Formal Synthesis at the Algorithmic Level*

Christian Blumenröhr and Viktor Sabelfeld

Institute for Circuit Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid)
University of Karlsruhe, Germany
{blumen,sabelfel}@ira.uka.de
http://goethe.ira.uka.de/fsynth

Abstract. In our terminology, the term "formal synthesis" stands for a synthesis process where the implementation is derived from the specification by applying elementary mathematical rules within a theorem prover. As a result the implementation is guaranteed to be correct. In this paper we introduce a new methodology to formally derive register-transfer structures from descriptions at the algorithmic level via program transformations. Some experimental results at the end of the paper show how the run-time complexity of the synthesis process in our approach could be.

1 Introduction

The synthesis of hardware systems is heading toward more and more abstract design levels. This is due to the fact that the systems are becoming more complex and so does the synthesis process for deriving them. Therefore, the correctness of hardware components has become an important matter — especially in safety-critical domains. By correctness we mean that the synthesis result (implementation) satisfies the synthesis input (specification), in a formal mathematical sense. It is assumed that the specifications are correct, which has to be examined separately, e.g., by model-checking certain properties or by simulation. For proving the correctness of implementations, simulation is no longer suitable, since it is normally (i.e. for large designs) not exhaustive in reasonable time. Formal post-synthesis verification [1] on the other hand needs manual interactions at higher abstraction levels; it can be automated at the gate level, but is extremely costly — and can only be applied, if some very simple synthesis steps have been performed. Therefore, it is our objective to perform synthesis via logical transformations and thus to guarantee "correctness by construction".

There are many approaches, that claim to fulfill this paradigm. When regarding the state of the art in this area, one can distinguish two concepts: *transformational design* and *formal synthesis*. In transformational design [2], the synthesis process is based on correctness-preserving transformations. However, in most cases a lot of intuition is used during the proofs [3]. Furthermore the proofs

* This work has been partly financed by the Deutsche Forschungsgemeinschaft, Project SCHM 623/6-2.

L. Pierre and T. Kropf (Eds.): CHARME'99, LNCS 1703, pp. 187–202, 1999.
© Springer-Verlag Berlin Heidelberg 1999

are often based on non-mathematical formalizations [4] and are performed in a paper&pencil style [5], which means that they have to be examined by others to verify them. However, the most restrictive fact in transformational design is that the implementations of the transformations are not proven to be correct. The transformations are realized by complex software programs, that might be error-prone. Therefore these approaches do not fulfill the above mentioned paradigm.

In formal synthesis approaches the synthesis process is performed within some logical calculus. The circuit descriptions are formalized in a mathematical manner and the transformations are based on some logical rules. The DDD system [6], e.g., starts from a specification in a Lisp-like syntax. The behavior is specified as an iterative system of tail-recursive functions. This is translated into a sequential description which can be regarded as a network of simultaneous signal definitions comprising variables, constants, delays and expressions involving operations. Then a series of transformations are applied to refine the description into an implementation. The disadvantage of this method is that the description language is not strongly typed. Therefore the consistency of an expression has to be checked separately. Furthermore, although all the transformations are based on functional algebra, their implementations have not been formally verified, nor are they based on a small core of elementary rules. Finally, the derivation process needs manual interactions. An automatic design space exploration method is not provided.

Our work is based on a functional hardware description language named Gropius, which ranges from the gate level to the system level. Gropius is strongly-typed, polymorphic and higher-order. Each construct of Gropius is defined within the higher-order logic theorem prover HOL [7] and since it is a subset of higher-order logic, Gropius has a mathematically exact semantics. This is the precondition for proving correctness. The implementation of HOL is not formally verified. However, since the implementation of the correctness-critical part of HOL — i.e. deriving new theorems — is very small and is independent of the size of our formal synthesis system, our approach can be considered to be extremely safe as to correctness. In the next section, we briefly introduce the way we represent circuit descriptions at the algorithmic level and give a small program as a running example.

Existing approaches in the area of formal synthesis deal with lower levels of abstraction (register-transfer (RT) level, gate level) [8,9,6,10,11] or with pure dataflow graphs at the algorithmic level [12]. This paper addresses formal synthesis at the algorithmic level. The approach goes beyond pure basic blocks and allows synthesizing arbitrary computable, i.e. μ-recursive programs.

The starting point for high-level synthesis (HLS) is an algorithmic description. The result is a structure at the RT level. Usually, hardware at the RT-level consists of a data-path and a controller. In conventional approaches [13], first, all loops in a given control/dataflow graph (CDFG) are cut, thus introducing several acyclic program pieces each corresponding to one clock tick. The number of these cycle-free pieces hereby grows exponentially with the size of the CDFG. Afterwards scheduling, allocation and binding are performed separately on these

parts leading to a data-path and a state transition table. Finally, the controller and the communication part are generated.

We have developed a methodology that absolutely differs from this standard. In our approach, the synthesis process is not reduced to the synthesis of pure dataflow graphs, but the circuit description always remains compact and the RT-level structure is derived via program transformations. Besides an RT-level structure, our approach additionally delivers an accompanying proof in terms of a theorem telling that this implementation is correct. High-level synthesis is performed in four steps. In the first two steps which are explained in Section 3, scheduling and register allocation/binding are performed. Based on pre-proven program equations which can be steered by external design exploration techniques, the program is first transformed into an equivalent but optimized program, and then this program is transformed into an equivalent program with a single while-loop. The third step (Section 4) performs interface synthesis. An interface behavior can be selected and the program is mapped by means of a pre-proven implementation theorem to a RT-level structure, that realizes the interface behavior with respect to the program. In the last step, which is not addressed explicitely here, functional units are allocated and bound. Section 5 will give some experimental results.

2 Formal Representation of Programs

At the algorithmic level, behavioral descriptions are represented as pure software programs. The concrete timing of the circuit, that has to be synthesized, is not yet considered. In Gropius, we distinguish between two different algorithmic descriptions. DFG-terms represent non-recursive programs that always terminate (Data Flow Graphs). They have some type $\alpha \rightarrow \beta$. P-terms are means for representing arbitrary computable functions (Programs). Since P-terms may not terminate, we have added an explicit value to represent nontermination: a P-term either has the value $\mathsf{Defined}\,(x)$ indicating that the function application terminates with result x, or in case of nontermination the value is Undefined. The type of P-terms is expressed by $\alpha \rightarrow (\beta)\mathsf{partial}$.

In our approach, P-terms are used for representing entire programs as well as blocks. Blocks are used for representing inner pieces of programs. In contrast to programs, the input type equals the output type. This is necessary for loops which apply some function iteratively. In Gropius, there is a small core of 8 basic control structures for building arbitrary computable blocks and programs based on basic blocks and conditions. Basic blocks (type $\alpha \rightarrow \alpha$) and conditions (type $\alpha \rightarrow \mathsf{bool}$) itself are represented by DFG-terms. In Table 1, only those control structures are explained that are used in this paper. Based on this core of control structures further control structures like for- and repeat-loops can be derived by the designer.

In the rest of the paper a specific pattern called single-loop form (SLF) plays an important role. Programs in SLF have the following shape:

$$\mathsf{PROGRAM}\ \mathit{out\_init}\ (\mathsf{LOCVAR}\ \mathit{var\_init}\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a))) \qquad (1)$$

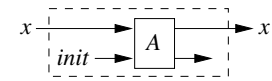
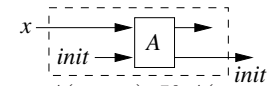
PARTIALIZE	Type: $(\alpha \to \alpha) \to \alpha \to (\alpha)\mathsf{partial}$. Turns a basic block a to a block (PARTIALIZE a). Since basic blocks always terminate, (PARTIALIZE a) maps some x to $\mathsf{Defined}(a(x))$. Undefined is never reached.
WHILE	Type: $(\alpha \to \mathsf{bool}) \to (\alpha \to (\alpha)\mathsf{partial}) \to \alpha \to (\alpha)\mathsf{partial}$. Basis for formalizing true μ–recursion. Given a block A and a condition c, (WHILE $c\,A$) maps some parameter x to some value $\mathsf{Defined}(y)$ by iterating A until the value y is reached with $\neg(c\ y)$. In case of nontermination the result becomes Undefined.
THEN	Type: $(\alpha \to (\alpha)\mathsf{partial}) \to (\alpha \to (\alpha)\mathsf{partial}) \to \alpha \to (\alpha)\mathsf{partial}$. Binary function used in infix notation. Two blocks A and B are executed consecutively. The result of (A THEN B) becomes Undefined, iff one of the two blocks does not terminate.
LOCVAR	Type: $\beta \to ((\alpha \times \beta) \to (\alpha \times \beta)\mathsf{partial}) \to \alpha \to (\alpha)\mathsf{partial}$. Introduce a local variable. Given an arbitrary initial value *init* for the local variable, the function (LOCVAR *init* A) first maps some input x to $A(x, init)$. If the result becomes $\mathsf{Defined}(x', init')$, then $\mathsf{Defined}(x')$ is returned. In case of nontermination Undefined is returned. 
PROGRAM	Type: $\beta \to ((\alpha \times \beta) \to (\alpha \times \beta)\mathsf{partial}) \to \alpha \to (\beta)\mathsf{partial}$. Turns a block into a program. Given an arbitrary initial value *init* for the output variable, the function (PROGRAM *init* A) first maps some input x to $A(x, init)$. If $A(x, init)$ terminates with $\mathsf{Defined}(x', init')$, then $\mathsf{Defined}(init')$ is returned. In case of nontermination Undefined is returned. 

Table 1. Core control structures

The expressions *out_init* and *var_init* denote arbitrary constants, c is an arbitrary condition and a an arbitrary basic block.

Basically, no front-end is required, since Gropius is both the input language and the intermediate format for transforming the circuit description. However, since the "average designer" may not be willing to specify in a mathematical notation, it would also be possible to automatically translate Gropius-descriptions from other languages like Pascal. But on the other hand this adds an error-prone part into the synthesis process you can abandon, since Gropius is easy to learn — there are only few syntax rules.

Fig. 1 shows a program in an imperative representation style (Pascal) and a corresponding description in Gropius. The program computes the n^{th} Fibonacci number and uses a fast algorithm which has a complexity of $O(\log_2 n)$.

The program fib has type $\mathsf{num} \to (\mathsf{num})\mathsf{partial}$ (num is the type of natural numbers). The input variable is n. The construct (PROGRAM1) introduces a variable (here called $y1$) which indicates the output of the program and hence

Imperative Program	Representation in Gropius
FUNCTION FIB	val fib =
(var n : int) : int;	PROGRAM 1
VAR $a1, a2, y1, y2, m$: int;	LOCVAR $(1, 1, 0, 0)$
VAR r, s : int;	PARTIALIZE $(\lambda((n, y1), a1, a2, y2, m).$
BEGIN	$((n, y1), a1, a2, y2, (n$ DIV $2) + 1))$
$a1 := 1;\ a2 := 1;$	THEN
$y1 := 1;$	WHILE $(\lambda((n, y1), a1, a2, y2, m).\neg(m = 0))$
$y2 := 0;$	PARTIALIZE $(\lambda((n, y1), a1, a2, y2, m).$
$m := n$ div $2 + 1;$	let $c =$ ODD m in
WHILE $m <> 0$ DO	let $m1 = m - 1$ in
IF odd m	let $m2 = m$ DIV 2 in
THEN BEGIN	let $m3 =$ MUX$(c, m1, m2)$ in
$r := y1;$	let $x = a1 + a2$ in
$y1 := y1 * a1 + y2 * a2;$	let $x1 =$ MUX$(c, y1, a1)$ in
$y2 := r * a2 + y2 * (a1 + a2);$	let $x2 =$ MUX$(c, y2, a2)$ in
$m := m - 1$	let $x3 = x1 * a1$ in
END	let $x4 = x2 * a2$ in
ELSE BEGIN	let $x5 = x3 + x4$ in
$s := a1;$	let $x6 = x1 * a2$ in
$a1 := a1 * a1 + a2 * a2;$	let $x7 = x2 * x$ in
$a2 := s * a2 + a2 * (s + a2);$	let $x8 = x6 + x7$ in
$m := m$ div 2	let $y1' =$ MUX$(c, x5, y1)$ in
END;	let $a1' =$ MUX$(c, a1, x5)$ in
IF odd n	let $a2' =$ MUX$(c, a2, x8)$ in
THEN	let $y2' =$ MUX$(c, x8, y2)$ in
RETURN $y2$	$((n, y1'), a1', a2', y2', m3))$
ELSE	THEN
RETURN $y1$	PARTIALIZE $(\lambda((n, y1), a1, a2, y2, m).$
END;	$((n,$ MUX(ODD $n, y2, y1)), a1, a2, y2, m))$

Fig. 1. Program for calculating the n^{th} Fibonacci number

the n^{th} Fibonacci number. A special "return"-statement is not necessary since the result is stored in $y1$. The initial value of the output is the same as for the local variable $y1$ in the imperative program. (LOCVAR$(1, 1, 0, 0)$) is used to introduce local variables with initial values 1, 1, 0 and 0, respectively. These local variables correspond to the local variables $a1, a2, y2$ and m in the imperative program. DFG-terms are formalized using λ-expressions [14]. An expression $(\lambda x.a[x])$ denotes a function, which maps the value of a variable x to the expression $a[x]$, which has some free occurrences of x. Two basic laws of the λ-calculus are β-conversion and η-conversion:

$$(\lambda x.a\,[x])\ b \xrightarrow{\beta} a[b/x] \qquad (\lambda x.a\ x) \xrightarrow{\eta} a$$

let-terms (let $x = y$ in z) are a syntactic variant of β-redices $(\lambda x.z)\, y$. By applying β-conversion some or all of the let-terms can be eliminated. In contrast to LOCVAR, let-terms introduce local variables only within basic blocks. The expression MUX is an abbreviation for a conditional expression within basic blocks. $\mathsf{MUX}(c, a, b)$ returns a, if the condition c is true, otherwise it returns b.

The correspondence between the two descriptions in Fig. 1 is not one-to-one. To yield a more efficient description with less addition and multiplication operations, the following two theorems for conditionals have been applied:

$$\vdash f\ \mathsf{MUX}(c, a, b) = \mathsf{MUX}(c, f\ a, f\ b) \quad \vdash \mathsf{MUX}(c, a, b)\ g = \mathsf{MUX}(c, a\ g, b\ g) \quad (2)$$

The imperative program can first be translated into a Gropius description containing the same eight multiplications and six additions in the loop-body. Then the theorems (2) can be applied to generate the description shown in Fig. 1, which needs only four multiplications and three additions in the loop-body.

3 Program Transformations

The basic idea of our formal synthesis concept is to transform a given program into an equivalent one, which is given in SLF. This is motivated by the fact that hardware implementations are nothing but a single while-loop, always executing the same basic block. Every program can be transformed into an equivalent SLF-program (Kleene's normal form of μ-recursive functions). However for a given program there might not be an unique SLF, but there are infinitely many equivalent SLF-programs. In the loop-body of a SLF-program, all operations of the originally given program are scheduled. The loop-body behaves like a case-statement, in which within a single execution certain operations are performed according to the control state indicated by the local variables (LOCVAR *var_init*). After mapping the SLF-program to a RT-level structure (see Section 4), every execution of the loop-body corresponds to a single control step. The cost of the RT-implementation therefore depends on which operations are performed in which control step. Thus every SLF corresponds to a RT-implementation with certain costs. Performing high-level synthesis therefore requires to transform the program into a SLF-program, that corresponds to a cost-minimal implementation.

In the HOL theorem prover we proved several program transformation theorems which can be subdivided into two groups. The first group consists of 27 theorems. One can prove that these theorems are sufficient to transform every program into an equivalent program in SLF. The application of these theorems is called the standard-program-transformation (SPT). During the SPT, control structures are removed and instead auxiliary variables are introduced holding the control information. Theorem (3) is an example:

$$\begin{array}{l}\vdash \mathsf{WHILE}\, c_1\, (\mathsf{LOCVAR}\, v\, (\mathsf{WHILE}\, c_2\, (\mathsf{PARTIALIZE}\, a))) \;=\\ \quad \mathsf{LOCVAR}\, (v, \mathsf{F})\\ \qquad \mathsf{WHILE}\, (\lambda(x, h_1, h_2).\, c_1\, x \;\vee\; h_2)\\ \qquad\quad \mathsf{PARTIALIZE}\, (\lambda(x, h_1, h_2).\mathsf{MUX}\, (c_2\, (x, h_1), (a\, (x, h_1), \mathsf{T}), (x, v, \mathsf{F})))\end{array} \tag{3}$$

Two nested while-loops with a local variable at the beginning of the outer loop-body are transformed to a single while-loop. The local variable is now outside the loop and there is an additional local variable with initial value F. This variable holds the control information, whether the inner while-loop is performed (value is T) or not (value is F).

Although the SLF representation is not unique, the SPT always leads to the same SLF for a given program by scheduling the operations in a fixed way. Therefore, the SPT unambiguously assigns costs to every program. To produce other, equivalent SLF representations, which result in another scheduling and thus in other costs for the implementation, the theorems of the second group have to be applied before performing the SPT. Currently, we proved 19 optimization-program-transformation (OPT) theorems. These OPT-theorems can be selected manually, but it is also possible to integrate existing design space exploration techniques which steer the application of the OPT-theorems. The OPT-theorems realize transformations which are known from the optimization of compilers in the software domain [15]. Two of these transformations are loop-unrolling and loop-cutting.

Loop unrolling reduces the execution time since several operations are performed in the same control step. On the other hand, it increases the combinatorial depth and therefore the amount of hardware. Theorem (4) shows the loop unrolling theorem. It describes the equivalence between a while-loop and an n-fold unrolled while-loop with several loop-bodies which are executed successively. Between two loop-bodies, the loop-condition is checked to guarantee that the second body is only executed if the value of the condition is still true. FOR_N is a function derived from the core of Gropius. (FOR_N n A) realizes an n-fold application of the same block A. Theorem (5) can be used to remove FOR_N after having instantiated n (SUC is the successor function).

$$\begin{array}{l}\vdash \mathsf{WHILE}\, c\, (\mathsf{PARTIALIZE}\, a) \;=\\ \quad \mathsf{WHILE}\, c\, \big((\mathsf{PARTIALIZE}\, a)\ \mathsf{THEN}\\ \qquad\qquad (\mathsf{FOR\_N}\, n\, (\mathsf{PARTIALIZE}\, (\lambda x.\, \mathsf{MUX}\, (c\, x, a\, x, x)))))\big)\end{array} \tag{4}$$

$$\vdash (\mathsf{FOR\_N}\ 1\ A \;=\; A) \;\wedge\; (\mathsf{FOR\_N}\ (\mathsf{SUC}\ n)\ A \;=\; A\ \mathsf{THEN}\ (\mathsf{FOR\_N}\ n\ A)) \tag{5}$$

The counterpart to loop unrolling is the loop cutting: the loop is cut into several smaller parts. Each part then corresponds to a separate control step. This results in a longer execution time; however, the hardware consumption might be reduced, if the parts can share function units.

$$\begin{array}{l}\vdash \mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ (\mathsf{list\_o}\ (k :: r))) \ = \\ \quad \mathsf{LOCVAR}\ (\mathsf{enum}\ (\mathsf{SUC}\ (\mathsf{LENGTH}\ r))\ 0) \\ \qquad \mathsf{WHILE}\ (\lambda(x,h).\, c\ x\ \vee\ \neg(h\ =\ 0)) \\ \qquad\quad \mathsf{PARTIALIZE}\ (\lambda(x,h).\, ((\mathsf{CASE}\ (k :: r)\ h)\ x,\ \mathsf{next}\ (\mathsf{SUC}\ (\mathsf{LENGTH}\ r))\ h)\ )\end{array} \tag{6}$$

In (6) the loop cutting theorem is shown. It assumes that the body of a while-loop has been scheduled into a composition of functions. The term (list_o L) denotes a composition of functions given by the list L. By restricting L to be of the form $(k :: r)$ it is guaranteed that the list is not empty. Each function of the list L must have the same type $\alpha \to \alpha$. The input and output types must be equal, since loop-bodies are executed iteratively. Given a while-loop with such a scheduled loop-body (list_o $(k :: r)$), theorem (6) turns it into an equivalent while-loop, which executes its body (LENGTH r)[1] times more often than the original while-loop. Within one execution of the loop-body exactly one of the functions of the list $(k :: r)$ is applied. The control information, which function is to be performed is stored in a local variable that has been introduced. This variable has an enumeration datatype. Its initial value is 0 and its value ranges from 0 to (LENGTH r). The semantics of enum is shown in (7). If the local variable has value 0, the loop-condition c is checked, whether to perform the loop-body or not. If the local variable's value differs from 0, the loop-body will be executed independent of c. (CASE L i) picks the i^{th} function of the list L. Therefore, within one execution of the loop-body, a function of the list $(k :: r)$ is selected according to the value h of the local variable and then this function is applied to the value x of the global input variable. Furthermore, the new value of the local variable is determined. The semantics of next is shown in (7).

$$\vdash \mathsf{enum}\, n\, m \ =\ \mathsf{MUX}(m < n, m, 0) \quad \vdash \mathsf{next}\, n\, x \ =\ \mathsf{MUX}(\mathsf{SUC}\ x \ <\ n, \mathsf{SUC}\ x, 0) \tag{7}$$

Returning to our example program fib, the body of the while-loop can be scheduled in many ways. The decision on how the body should be scheduled can be made outside the logic by incorporating existing scheduling techniques for data-paths. Table 2 shows the results of applying the ASAP (as-soon-as-possible), force-directed [16] and list-based scheduling techniques to our example program. The ASAP algorithm delivers the minimal number of control steps. In addition to this, the force-directed-algorithm tries to minimize the amount of hardware. The list-based scheduling on the other hand restricts the amount of hardware components and tries to minimize the execution time. For each control step, we list the local variables of the loop from Fig. 1 that hold the result of the corresponding operations. Note that no chaining was allowed in the implementation of these scheduling programs. However, this is not a general restriction. For performing the list-based scheduling, the number of multiplications and additions was each restricted to two.

In the next step, the number and types of the registers will be determined that have to be allocated. This is also listed in Table 2. Before actually scheduling

[1] LENGTH L returns the length of a list L. Instantiating the theorem with a concrete list, yields a concrete value for this expression. Similarly, (list_o L) then corresponds to a concrete term.

Control step	ASAP	Force-directed	List-based
1	$c, m1, m2, x$	c	$c, m1, m2, x$
2	$m3, x1, x2$	$m1, m2, x, x1, x2$	$m3, x1, x2$
3	$x3, x4, x6, x7$	$x3, x4, x6, x7$	$x3, x4$
4	$x5, x8$	$m3, x5, x8$	$x5, x6, x7$
5	$y1', a1', a2', y2'$	$y1', a1', a2', y2'$	$x8, y1', a1'$
6	$- - -$	$- - -$	$a2', y2'$
Allocated registers	Type bool : 1 Type num : 10	Type bool : 1 Type num : 10	Type bool : 1 Type num : 11

Table 2. Control information extracted by different scheduling algorithms

the loop-body by a logical transformation, additional input variables have to be introduced, since the number of input and output variables directly corresponds to the number of registers necessary at the RT-level. The number of additional variables is $(\#regalloc - \#invars)$ with $\#invars$ being the number of input variables of the loop-body and $\#regalloc$ being the number of allocated registers. For our example fib in the case of allocation after the ASAP scheduling, this value is $11 - 6 = 5$. Therefore 5 additional input variables for the loop have to be introduced. This is done by theorem (8). Applying it to the loop in Fig. 1 with appropriately instantiating i gives program (9).

$$\vdash \forall i.\, \mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a) = \mathsf{LOCVAR}\ i\ (\mathsf{WHILE}\ (\lambda(x,h).cx)\ (\mathsf{PARTIALIZE}(\lambda(x,h).(ax,i)))) \tag{8}$$

$$\begin{array}{l} \mathsf{LOCVAR}(\mathsf{F},0,0,0,0) \\ \quad \mathsf{WHILE}(\lambda(((n,y1),a1,a2,y2,m),h1,h2,h3,h4,h5).\neg(m=0)) \\ \quad\quad \mathsf{PARTIALIZE}(\lambda(((n,y1),a1,a2,y2,m),h1,h2,h3,h4,h5). \\ \quad\quad\quad \mathsf{let}\ c = \mathsf{ODD}\ m\ \mathsf{in}\quad \ldots\quad \mathsf{in}\ (((n,y1'),a1',a2',y2',m3),\mathsf{F},0,0,0,0)) \end{array} \tag{9}$$

The additional variables are only dummies for the following scheduling and register allocation/binding. They must not be used within the loop-body. Since the original input variables are all of type num, one variable of type bool ($h1$) and four variables of type num ($h2, \ldots, h5$) are introduced. Some default initial values are used for each type. Since the output type must equal the input type, additional outputs have to be introduced as well.

Now the DFG-term representing the loop-body can be scheduled and register binding can be performed, both by logical conversions within HOL. Fig. 2 shows the resulting theorem after this conversion. The equivalence between the original and the scheduled DFG-term is proven by normalizing the terms, i.e. performing β-conversions on all β-redices [19]. The register binding was performed based on the result of a heuristic that tries to keep a variable in the same register as long as possible to avoid unnecessary register transfer. The right hand side of the theorem in Fig. 2 is actually an expression of the form (list_o L) with L being a list of five DFG-terms. The theorem in Fig. 2 can be used to transform the circuit description fib by rewriting and afterwards the loop-cutting theorem (6) can be applied.

$$
\begin{array}{l}
\vdash \lambda(((n,y1),a1,a2,y2,m),h1,h2,h3,h4,h5). \\
\quad \mathsf{let}\ c = \mathsf{ODD}\ m\ \mathsf{in} \quad \ldots \quad \mathsf{in} \quad (((n,y1'),a1',a2',y2',m3),\mathsf{F},0,0,0,0) \\
= \\
\lambda(((r1,r2),r3,r4,r5,r6),r7,r8,r9,r10,r11). \\
\quad \mathsf{let}\ y1' = \mathsf{MUX}(r7,r2,r6)\ \mathsf{in}\ \mathsf{let}\ a1' = \mathsf{MUX}(r7,r1,r2)\ \mathsf{in} \\
\quad \mathsf{let}\ a2' = \mathsf{MUX}(r7,r8,r3)\ \mathsf{in}\ \mathsf{let}\ y2' = \mathsf{MUX}(r7,r3,r9)\ \mathsf{in} \\
\quad (((r5,y1'),a1',a2',y2',r4),\mathsf{F},0,0,0,0) \\
\circ \\
\lambda(((r1,r2),r3,r4,r5,r6),r7,r8,r9,r10,r11). \\
\quad \mathsf{let}\ r2' = r11 + r10\ \mathsf{in}\ \mathsf{let}\ r3' = r3 + r2\ \mathsf{in} \\
\quad (((r1,r2)',r3',r4,r5,r6),r7,r8,r9,r10,r11) \\
\circ \\
\lambda(((r1,r2),r3,r4,r5,r6),r7,r8,r9,r10,r11). \\
\quad \mathsf{let}\ r11' = r2 * r1\ \mathsf{in}\ \mathsf{let}\ r10' = r10 * r8\ \mathsf{in}\ \mathsf{let}\ r3' = r2 * r8\ \mathsf{in} \\
\quad \mathsf{let}\ r2' = r10 * r3\ \mathsf{in}\ (((r1,r2'),r3',r4,r5,r6),r7,r8,r9,r10',r11') \\
\circ \\
\lambda(((r1,r2),r3,r4,r5,r6),r7,r8,r9,r10,r11). \\
\quad \mathsf{let}\ r4' = \mathsf{MUX}(r7,r2,r4)\ \mathsf{in}\ \mathsf{let}\ r2' = \mathsf{MUX}(r7,r6,r1)\ \mathsf{in} \\
\quad \mathsf{let}\ r10' = \mathsf{MUX}(r7,r9,r8)\ \mathsf{in}\ (((r1,r2'),r3,r4',r5,r6),r7,r8,r9,r10',r11) \\
\circ \\
\lambda(((n,y1),a1,a2,y2,m),h1,h2,h3,h4,h5). \\
\quad \mathsf{let}\ r7' = \mathsf{ODD}\ m\ \mathsf{in}\ \mathsf{let}\ r2' = m - 1\ \mathsf{in}\ \mathsf{let}\ r4' = m\ \mathsf{DIV}\ 2\ \mathsf{in} \\
\quad \mathsf{let}\ r3' = a1 + a2\ \mathsf{in}\ (((a1,r2'),r3',r4',n,y1),r7',a2,y2,r10,r11)
\end{array}
$$

Fig. 2. Theorem after scheduling and register allocation/binding

Besides loop-unrolling and loop-cutting, several other OPT-theorems can be applied. After that the SPT is performed generating a specific program in SLF. Fig. 3 shows the theorem after performing the SPT without applying any OPT-theorem before. When the program in SLF is generated without any OPT, then the operations in the three blocks before, within and after the while-loop in the original program fib, will be performed in separate executions of the resulting loop-body. Therefore, function units can be shared among these three blocks. Although ,e.g., two DIV-operations appear in Fig. 3, only one divider is necessary. One division comes from the block before the loop and the other results from the loop-body. These two division-operations are therefore needed in different executions of the new loop-body. They can be shifted behind the multiplexer by using one of the theorems (2). The allocation and binding of functional units is the last step in our high-level synthesis scenario. Before this, interface synthesis must be applied.

4 Interface Synthesis

At the algorithmic level, circuit representations consist of two parts. An algorithmic part describes the functional relationship between input and output. Time

$$
\begin{array}{l}
\vdash \mathsf{fib} = \\
\quad \mathsf{PROGRAM1} \\
\quad\ \mathsf{LOCVAR}((1,1,0,0),\mathsf{F},\mathsf{F}) \\
\quad\quad \mathsf{WHILE}(\lambda(((n,y1),a1,a2,y2,m),h1,h2).\neg h1 \vee \neg h2) \\
\quad\quad\ \mathsf{PARTIALIZE}(\lambda(((n,y1),a1,a2,y2,m),h1,h2). \\
\qquad\quad \mathsf{let}\ x1' = \neg(m=0)\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ c = \mathsf{ODD}\ m\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x1'' = \mathsf{MUX}(c,y1,a1)\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x2 = \mathsf{MUX}(c,y2,a2)\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x5 = x1'' * a1 + x2 * a2\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x8 = x1'' * a2 + x2 * (a1 + a2)\ \mathsf{in} \\
\qquad\quad (\,((n, \mathsf{MUX}(h2, \mathsf{MUX}(x1', \mathsf{MUX}(c,x5,y1), \mathsf{MUX}(\mathsf{ODD}\ n, y2, y1)), y1)), \\
\qquad\quad\ \mathsf{MUX}(h2 \wedge x1', \mathsf{MUX}(c,a1,x5), a1), \\
\qquad\quad\ \mathsf{MUX}(h2 \wedge x1', \mathsf{MUX}(c,a2,x8), a2), \\
\qquad\quad\ \mathsf{MUX}(h2 \wedge x1' \wedge c, x8, y2), \\
\qquad\quad\ \mathsf{MUX}(h2, \mathsf{MUX}(x1', \mathsf{MUX}(c, m-1, m\ \mathsf{DIV}\ 2), m), (n\ \mathsf{DIV}\ 2) + 1)), \\
\qquad\quad\ \mathsf{MUX}(h2, \neg x1', h1), \mathsf{T}\ )\ )
\end{array}
$$

Fig. 3. Single-loop-form of fib

is not yet considered. During high-level synthesis the algorithmic description is mapped to a RT-level structure. To bridge the gap between these two different abstraction levels one has to determine how the circuit communicates with its environment. Therefore, as second component of the circuit representation, an interface description is required.

In contrast to most existing approaches, we strictly separate between the algorithmic and the interface description. We provide a set of at the moment nine interface patterns, of which the designer can select one. Some of these patterns are used for synthesis of P-terms and others for synthesis of DFG-terms. The orthogonal treatment of functional and temporal aspects supports reuse of designs in a systematic manner, since the designer can use the same algorithmic description in combination with different interface patterns.

Remark: At the algorithmic level we only consider single processes that do not communicate with other processes. In addition to this, we have developed an approach for formal synthesis at the system level, where several processes interact with each other [17].

Fig. 4 shows the formal definition of two of those interface patterns. Beside the data signals of an algorithmic description P, the interface descriptions contain additional control signals which are used to steer the communication, to stop or to start the execution of the algorithm.

The two patterns are functions which map an arbitrary program P and the signals $(in, start, out, ready)$ and $(in, reset, out, ready)$, respectively, to a relation between these signals with respect to the program P. The pattern P_IFC_START states that at the beginning the process is idle, if the *start*-signal is not active. As long as the process is idle and no calculation is started, the

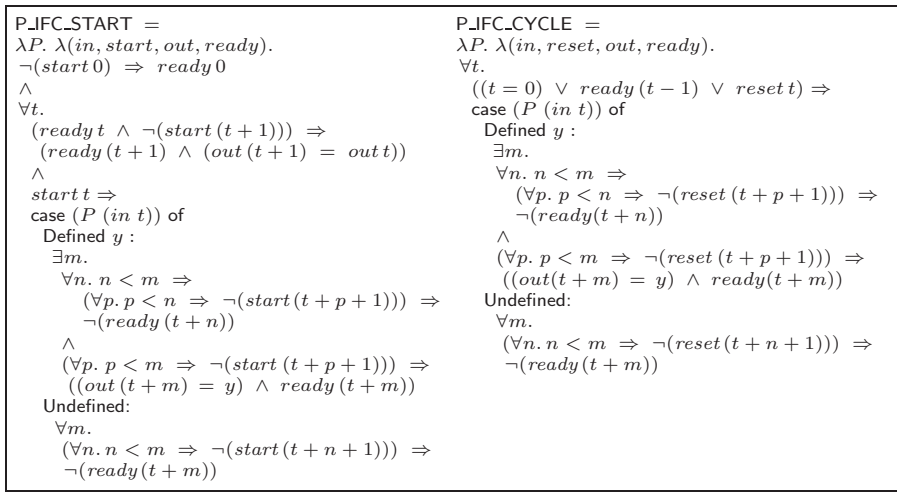


Fig. 4. Formal definition of two interface patterns

process will be idle in the next clock tick and *out* holds its last value. If *start* is active, i.e. a new calculation begins, two alternatives may occur:

- the calculation $P\,(in\ t)$ will terminate after some time steps m. As long as the calculation is performed, the process is active, i.e. *ready* is F. When the calculation is finished at time step $t + m$, *out* holds the result y and *ready* is T, indicating that the process is idle again. However, the calculation can only be finished, if *start* is not set to T while the calculation is performed.
- the calculation $P\,(in\ t)$ will not terminate. Then the process will be active producing no result until a new calculation is started by setting *start* to T.

The pattern P_IFC_CYCLE describes a process, which always performs a calculation and starts a new one as soon as the old one has been finished. The *reset*-signal can be used here to stop a (non-terminating) calculation and to start a new one.

For each interface pattern that we provide, we also have proven a correct implementation theorem. All the implementation theorems corresponding to patterns for P-terms expect the programs to be in SLF. The formal Gropius-descriptions of implementations at the RT-level can be found in [18]. In (10), an implementation theorem is shown, stating that an implementation pattern called IMP_START fulfills the interface pattern P_IFC_START for each program being in SLF (see also the pattern of the SLF in (1)).

$$
\begin{aligned}
&\vdash \forall a\ c\ out\_init\ var\_init.\\
&\quad \mathsf{IMP\_START}\ (a, c, out\_init, var\_init)\ (in, start, out, ready)\\
&\quad \Rightarrow\\
&\quad \mathsf{P\_IFC\_START}\\
&\qquad \big(\mathsf{PROGRAM}\ out\_init\ (\mathsf{LOCVAR}\ var\_init\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a)))\big)\\
&\qquad (in, start, out, ready)
\end{aligned}
\tag{10}
$$

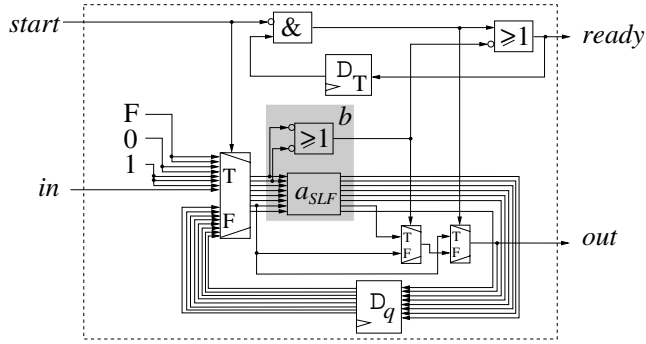


Fig. 5. RT implementation IMP_START for the program fib

$$\vdash \mathsf{IMP\_START}\ (a_{SLF}, c_{SLF}, out\_init_{SLF}, var\_init_{SLF})\ (in, start, out, ready) \Rightarrow \mathsf{P\_IFC\_START}\ \mathsf{fib}\ (in, start, out, ready) \tag{11}$$

The final theorem (11) is achieved by first instantiating the universal quantified variables in theorem (10) with the components $a_{SLF}, c_{SLF}, out\_init_{SLF}$ and var_init_{SLF} of the SLF in Fig. 3. Afterwards, the SLF-theorem $\vdash \mathsf{fib} = \mathsf{fib}_{SLF}$ in Fig. 3 is turned to $\vdash \mathsf{fib}_{SLF} = \mathsf{fib}$ (symmetry of equivalence) and rewriting is performed with this theorem. Theorem (11) says that the implementation sketched in Fig. 5[2] satisfies the program fib from Fig. 1 with respect to the interface pattern P_IFC_START.

The last step in our high-level synthesis scenario is allocation and binding of functional units (FU). This is done within the DFG-term b (grey area in Fig. 5). When synthesizing fib without any OPT, four multipliers, three adders and one divider have to be allocated besides some boolean function units. For sake of space, the logical transformation for this task is not shown here. When OPT-theorems like loop-unrolling or loop-cutting are applied, the amount of allocated hardware may be different. For more information on FU allocation/binding see also [19], where the synthesis of pure DFG-terms is described. Since the allocation and binding of functional units is performed within the DFG-term b, the method can be applied which is described there.

5 Experimental Results

Our formal synthesis approach consists of four steps. OPT and SPT for scheduling and register allocation/binding, applying an implementation theorem for interface synthesis and allocation/binding of functional units within the resulting basic block of the RT-implementation. SPT and interface synthesis consist of rewriting and β-conversions, which can be done fully automatically within the

[2] Due to lack of space the loop-body of the SLF (component a_{SLF}) is not explicitely shown in Fig. 5. q denotes an arbitrary initial value in the eight allocated registers.

HOL theorem prover. For the OPT and the FU-allocation/binding, however, heuristics are needed to explore the design space. Those non-formal methods can be integrated in the formal synthesis process, since the design space exploration part is separated from the transformation within the theorem prover. After establishing the formal basis for our synthesis approach, it is our objective in the future to develop further heuristics and to integrate more existing techniques for the OPT. An interesting approach is proposed in [20] where a method is described in which the design space is explored for performing similar transformations that are used in the OPT of our synthesis process.

To give an impression about the costs in formal synthesis, Fig. 6 shows the run-times (on SUN UltraCreator, Solaris 5.5.1, 196 MB main memory) of several programs for both performing the SPT and instantiating an implementation theorem. The descriptions of the programs in the programming language C can be found in [21]. Since we did not perform any OPT, it does not make sense to compare the results with other approaches with respect to the number of registers and FUs. As we have demonstrated on our small running example, very different implementations can be achieved if OPT-theorems are applied. Since only few heuristics for automatic invoking the OPT-theorems have been implemented, we have considered only the SPT and the interface synthesis. The cost for the SPT mainly increases with the number of the control structures but also with the number of operations in the program. The cost for the interface synthesis mainly increases with the size of the loop-body of the SLF-program.

The experiments have been run using a slight variant of the HOL theorem prover. As compared to the original HOL system, it has been made more efficient by changing the term representation and adding two core functions. See [22] for a detailed description and discussion about this. As we have demonstrated in the paper, the result of our synthesis process is a guaranteed correct implementation. A proof is given together with the implementation, stating that the implementation fulfills the specification. Therefore, one should be aware that the run-times must be compared with conventional synthesis plus exhaustive simulation. Furthermore, we believe that due to the complexity it is very hard (or even impossible) to develop automatic post-synthesis verification methods at this abstraction level which could prove the correctness of the synthesis process.

Program	SPT	Interface synthesis
fibonacci	2.1	0.1
gcd	1.5	0.2
bubble	4.4	0.5
fuzzy	19.0	1.2
kalman	103.9	2.8
diffeq	3.4	0.3
fidelity	5.1	0.6
dct	50.0	5.2
atoi	1.9	0.2

Fig. 6. Time [s] for synthesis experiments

6 Conclusion

In this paper, we presented a formal way for performing high-level synthesis. The main contribution is that we perform the whole synthesis process within a theorem prover. The result is therefore not only an implementation but also an

accompanying proof that this implementation is correct. Furthermore, we developed a new synthesis method, where the implementation is derived by applying program transformations instead of generating and analyzing a number of control paths that grows exponentially with the CDFG size. Last but not least we orthogonalize the treatment of algorithmic and temporal aspects and therefore support a systematic reuse of designs.

References

1. A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, 1992. 187
2. P. Middelhoek. *Transformational Design*. PhD thesis, Univ. Twente, NL, 1997. 187
3. M. McFarland. Formal analysis of correctness of behavioral transformations. *Formal Methods in System Design*, 2(3), 1993. 187
4. Z. Peng, K. Kuchcinski. Automated transformation of algorithms into register-transfer implementations. *IEEE Transactions on CAD*, 13(2):150–166, 1994. 188
5. R. Camposano. Behavior-preserving transformations for high-level synthesis. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, number 408 in LNCS, pp. 106–128, Ithaca, New York, 1989. Springer. 188
6. S. D. Johnson, B. Bose. DDD: A system for mechanized digital design derivation. In *Int. Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991. Available via "ftp://ftp.cs.indiana.edu/pub/techreports/TR323.ps.Z" (rev. 1997). 188
7. M. J. C. Gordon, T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993. 188
8. R. Sharp, O. Rasmussen. The T-Ruby design system. In *IFIP Conference on Hardware Description Languages and their Applications*, pp. 587–596, 1995. 188
9. E. M. Mayger, M. P. Fourman. Integration of formal methods with system design. In *Int. Conf. on VLSI*, pp. 59–70, Edinburgh, Scotland, 1991. North-Holland. 188
10. R. Kumar et al. Formal synthesis in circuit design-A classification and survey. In *FMCAD'96*, number 1166 in LNCS, pp. 294–309, Palo Alto, CA, 1996. Springer. 188
11. F. K. Hanna et al. Formal synthesis of digital systems. In *Applied Formal Methods For Correct VLSI Design*, volume 2, pp. 532–548. Elsevier, 1989. 188
12. M. Larsson. An engineering approach to formal digital system design. *The Computer Journal*, 38(2):101–110, 1995. 188
13. D. Gajski et al. *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer, 1992. 188
14. H.P . Barendregt. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 7: Functional Programming and Lambda Calculus, pp. 321–364. Elsevier, 1992. 191
15. A. Aho et al. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986. 193
16. P. G. Paulin, J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on CAD*, 8(6):661–679, 1989. 194
17. C. Blumenröhr. A formal approach to specify and synthesize at the system level. In *GI Workshop Modellierung und Verifikation von Systemen*, pp. 11–20, Braunschweig, Germany, 1999. Shaker-Verlag. 197

18. D. Eisenbiegler, R. Kumar. An automata theory dedicated towards formal circuit synthesis. In *TPHOL'95*, number 971 in LNCS, pp. 154–169, Aspen Grove, Utah, 1995. Springer. 198
19. D. Eisenbiegler et al. Implementation issues about the embedding of existing high level synthesis algorithms in HOL. In *TPHOLs'96*, number 1125 in LNCS, pp. 157–172, Turku, Finland, 1996. Springer. 195, 199
20. J. Gerlach, W. Rosenstiel. A Scalable Methodology for Cost Estimation in a Transformational High-Level Design Space Exploration Environment. In *DATE'98*, pp. 226–231, Paris, France, 1998. IEEE Computer Society. 200
21. *http: //goethe. ira.uka.de/fsynth/Charme/<name>.c.* 200
22. C. Blumenröhr et al. On the efficiency of formal synthesis — experimental results. *IEEE Transactions on CAD*, 18(1):25–32, 1999. 200