and the scalar product shall be used for constructing compound signals. On the RT level, more complex data types such as enumeration types, natural numbers, records and variants can be used. Additionally, one, num , (α)option and $\alpha + \beta$ shall also be used for representing data types at the RT level.

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 viceversa. Table 1 lists some useful encoding theorems and describes which types they are related to.

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 natural numbers 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 viceversa).

option, + and $\times$ are all type operators. The theorems OPTION_TRANS, SUM_TRANS and PROD_TRANS derive encodings for these type operators, i.e. under the assumptions that there are encodings for their parameters — let us say some $\alpha \rightleftharpoons \alpha'$ and $\beta \rightleftharpoons \beta'$ — the encoding for the entire type expressions (α)option, $\alpha + \beta$ and $\alpha \times \beta$, respectively, can be 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 encodings in the theorems OPTION_EXTEND, SUM_EXTEND and PROD_EXTEND are

** $\rightharpoonup$ only for natural numbers < 2

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

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

Table1. Encodings For Simple Data Types

applicable only in one direction. They all lead to "bigger" types in the sense that the new type contains some extra elements.

7 Algorithms for Deriving Correct Encodings

7.1 The Task

We have applied the automata theory to formally describe behavioural circuit descriptions of a synchronous VHDL subset. For a given behavioural description, we extracted the automata description in terms of its initial state q and the output and transition function f. In these automata derived from synchronous VHDL, 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, we used type expressions built with one, option and $+$. To derive a representation on the gate level, these types have to be mapped by tuples of booleans, i.e. data types bool and $\times$. There usually is a broad range of correct encodings. Let us assume, that only the number of bits is to be minimized and that every possible representation with a minimum number of bits is an appropriate encoding.

Each control state represents either the starting point or one of the `wait`-statement positions in the VHDL program. We will not 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 (sequence, if-then-else) consisting of two parts 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.

7.2 Derivation of a Minimal Bit Encoding

We will illustrate the minimal bit encoding algorithm by an example. Let us assume, that σ^c is as follows:

$$(\mathsf{one} + (\mathsf{one})\mathsf{option})\mathsf{option} + (\mathsf{one} + \mathsf{one}) \qquad \text{(I)}$$

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

$$(\mathsf{one} + (\mathsf{one} + (\mathsf{one} + \mathsf{one}))) + (\mathsf{one} + \mathsf{one}) \qquad \text{(II)}$$

Balancing 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 occurences. 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 $(\mathsf{one} + (\mathsf{one} + (\mathsf{one} + \mathsf{one})))$. 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:

$$((\mathsf{one} + \mathsf{one}) + (\mathsf{one} + \mathsf{one})) + (\mathsf{one} + \mathsf{one}) \qquad \text{(III)}$$

Extension 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 $\mathsf{one} + \mathsf{one}$ 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.

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

Substitution of + and 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.

$$\mathsf{bool} \times (\mathsf{bool} \times (\mathsf{bool} \times \mathsf{one})) \tag{V}$$

Finally SUM_NEUTRAL is applied to encode $\mathsf{bool} \times \mathsf{one}$ by bool.

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

7.3 Derivation of a One Hot Encoding

We use the same example σ^c as in the minimal bit encoding example:

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

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

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

Flattening Applying SUM_ASSOC repeatedly leads to:

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

Substitution of + and 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 encodes each repeatedly produces:

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

The previous type expression consisted of 6 states, where each of them corresponds to one one-subexpression.

8 Conclusion and Future Work

We have introduced a theory for automata representation and transformation. The transformations defined are constructive and hence lead to refinements and optimizations on the automata through different levels of abstraction. An illustration of how state encodings can be derived in a formal synthesis fashion was also given. The state encoding algorithm presented is similar to conventional synthesis algorithms except that correctness is guaranteed implicitly, since the algorithm is based on HOL.

Such formal synthesis algorithms offer an alternative to the conventional synthesis/verification approach. We believe, that in general formal synthesis can be much more efficient than synthesis combined with an extra verification step. The result of a non-formal synthesis is just the implementation, the information on *how* the implementation is derived gets lost and cannot be used during the post-synthesis verification step.

We believe, that formal synthesis algorithms can also be exploited in other areas of hardware synthesis such as boolean optimization, scheduling, system level synthesis. The automata theory will be a basis for circuit descriptions on the algorithmic and system level.

References

[Day92] Nancy Day. A comparison between statecharts and state transition assertions. In [hug92], pages 247–262.

[EiSK93] D. Eisenbiegler, K. Schneider, and R. Kumar. A functional approach for formalizing regular hardware structures. In [hug93], pages 101–114.

[hug92] Luc Claesen and Michael Gordon, editors. *Higher Order Logic Theorem Proving and Its Applications*, Leuven, Belgium, November 1992. North-Holland.

[hug93] Jeffrey J. Joyce and Carl-Johan H. Seger, editors. *Higher Order Logic Theorem Proving and Its Applications*, Vancouver, B.C., Canada, August 1993. Springer.

[Loew92] Paul Loewenstein. A formal theory of simulations between infinite automata. In [hug92], pages 227–246.

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

[ScKK93] R. Kumar K. Schneider and Thomas Kropf. Alternative proof procedures for finite-state machines in higher-order logic. In [hug93], pages 213–226.

This article was processed using the LaTeX macro package with LLNCS style