Deriving Structural RT-Implementations from Algorithmic Descriptions by means of Logical Transformations *

Christian Blumenröhr, Dirk Eisenbiegler
Institute for Circuit Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid),
University of Karlsruhe, Germany e–mail: {blumen,eisen}@ira.uka.de

Abstract

This paper presents a formal synthesis approach where the mapping of an algorithmic description towards its structural implementation at the RT-level is performed by means of basic logical transformations in a higher order logic theorem prover. The approach goes beyond pure basic blocks and allows representing and synthesizing arbitrary computable programs. A functional style is used for representing algorithms based on basic μ-recursive operators whose semantics is defined within higher order logic.

1 Introduction

With synthesis heading towards more and more abstract design levels, writing sound synthesis programs is becoming an important matter of scientific investigations. Due to the complexity of nowadays circuits and due to the considered abstraction level, both exhaustive simulation and full automatic verification approaches fail. The only reasonable approach towards producing correct synthesis results lies in a synthesis process based on correctness preserving transformations (see [Camp89]). As at the gate level, where circuits can easily be formalized and transformed according to a boolean calculus, there is a need to also find formal representations at higher levels of abstraction as on the algorithmic level.

There are many approaches, where synthesis is based on a sequence of behavior preserving transformations such as YSC [Camp89], CAMAD [PeKu94] and TRADES [MHMP96]. They all provide a "correctness by construction" approach with a fixed set of circuit transformations that are assumed to be correct or where there are paper&pencil proofs for their correctness. However, the implementations of the basic circuit transformations are comparatively complex and critical as to correctness. In our approach, circuit transformations are performed by applying basic mathematical rules within a theorem proving environment named HOL [GoMe93] thus guaranteeing correctness implicitly. We will call this design style *formal synthesis* (see [KBES96]). Most formal synthesis approaches introduced in the past are restricted to lower levels of abstraction (e.g. Lambda/Dialog [MaFo91],T-Ruby [ShRa95]).

This paper addresses formal synthesis at the algorithmic level. It is part of our ongoing work towards a formal synthesis tool named HASH (higher order logic applied to synthesis

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

data flow graphs. The extension to be presented in this paper allows synthesizing arbitrary algorithmic descriptions i.e. mixed control and data flow descriptions.

Starting point for high-level synthesis is an algorithmic description. The result of high-level synthesis is a structure at the RT-level consisting of a data-path and a controller. In conventional high-level synthesis, the first step is compiling the description into a control/data flow graph (CDFG) representation [GDWL94]. Based on this representation, several control states are introduced along the graph thus partitioning the graph into small cycle free pieces of program each corresponding to one clock tick. However, the number of the resulting subgraphs grows exponentionally with the nesting depth of the control structures. Then scheduling, allocation and binding are performed on the subgraphs leading to a symbolic state transition table and a separated data-path. Afterwards, the controller and the communication part are generated.

Our formal synthesis approach differs from that method. Based on a formalization of μ-recursion (see section 2), a set of theorems for deriving structural RT-level implementations from algorithmic descriptions has been proven. They can be applied to perform synthesis in two steps: First the program is transformed into an equivalent program with only one single while-loop (section 5). Afterwards a pre-proven theorem is applied mapping the transformed algorithmic description to a RT-level structure (section 4). The RT-level implementation produced in section 4 is based on a formalization scheme that is introduced in section 3.

2 Formal representation of programs

According to Church's Thesis, there are several equivalent schemes for describing computable functions. To be able to talk of algorithmic transformations in logic, one first has to define a programming language with a formal semantics. This paper uses simply typed λ-terms for representing computable functions (see [GoMe93]). The approach extends a formalization scheme for μ-recursion as it was presented in [EiSK93] and is part of our new hardware description language called GROPIUS. This language consists of four parts, each corresponding to one abstraction level from the gate- upto the system-level. GROPIUS has been developed as an intermediate representation format for our formal synthesis approach. To allow the designer to specify an algorithm using an imperative language, it is planned to offer a conversion from C into our representation style at the algorithmic level.

In this approach, the elementary parts within a μ-recursive program are basic blocks, conditions and constants. Basic blocks and conditions are compound expressions consisting of basic, nonrecursive operators. Functional expressions are to be used to describe these expressions, that correspond to (acyclic) data flow graphs. In our approach, a data flow graph (dfg) has the following syntax:

vblock	::=	variable | "(" { vblock "," } vblock ")"
expr	::=	variable | "(" { expr "," } expr ")" | operator "(" expr ")"
dfg	::=	"λ" vblock "." { "`let`" vblock "=" expr "`in`" } expr

The basic operations indicated by operator always terminate and so does the entire function. Basic blocks and conditions both are data flow graphs. Basic blocks have type $\alpha \to \alpha$ and conditions have type $\alpha \to \mathsf{bool}$.

When dealing with μ-recursion, one has to be aware of the fact that function applications may not terminate. Therefore, in this approach, the datatype (α) partial has been

introduced to express results of μ recursive functions.

$$\mathsf{partial} = \mathsf{Defined\ of}\ \alpha \mid \mathsf{Undefined}$$

A μ-recursive function P has type $\alpha \to (\beta)\mathsf{partial}$. $P(x) = \mathsf{Undefined}$ indicates that the function did not terminate and $P(x) = (\mathsf{Defined}\ y)$ indicates that the function did terminate with y as result.

Before introducing programs, i.e. arbitrary μ-recursive functions, we will first introduce blocks. Blocks are μ-recursive functions, but there is a restriction as to their type which is always $\alpha \to (\alpha)\mathsf{partial}$ instead of $\alpha \to (\beta)\mathsf{partial}$ for some arbitrary β. There are five control structures for building blocks based on basic blocks, conditions and constants: PARTIALIZE, THEN, IFTE, WHILE and LOCVAR. The syntax for blocks is as follows:

block	::=	"`PARTIALIZE`" basic_block | block "`THEN`" block |
		"`IFTE`" condition block block | "`WHILE`" condition block |
		"`LOCVAR`" constant block

PARTIALIZE, THEN, IFTE, WHILE and LOCVAR are functions, that map basic blocks, conditions, blocks and constants to a single block. It has to be noted, that blocks, basic blocks and conditions are functions themselves. Table 1 gives a definition in a mathematical notation. A precise definition in terms of higher order logic in HOL can be found in [EiSK93].

$$\mathsf{PARTIALIZE}\ P\ x = \mathsf{Defined}\ (P(x))$$

$$(P\ \mathsf{THEN}\ Q)\ x = \begin{cases} \mathsf{Defined}\ (y) & \exists z.\ P(x) = \mathsf{Defined}\ (z) \wedge Q(z) = \mathsf{Defined}\ (y) \\ \mathsf{Undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{IFTE}\ C\ P\ Q\ x = \begin{cases} \mathsf{Defined}\ (y) & C(x) \wedge (P(x) = \mathsf{Defined}\ (y)) \vee \\ & \neg(C(x)) \wedge (Q(x) = \mathsf{Defined}\ (y)) \\ \mathsf{Undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{WHILE}\ C\ P\ x = \begin{cases} \mathsf{Defined}\ (y) & \exists n.\ P^n(x) = \mathsf{Defined}\ (y) \wedge \neg(C(y)) \wedge \\ & \quad \forall m.\ m < n \Rightarrow \\ & \quad\quad \exists z.\ P^m(x) = \mathsf{Defined}\ (z) \wedge C(z) \\ \mathsf{Undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{LOCVAR}\ init\ P\ x = \begin{cases} \mathsf{Defined}\ (y) & \exists z.\ P(x, init) = \mathsf{Defined}\ (y, z) \\ \mathsf{Undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{PROGRAM}\ init\ P\ x = \begin{cases} \mathsf{Defined}\ (y) & \exists z.\ P(x, init) = \mathsf{Defined}\ (z, y) \\ \mathsf{Undefined} & \text{otherwise} \end{cases}$$

Table 1: Semantics of control structures

PARTIALIZE is used to turn a basic block P of type $\alpha \to \alpha$ to a block PARTIALIZE(P) of type $\alpha \to (\alpha)\mathsf{partial}$. PARTIALIZE($P$) is a function that maps some x to Defined($P(x)$). It always terminates, Undefined is never reached. THEN is used in infix notation. It maps two blocks to a single block that executes the two blocks consecutively. The second block is executed only if the first block did terminate. The result of $P\,\mathsf{THEN}\,Q$ becomes Undefined, iff one of the two blocks does not terminate. IFTE is a means for expressing conditional block applications. The expression IFTE C P Q maps some x either to $P(x)$ or to $Q(x)$ depending on whether the condition $C(x)$ becomes true or false, respectively. WHILE is used to describe loops. Starting with an initial state x, the body is iterated until a state y is reached that does not fulfill a condition. If such a state does not exist, the result

becomes Undefined. LOCVAR is used to introduce local variables. Given an arbitrary initial value *init* for the local variable, the function is applied to the pair consisting of the actual state x and *init*. If the result is defined, the second part of it, which corresponds to the local variable, will be dropped, and only the first part is returned.

The reason for restricting blocks to type $\alpha \to (\alpha)$partial rather than $\alpha \to (\beta)$partial are loops. To iterate some function, the input and the output must have the same type. However, we also want to allow programs, where input type and output type differ. Therefore, an additional operator named PROGRAM has been defined. It converts a block P of type $\alpha \times \beta \to (\alpha \times \beta)$partial and some initial value *init* of type β to a function (PROGRAM *init* P) of type $\alpha \to (\beta)$partial.

program ::= "PROGRAM" constant block

(PROGRAM *init* P) maps some x of type α to $P(x, init)$. If $P(x, init)$ terminates, then the result is a pair. PROGRAM$(P, init)$ then returns the second component of $P(x, init)$ and drops the first. For a formal definition see table 1.

2.1 An example

The way we formalize programs may at first glance seem a little awkward. Usually, imperative programming languages are used and one needs a WP-calculus or a Hoare-calculus to give the basic language elements a formal semantics, in order to describe mathematically the input/output function. In our approach programs are directly described in mathematical notation.

Figure 1 shows some "ordinary" program in an imperative notation and the corresponding translation into our formalism. The function gcd maps an input pair (p, q)

Imperative Program	Functional Representation
FUNCTION gcd	gcd =
(var p, q : integer) : integer;	PROGRAM 0
VAR a, b : integer	LOCVAR $(0, 0)$
BEGIN	(
$a := p$;	PARTIALIZE $(\lambda((p,q), y, (a,b)).((p,q), y, (p,b))$ THEN
$b := q$;	PARTIALIZE $(\lambda((p,q), y, (a,b)).((p,q), y, (a,q))$ THEN
WHILE $(a \neq b)$ DO	WHILE $(\lambda((p,q), y, (a,b)).a \neq b)$ (
BEGIN	
IF $(a > b)$ THEN	IFTE $(\lambda((p,q), y, (a,b)).a > b)$
$a := a - b$;	PARTIALIZE $(\lambda((p,q), y, (a,b)).((p,q), y, (a-b, b))$
ELSE	
$b := b - a$;	PARTIALIZE $(\lambda((p,q), y, (a,b)).((p,q), y, (a, b-a))$
END;	) THEN
RETURN a;	PARTIALIZE $(\lambda((p,q), y, (a,b)).((p,q), a, (a,b))$
END;	)

Figure 1: Greatest common divisor

onto a single output. The pair (a, b) stands for the auxiliary variables with initial value $(0, 0)$, and y corresponds to the output, which has the initial value 0 and returns the value

initial values for local variables can be fixed.

2.2 Programs in single-loop-form

In the following, programs of a specific pattern named single-loop-form (slf) will play an important role: programs will first be turned into single-loop-form (see section 5) and then a theorem will be applied for mapping single-loop-form programs to hardware (see section 4). Programs in single-loop-form are constructed as follows:

$$\begin{array}{l}\mathsf{PROGRAM}\ \mathit{out\_init}\\ \quad \mathsf{LOCVAR}\ \mathit{var\_init}\\ \quad\quad (\mathsf{PARTIALIZE}\ P)\ \mathsf{THEN}\ (\mathsf{WHILE}\ C\ (\mathsf{PARTIALIZE}\ Q))\ \mathsf{THEN}\ (\mathsf{PARTIALIZE}\ R)\end{array} \tag{1}$$

3 Formal representation at the RT-level

To formally represent hardware and to perform formal synthesis at the RT- and gate level, a theory called "Automata" has been developed. In this paper, we only shortly present the formal representation of automatons, which the next section is based on. A detailed description of the theory can be found in [Eise97b].

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. Here, an automaton will be represented by a pair (P, q), where q is the initial state and P is a compound output and transition function. P has type $\iota \times \sigma \rightarrow \omega \times \sigma$ and q has type σ where ι, ω and σ are the types of the input alphabet, the output alphabet and the state set. Using a compound output and transition function makes sense, since it is not possible to unambiguously assign combinatorial units to either the output function or the transition function. Due to the fact that we only use typed expressions, it is not necessary to explicitly describe the input alphabet, the output alphabet and the state set.

P and q unambiguously determine the behavior of an automaton. The higher order logic function automaton describes the semantics of an automaton by mapping (P, q) to a function automaton(P, q) that maps a time dependent input signal (type num $\rightarrow$ ι) to a time dependent output signal (type num $\rightarrow$ ω). Figure 2 sketches, how some automaton(P, q) could be "implemented" using a combinatorial component realizing P and a memory unit D with initial value q.

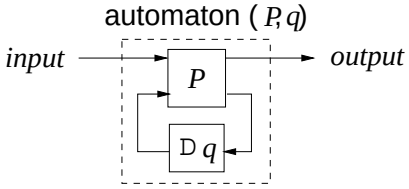

Figure 2: automaton

4 Interface synthesis

Unlike other approaches, we distinguish between the algorithmic i/o-behavior description, and an interface description specifying how the algorithm communicates with the environment. The algorithmic description only specifies the functional relation between some input values and some output values. Time is not yet considered. The interface description maps the algorithm to a specific timing in terms of signal events at the interface of the circuit. In our approach, the designer writes some arbitrary algorithm and can than select among a fixed set of interface description patterns. Since the designer does not have to restart the synthesis from scratch by changing the specification, if another interface behavior is selected as planned previously, this division supports *design reuse*.

On the other hand is this method more restrictive than other approaches, where both descriptions are interwoven. However, we believe that many failures during the synthesis process can be avoided, if one does not mingle i/o-behavior and timing aspect within a single description. In our approach timing and i/o-behavior are developed independently.

There are many possible interface specification patterns. For each interface pattern a specific hardware implementation has to be found and an instantiation theorem has to be proven stating that the implementation definitely implements the algorithm with respect to the interface specification. Usually, the interface of the implementation not only consists of the data signal from the i/o-behavior given by the algorithmic description, but there are also additional control signals introduced such as *start*, *reset* and *ready* to provide a handshake-like communication and to allow interrupting the execution of the algorithm.

Figure 3 shows an interface specification pattern named INTERFACE. It describes the relation between some signals *input*, *start*, *reset*, *output* and *ready* with respect to some arbitrary algorithm S. The specification states that the circuit has an internal state named *ready* with initial value true. *ready* is also forwarded to the output. $ready(t) = \mathsf{F}$ means that at time t the circuit is executing the algorithm, whereas $ready(t) = \mathsf{T}$ means that the circuit is waiting for new input. If no calculation is performed, the *output*-signal holds the result of the last program execution. A calculation can only be started, if *ready* is set, otherwise setting *start* is ignored. After a calculation has been started, *ready* is set to F. If the result is defined, *ready* will be set after a certain time and at the same time the value will appear at the output. The execution of the program can be interrupted by setting the *reset* signal. If the algorithm does not terminate, *ready* will be down until *reset* is set.

For this interface specification pattern we found a general hardware implementation for slf-programs S with arbitrary basic blocks P, C, Q and R and arbitrary values *var_init* and *out_init*. In figure 4 the structure of the implementation is defined in a formal manner using the automaton construct from section 3. The basic blocks P,C,Q and R directly correspond to combinational circuits. Figure 5 gives a graphical representation of this structure. The structure can be divided into two parts: The upper part corresponds to the controller (signals *reset*, *start* and *ready*), the rest of the circuit is used for computations (circuits P, C, Q and R), communication (MUX-circuits) and data storage (D-circuits).

We have proven a general theorem (2) stating that the implementation of *every* slf-program with arbitrary components P, C, Q and R and arbitrary values *var_init* and *out_init* fulfills the given interface specification. After having turned an arbitrary program into a single-loop program, which is to be explained in the next section, one can apply this theorem to produce a hardware structure at the RT-level by means of logical refinement. Using our formal synthesis techniques at the RT- and gate level (see [EiKB97, Eise97b])

$$
\begin{array}{l}
\mathsf{INTERFACE}\ (input, start, reset, output, ready, S) = \\
\quad \neg(start\,0) \Rightarrow ready\,0\ \wedge \\
\quad \forall t.\ \ reset\,t \Rightarrow ready\,t \wedge \\
\qquad ready\,t \wedge \neg(start\,(t+1)) \Rightarrow ready\,(t+1)\ \wedge (output\,(t+1) = output\,t) \wedge \\
\qquad (t = 0\ \Rightarrow\ \mathsf{T} \mid ready\,(t-1))\ \wedge\ start\,t\ \Rightarrow \\
\qquad\quad \mathsf{CASE}\ (S(input\,t))\ \mathsf{OF} \\
\qquad\qquad \mathsf{Defined}\ y\ :\quad \exists m.\ \ (\forall n.\, n \leq m\ \Rightarrow\ \neg(reset\,(t+n)))\ \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad output\,(t+m)\ =\ y\ \wedge\ ready\,(t+m) \wedge \\
\qquad\qquad\qquad\qquad\qquad \forall n.\, n < m\ \Rightarrow\ (\forall p.\, p \leq n\ \Rightarrow\ \neg(reset\,(t+p)))\ \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\quad \neg(ready\,(t+n)) \\
\qquad\qquad \mathsf{Undefined}\ :\ \ \forall m.\, (\forall n.\, n \leq m\ \Rightarrow\ \neg(reset\,(t+n)))\ \Rightarrow\ \neg(ready\,(t+m))
\end{array}
$$

Figure 3: A possible interface behavior

we provide a universal concept for formal synthesis from the algorithmic level down to the gate level.

$$
\begin{array}{l}
\vdash\ \forall P\ C\ Q\ R\ var\_init\ out\_init. \\
\quad \mathsf{IMPLEMENTATION}\ (input, start, reset, output, ready, P, C, Q, R, var\_init, out\_init) \\
\quad \Rightarrow \\
\quad \mathsf{INTERFACE}\ (input, start, reset, output, ready, \\
\qquad \mathsf{PROGRAM}\ out\_init \\
\qquad\quad \mathsf{LOCVAR}\ var\_init \\
\qquad\qquad (\mathsf{PARTIALIZE}\ P)\ \mathsf{THEN}\ (\mathsf{WHILE}\ C\ (\mathsf{PARTIALIZE}\ Q))\ \mathsf{THEN}\ (\mathsf{PARTIALIZE}\ R)
\end{array}
\qquad (2)
$$

5 Converting programs to single-loop-form (slf)

The slf-program is derived by applying a set of pre-proven theorems. During this transformation process, the number of control structures is reduced, and at the same time several boolean auxiliary variables are introduced. In general there is not only one slf-representation for a program, but there are several equivalent slf-programs. According to section 4, each slf-program corresponds to a specific hardware implementation with a specific timing behavior and a specific hardware consumption. Different theorem applications may lead to different equivalent slf-programs with different implementation costs.

5.1 Transformation theorems

We have proven in a very meticulous manner a minimal set of 21 transformation theorems that are necessary for converting *every* program to an equivalent slf-representation. Rewriting with these theorems in an arbitrary order always terminates and always ends up in a slf-program. The result produced does not depend on the order, in which the theorems are applied. Furthermore we provide additional transformation theorems. This allows us to produce different slf-representations leading to different costs in terms of timing and hardware consumption.

Due to lack of space, we cannot introduce all proven theorems. A complete list of the theorems is given in [BlEi98]. As an example we will present four transformation theorems

IMPLEMENTATION $(input, start, reset, output, ready, P, C, Q, R, var\_init, out\_init) =$

$\exists q_1\ q_2\ q_3.\ (\lambda t.\ (output\ t, ready\ t)) =$

automaton

$(\ (\lambda((a,b,c),(w,x,y,z)).$

let $d = b\ \wedge\ z$ in
let $((e,f),g) = P((a, out\_init), var\_init)$ in
let $h = \mathsf{MUX}\ (d,e,y)$ in
let $i = \mathsf{MUX}\ (d,f,x)$ in
let $j = \mathsf{MUX}\ (d,g,w)$ in
let $k = \neg b\ \wedge\ z$ in
let $l = C((h,i),j)$ in
let $((m,n),o) = Q((h,i),j)$ in
let $((p,q),r) = R((h,i),j)$ in
let $s = k\ \vee\ \neg l$ in
let $t = c\ \vee\ s$ in
let $u = \mathsf{MUX}\ (l,n,q)$ in
let $v = \mathsf{MUX}\ (k,i,u)$ in
$((v,t),(o,v,m,t)))$

$,(q_1,q_2,q_3,\mathsf{T})\ )$

$(\lambda t.\ (input\ t, start\ t, reset\ t))$

Figure 4: Representation at RT-level

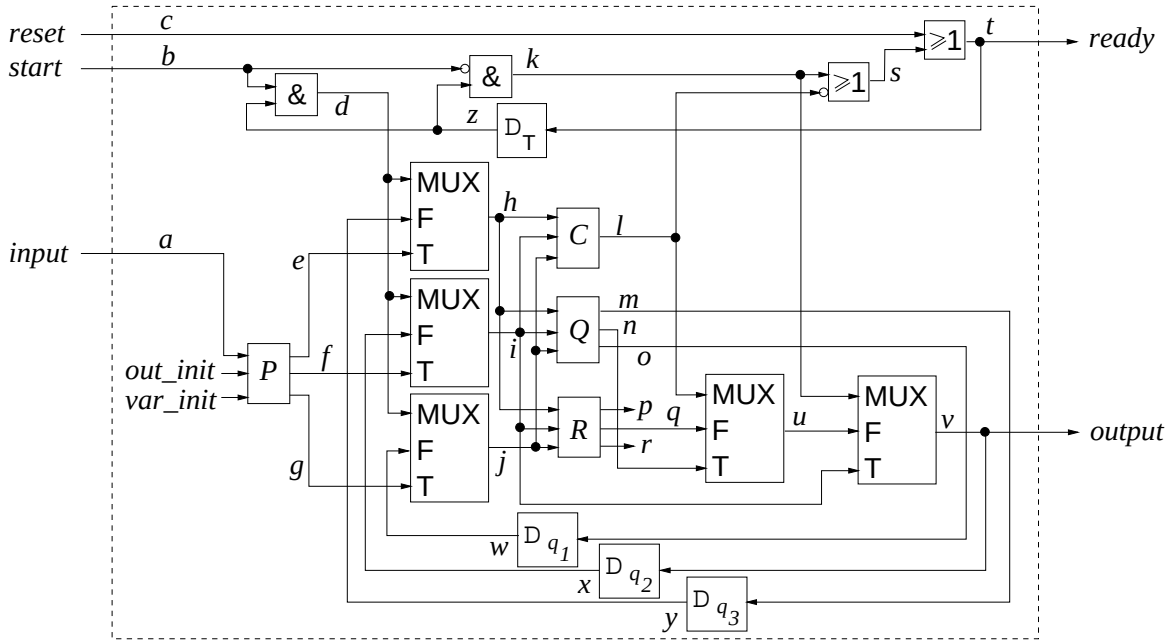

Figure 5: RT-Implementation

(3), (4), (5) and (6). Theorems (4) and (6) are alternatives to (3) and (5), respectively, i.e. they can be applied in the same situation but lead to different costs in terms of timing and hardware consumption.

The theorems (3) and (4) turn a sequence consisting of a basic block and a while-loop with basic block as body into a single while-loop. Theorem (3) introduces a boolean variable with initial value F. P is executed, if the variable's value is F, otherwise Q is executed. The loop condition has changed, so that the body is performed at least once. After executing P in the first run, the local variable is set to T and P will never be executed again. Theorem (4) describes a transformation where after executing P also Q may be performed in the first cycle of the loop.

Theorem (3) leads to an implementation where the operations in P and Q can be shared since they are never executed in the same clock cycle. However the implementation becomes comparatively slow. Theorem (4) on the other hand leads to a faster implementation with a higher hardware consumption since P and Q can be executed in the first clock cycle and the loop-condition must be implemented twice.

$$
\begin{array}{l}
\vdash \forall P\, C\, Q. \\
\quad (\mathsf{PARTIALIZE}\ P)\ \mathsf{THEN}\ (\mathsf{WHILE}\ C\ (\mathsf{PARTIALIZE}\ Q)) \ = \\
\quad\ \ \mathsf{LOCVAR}\ \mathsf{F} \\
\quad\quad \mathsf{WHILE}\ (\lambda(x,h).\ C(x) \lor \neg h) \\
\quad\quad\ \ \mathsf{IFTE}\ (\lambda(x,h).\ h)\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (Q(x),h)))\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (P(x),\mathsf{T})))
\end{array} \tag{3}
$$

$$
\begin{array}{l}
\vdash \forall P\, C\, Q. \\
\quad (\mathsf{PARTIALIZE}\ P)\ \mathsf{THEN}\ (\mathsf{WHILE}\ C\ (\mathsf{PARTIALIZE}\ Q)) \ = \\
\quad\ \ \mathsf{LOCVAR}\ \mathsf{F} \\
\quad\quad \mathsf{WHILE}\ (\lambda(x,h).\ C(x) \lor \neg h) \\
\quad\quad\ \ \mathsf{IFTE}\ (\lambda(x,h).\ h)\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (x,h)))\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (P(x),\mathsf{T}))) \\
\quad\quad\ \ \mathsf{THEN} \\
\quad\quad\ \ \mathsf{IFTE}\ (\lambda(x,h).\ C(x))\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (Q(x),h)))\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (x,h)))
\end{array} \tag{4}
$$

The transformation theorems (5) and (6) are dedicated towards two nested while loops with a local variable which is to be shifted outwards. Again, there are basically two possible ways to transform the program. In both cases, a boolean local variable is introduced. It indicates, whether the inner loop is executed or not. The two programs again differ in timing and hardware consumption. Applying theorem (6) leads to a faster implementation with additional hardware since the inner loop-condition C_2 has to be implemented twice.

$$
\begin{array}{l}
\vdash \forall C_1\, C_2\, P\, \mathit{init}. \\
\quad \mathsf{WHILE}\ C_1\ (\mathsf{LOCVAR}\ \mathit{init}\ (\mathsf{WHILE}\ C_2\ (\mathsf{PARTIALIZE}\ P))) \ = \\
\quad\ \ \mathsf{LOCVAR}\ \mathit{init} \\
\quad\quad \mathsf{LOCVAR}\ \mathsf{F} \\
\quad\quad\ \ \mathsf{WHILE}\ (\lambda((x,h_1),h_2).\ C_1(x) \lor h_2) \\
\quad\quad\quad \mathsf{IFTE}\ (\lambda((x,h_1),h_2).\ C_2(x,h_1))\ \ (\mathsf{PARTIALIZE}\ (\lambda((x,h_1),h_2).\ (P(x,h_1),\mathsf{T}))) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \ (\mathsf{PARTIALIZE}\ (\lambda((x,h_1),h_2).\ ((x,\mathit{init}),\mathsf{F})))
\end{array} \tag{5}
$$

$$
\begin{array}{l}
\vdash \forall C_1\, C_2\, P\, init. \\
\quad \mathsf{WHILE}\, C_1\, (\mathsf{LOCVAR}\, init\, (\mathsf{WHILE}\, C_2\, (\mathsf{PARTIALIZE}\, P))) \;= \\
\quad\quad \mathsf{LOCVAR}\, init \\
\quad\quad\quad \mathsf{LOCVAR}\ \mathsf{F} \\
\quad\quad\quad\quad \mathsf{WHILE}\, (\lambda((x,h_1),h_2).\, C_1(x) \vee h_2) \\
\quad\quad\quad\quad\quad \mathsf{IFTE}\, (\lambda((x,h_1),h_2).\, C_2(x,h_1))\ (\mathsf{PARTIALIZE}\, (\lambda((x,h_1),h_2).\, (P(x,h_1),\mathsf{T}))) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\;\; (\mathsf{PARTIALIZE}\, (\lambda((x,h_1),h_2).\, ((x,h_1),h_2))) \\
\quad\quad\quad\quad\quad \mathsf{THEN} \\
\quad\quad\quad\quad\quad \mathsf{IFTE}\, (\lambda((x,h_1),h_2).\, C_2(x,h_1))\ (\mathsf{PARTIALIZE}\, (\lambda((x,h_1),h_2).\, ((x,h_1),h_2))) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\;\; (\mathsf{PARTIALIZE}\, (\lambda((x,h_1),h_2).\, ((x,init),\mathsf{F})))
\end{array}
\tag{6}
$$

Converting programs into a slf-representation can be executed bottom-up from the leaves of the syntax tree automatically by embedding heuristics that decide, whether to apply theorems that lead to faster or to hardware saving implementations. The embedding of non-formal methods is a general strategy of our formal synthesis approach HASH (see[KBES96]). Besides full automation, the theorems can also be selected interactively by the designer.

5.2 Further optimizations

The theorems are not optimal in a sense that for more complex transformation steps too much auxiliary variables are introduced. Given for instance two nested loops with a basic block at the beginning of the outer loop, a transformed program with only a single loop can be generated by first applying one of the theorems (3) or (4) to move the basic block into the inner loop. Afterwards, one of the theorems (5) or (6) must be applied to extract the local variable that was introduced by the theorem before. According to a desired timing behavior and hardware consumption, four compound transformations are possible leading to four different slf-programs. With this method, always two auxiliary variables are introduced. However, one can think of transformation theorems that introduce only a single variable. In fact, it is comparatively easy to prove that in each case one variable is unnecessary. There are two alternatives to handle this problem. Either more complex and compact transformation theorems are applied that can be derived using the already proven transformation theorems, or the hardware implementation is passed to state minimization.

When transforming programs to single-loop-form, scheduling is already performed by choosing the theorems and therefore deciding, whether some operations should be performed in one clock cycle or not. However it is also possible to execute the body of a while loop several times within one clock cycle (loop-unrolling) or to cut the body in pieces and execute the pieces in different clock cycles (loop-cutting). Loop cutting requires a scheduling information assigning each operation of a basic block to a phase. Dividing a basic block into several control steps according to results of existing heuristics has already been implemented in [EiBK96] and [BlEi97]. In [EiBK96], also performing allocation and binding has been introduced in a formal manner. However, these methods are restricted to pure basic blocks. We have derived general theorems for both loop-unrolling and loop-cutting (see [BlEi98]), which allows a more flexible synthesis process with respect to hardware consumption and timing also for μ-recursive functions.

6 Conclusion

In this paper, we have presented a new methodology for deriving RT-level structures from circuit descriptions at the algorithmic level. First it is formal. The implementation

correctness. Second it provides a new synthesis concept. The implementation was derived by applying program transformations rather than extracting a control and data flow graph and analyzing an exponential number of control paths. Together with our formal synthesis approach applied to RT- and gate level (see [EiKB97, Eise97b]) we thus provide a universally concept for formal synthesis from the algorithmic level down to the gate level.

Additionally, due to their functional representation, both the algorithmic description and the implementation at RT-level can be simulated efficiently. Therefore, the designer can check at every abstraction level, whether the implementation fulfills the specification he has in mind.

References

[BlEi97] C. Blumenröhr and D. Eisenbiegler. An efficient representation for formal synthesis. In *10th International Symposium on System Synthesis*, pages 9–15, Antwerp,Belgium, September 1997. IMEC, IEEE Computer Society Press.

[BlEi98] C.Blumenröhr and D. Eisenbiegler. Performing high-level synthesis via program transformations within a theorem prover. submitted to Fourth International Conference on Mathematics of Program Construction, June 1998.

[Camp89] R. Camposano. Behavior-preserving transformations for high-level synthesis. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, number 408 in Lecture Notes in Computer Science, pages 106–128, Ithaca, New York, July 1989. Mathematical Science Institute, Cornell University, Springer-Verlag.

[EiBK96] D. Eisenbiegler, C. Blumenröhr, and R. Kumar. Implementation issues about the embedding of existing high level synthesis algorithms in HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics:9th International Conference, TPHOLs'96*, number 1125 in Lecture Notes in Computer Science, pages 157–172, Turku,Finland, August 1996. Springer-Verlag.

[EiKB97] D. Eisenbiegler and R. Kumar and C. Blumenröhr . A constructive approach towards correctness of synthesis-application within retiming. In *The European Design & Test Conference*, pages 427–432, Paris, France, March 1997. IEEE Computer Society and ACM/SIGDA, IEEE Computer Society Press.

[Eise97b] D. Eisenbiegler. Automata — A theory dedicated towards formal circuit synthesis. Technical Report Internal report 14/97, Universität Karlsruhe, 1997. http: //goethe. ira. uka.de/fsynth/publications/postscript/Eise97b.ps.gz.

[EiSK93] D. Eisenbiegler, K. Schneider, and R. Kumar. A functional approach for formalizing regular hardware structures. In Thomas F. Melham and Juanito Camileri, editors, *Higher Order Logic Theorem Proving and its Applications*, number 859 in Lecture Notes in Computer Science, pages 101–114, Valletta, Malta, September 1994. Springer-Verlag.

[GDWL94] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis; Introduction to Chip and System Design.* Kluwer Academic Publishers, 1994.

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

[KBES96] R. Kumar , C. Blumenröhr, D. Eisenbiegler, and D. Schmid . Formal synthesis in circuit design-A classification and survey. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design. First International Conference,FMCAD'96*, number 1166 in Lecture Notes in Computer Science, pages 294–309, Palo Alto, CA, USA, November 1996. Springer-Verlag.

[MaFo91] E.M. Mayger and M.P. Fourman. Integration of formal methods with system design. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 59–70, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[MHMP96] P. Middelhoek, C.Huijs, G.Mekenkamp, and E.Prangsma et al. A methodology for the design of guaranteed correct and efficient digital systems. In *IEEE International High Level Design Validation and Test Workshop*, Oakland, California, November 1996.

[PeKu94] Z. Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(2):150–166, February 1994.

[ShRa95] R. Sharp and O. Rasmussen. The T-Ruby design system. In *CHDL '95*, pages 587–596, 1995.