Performing High-Level Synthesis via Program Transformations within a Theorem Prover *

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
http://goethe.ira.uka.de/fsynth

Abstract

In this paper, we present a new methodology towards performing high-level synthesis. During high-level synthesis an algorithmic description is mapped to a structure of hardware components. In our approach, high-level synthesis is performed via program transformations. All transformations are performed within a higher order logic theorem prover thus guaranteeing correctness. Our approach is not restricted to data flow graphs but supports arbitrary computable functions, i.e. mixed control/data flow graphs. Furthermore, the treatment of algorithmic and interface descriptions is orthogonalised, allowing systematic reuse of designs.

1. Introduction

In hardware synthesis, design errors may be extremely expensive. This implies that one has to find a design methodology, that is safe in a sense that it guarantees correctness throughout the design process. Due to the complexity of nowadays circuits, simulation can never be exhaustive. Also post-synthesis verification is always NP-complete. This paper presents an approach towards design correctness where synthesis is performed via a sequence of logical transformations thus guaranteeing correctness by construction.

This paper addresses 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 of hardware). In our previous work [3], algorithmic synthesis was restricted to pure data flow graphs. The extensions to be presented in this paper allow synthesising arbitrary algorithmic descriptions i.e. mixed control/data flow descriptions.

Our work is based on a formal hardware description language named Gropius[1] ranging from the gate level to the system level. Gropius [2] is a language with a formally exact semantics, where each construct is derived from logic within the HOL [7] theorem prover. In this paper, we will introduce the part of Gropius, that is related to the algorithmic level (section 2), and we will present a new formal hardware synthesis methodology, where the implementation is derived by applying a sequence of program transformations.

There are many hardware description languages. However, there are several reasons, why we believe that Gropius is superior to most conventional description languages. Gropius is defined with a precise formal semantics, it is functional, strongly typed, higher order and polymorphic.

There are also other approaches, where the synthesis process is based on a transformational design style. Unlike such approaches, we formalise the algorithmic description in a mathematical manner and perform the transformations directly in this representation style within the theorem prover HOL. Due to the fact that deriving theorems in HOL is restricted to a small core of rules and axioms, our approach can be considered to be extremely safe as to correctness. This design style, that we call *formal synthesis*, is superior to those, where the correctness of transformations is proved, but the correctness of their implementation is not. In those approaches, there are only paper&pencil proofs for the correctness [9] or the circuit transformations are based on a (non-mathematical) formalisation. Proofs are performed by intuition and not within a mathematical logic [8]. In the CAMAD system [11] for instance, the algorithmic description is given in a Pascal-like notation. For transforming the program, it is translated into a formalisation based on timed Petri-nets. Both the transformations from Pascal to Petri-nets and the transformations within the Petri-nets are pieces of software that are complex and safety critical. There is no explicit proof for the correctness of

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

[1] WALTER GROPIUS (1883-1969), founder of the BAUHAUS *(form follows function)*.

the implementation of these safety critical parts. There are also other formal synthesis approaches, where circuit transformations are performed by applying basic mathematical rules within a theorem prover. However, these are mostly restricted to lower abstraction levels (e.g. Lambda/Dialog [6]) or they are restricted towards checking some plausibility criteria rather than performing a complete proof. See [10] for a survey on formal synthesis approaches.

The starting point for high-level synthesis is an algorithmic description. The result of high-level synthesis is a structure at the Register Transfer level (RT-level). Usually, hardware at the RT-level consists of a data-path and a controller. In the conventional approaches [5], several control states are introduced along a given control/data flow description thus partitioning it into small cycle free pieces of program, each corresponding to one clock tick. Then scheduling, allocation and binding are performed on this exponential number of cycle-free pieces leading to a data-path and a symbolic state transition table. Afterwards, the controller and the communication part are generated.

We have developed a methodology that totally differs from the standard. In our approach, the implementation is derived via program transformations. Synthesis is performed in two steps. The first step transforms the program into an equivalent program with a specific shape that we call SLF-representation (section 4). In this step, pre-proven program equations are applied. In the second step, a pre-proven implementation theorem is applied for mapping the SLF-program to a RT-level structure (section 3). There are several of such implementation theorems each corresponding to a specific pattern for the interface behaviour of the hardware implementation.

2. Formal representation of programs

At the algorithmic abstraction level, the behaviour of the circuit, that has to be synthesised, is represented as a pure software program. The concrete timing of the circuit is not yet considered. Gropius offers appropriate means for this level of abstraction. We will now briefly introduce them — for a detailed description see [1, 2].

In Gropius, we distinguish between two different algorithmic descriptions: DFG-terms and P-terms. Both DFG-terms and P-terms can be used as a starting point for synthesising hardware. DFG-terms represent simple, non-recursive programs that always terminate (Data Flow Graphs). P-terms are a means for representing arbitrary computable functions (Programs).

Both P-terms and DFG-terms are functions. DFG-terms always terminate. The evaluation of P-terms, however, may not terminate. In our approach, P-terms are used for representing entire programs as well as blocks. Blocks are used for representing inner pieces of programs. Blocks are based on conditions and basic blocks. Both basic blocks and conditions are DFG-terms with basic blocks having same input and output type and conditions having a boolean output type.

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: PARTIALIZE (convert a basic block into a block), WHILE (loop), THEN (sequence of blocks), IFTE (conditional branching), LOCVAR (local variable), LEFTVAR and RIGHTVAR (apply block to left/right part of state), PROGRAM (convert a block into a program)

The syntax of DFG-terms, blocks and programs are defined with the following Backus-Naur form:

vblock ::= *variable* | "(" { *vblock* "," } *vblock* ")"

expr ::= *variable* | "(" { *expr* "," } *expr* ")" | *operator* "(" *expr* ")"

DFG-term ::= "λ" *vblock* "." { "let" *vblock* "=" *expr* "in" } *expr*

block ::= "PARTIALIZE" *basic_block* | "WHILE" *condition block* |
block "THEN" *block* | "IFTE" *condition block block* |
"LOCVAR" *constant block* |
"LEFTVAR" *block* | "RIGHTVAR" *block*

program ::= "PROGRAM" *constant block*

Providing only a small basic language for representing programs leads to a small number of syntactic constructs to be considered and therefore reduces the number of program transformations that have to be derived. However, Gropius allows deriving new control structures by the programmer. Here are some examples:

$\vdash$ NOP $=$ PARTIALIZE $(\lambda x.\ x)$
$\vdash$ FOR $m\ n\ s\ A\ =$
LOCVAR m
(WHILE $(\lambda(x,h).\ h\ \leq\ n)$
(A THEN (PARTIALIZE $(\lambda(x,h).\ (x,h\ +\ s))$)))
$\vdash$ LOOP $A\ c\ B\ =\ A$ THEN (WHILE c (B THEN A))
$\vdash$ REPEAT $A\ c\ =$ LOOP $A\ c$ NOP

In our approach, the mapping from an algorithmic description to hardware is performed in two steps: programs are first turned into a specific pattern called single-loop form (see section 4) and then an implementation theorem is applied for mapping the SLF program to hardware (see section 3). Programs in SLF have the following shape.

PROGRAM o_init (LOCVAR v_init (WHILE c (PARTIALIZE a)))

In this expression o_init and v_init denote arbitrary constants, c is a condition and a an arbitrary basic block.

3. Converting programs to the Register Transfer level

The algorithmic description only defines the mapping from input values to output values. Time is not yet considered. To bridge the gap between the algorithmic description and the hardware implementation, the interface behaviour

has to be described, specifying how the algorithm communicates with its environment.

Many approaches in the high-level synthesis domain use a notation, where algorithmic description and interface description are mingled [11]. In our approach algorithmic description and interface description are strictly separated. A fixed set of interface patterns is provided. The circuit designer can first write some ordinary, time independent algorithm and can then select one of the interface patterns, thus defining the way the circuit communicates with the environment. It is easy for the designer to switch from one interface behaviour to another without changing the program. Therefore, this methodology supports the reuse of designs in a systematic manner.

There are many possible interface specification patterns. Usually, the interface of the implementation not only consists of the data signals from the algorithmic description, but there are also additional control signals. They are used to steer the communication and to allow interrupting the execution of the algorithm.

The formula below shows the formal definition of a interface specification pattern called IFC1. It describes the relation between some interface signals *input*, *reset*, *start*, *output* and *ready* with respect to some arbitrary program f.

IFC1 (*input*, *reset*, *start*, *output*, *ready*, f) $=$
$\forall t.\ (((t = 0)\ \vee$ *reset* $t)\ \wedge\ \neg($*start* $t)\ \Rightarrow$ *ready* $t) \wedge$
((*ready* $t\ \wedge\ \neg($*start* $(t\ +\ 1))))\ \Rightarrow$
(*ready* $(t + 1)\ \wedge$ (*output* $(t + 1)\ =$ *output* $t))) \wedge$
((((($t = 0$) $\vee$ *ready* $(t - 1)\ \vee$ *reset* $t)\ \wedge$ *start* $t)\ \Rightarrow$
CASE (f (*input* $t)$) OF
Defined y
$(\exists m.$
$((\forall n < m.\ \neg($*reset* $(t + n + 1)))\ \Rightarrow$
((*output* $(t + m) = y)\ \wedge$ *ready* $(t + m))) \wedge$
$(\forall n < m.$
$(\forall p < n.\ \neg($*reset* $(t + p + 1)))\ \Rightarrow$
$\neg($*ready* $(t + n)))))$
Undefined
$(\forall m.(\forall n < m.\ \neg($*reset* $(t + n + 1)))\ \Rightarrow$
$\neg($*ready* $(t + m))))$

For each interface pattern, that we provide, we also give a correct implementation pattern in terms of an implementation theorem. All implementation theorems expect the algorithmic description to be in SLF. Figure 1 shows the structure of the general hardware implementation that we found for interface pattern IFC1.

We represented this structure in logic and named it IMP1[2]. The following theorem states, that IMP1 fulfils IFC1 for every program f being in SLF with some DFG-terms a and c and arbitrary constants v_init and o_init.

$\vdash\ \forall a\ c$ *v_init o_init*.
IMP1 (*input*, *reset*, *start*, *output*, *ready*, a, c, *v_init*, *o_init*)
$\Rightarrow$
IFC1 (*input*, *reset*, *start*, *output*, *ready*,
PROGRAM o_init
(LOCVAR v_init (WHILE c (PARTIALIZE a))))

[2]For sake of space we will not give the structural description in Gropius. See [2] for structural RT-level descriptions in Gropius.

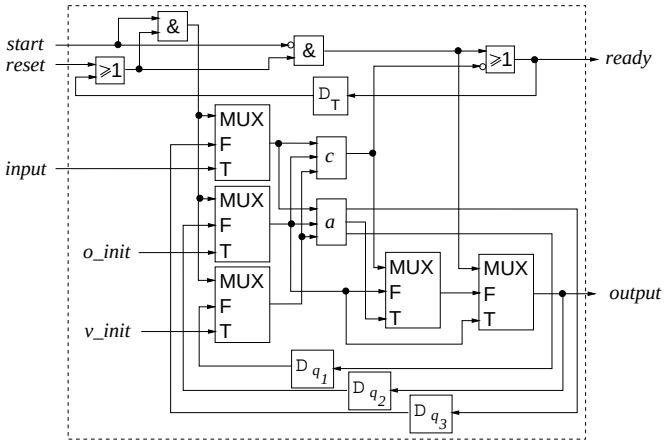

Figure 1. RT-Implementation IMP1

When mapping an algorithmic description to hardware, certain cost functions have to be considered. The main critical aspects are consumption of area and timing behaviour. In general, these optimisation goals are contradictory. When regarding the RT-level implementation in figure 1, one can see that the two DFG-terms a and c directly determine the hardware costs.

4. Converting programs to single-loop form

Every P-term can be transformed into a SLF. However, for every P-term there is not a unique SLF, but there are several equivalent SLFs. Different SLFs lead to different implementations with different costs with respect to hardware consumption and execution speed. In our approach, doing a good high-level synthesis means producing a SLF which corresponds to a cost minimal implementation.

Within the HOL theorem prover environment, we have proven several transformation theorems, which can be subdivided into two groups: SPT (standard program transformations) theorems and OPT (optimisation program transformation) theorems. SPT theorems are used to convert arbitrary programs to SLF. Rewriting with the set of SPT theorems is confluent (i.e. applying them in an arbitrary order always leads to the same result) and always leads to a SLF. OPT theorems are used for optimising the control structures.

The SPT theorem set comprises a fixed number of 27 equations (see theorem 1 for an example). In simplified terms, the equations reduce the number of control structures (THEN, WHILE,. . .) by adding new data variables holding the current control information. Furthermore, local variables are shifted from the inside to the outside of control

structures.

$$\begin{array}{l}\vdash\ \mathsf{WHILE}\ c_1\ (\mathsf{LOCVAR}\ init\ (\mathsf{WHILE}\ c_2\ (\mathsf{PARTIALIZE}\ a))) = \\ \quad \mathsf{LOCVAR}\ init \\ \quad\quad \mathsf{LOCVAR}\ \mathsf{F} \\ \quad\quad\quad \mathsf{WHILE}\ (\lambda((x,h_1),h_2).\, c_1\ x\ \vee\ h_2) \\ \quad\quad\quad\quad \mathsf{PARTIALIZE}\ (\lambda((x,h_1),h_2). \\ \quad\quad\quad\quad\quad \mathsf{MUX}\ (c_2\ (x,h_1),(a\ (x,h_1),\mathsf{T}),((x,init),\mathsf{F})))\end{array} \quad (1)$$

Currently, 12 OPT theorems have been proven. Unlike the SPT theorem set, this set is not fixed and may be extended. A very powerful OPT-theorem implements loop-unrolling. The theorem 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. The advantage of loop-unrolling is, that the combinatorial depth is increased, which reduces the number of clock ticks that are required for executing the program. Theorem (2) gives the definition of a special for-loop FOR_N, which realizes an n-fold application of the same function (see section 2). Theorem (3) shows the general loop-unrolling theorem, and theorem (4) can be used to remove the function FOR_N, after having instantiated n.

$$\vdash\ \mathsf{FOR\_N}\ n\ A\ =\ \mathsf{FOR}\ 1\ n\ 1\ (\mathsf{LEFTVAR}\ A) \quad (2)$$

$$\begin{array}{l}\vdash\ \mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a)\ = \\ \quad \mathsf{WHILE}\ c\ ((\mathsf{PARTIALIZE}\ a)\ \mathsf{THEN} \\ \quad\quad\quad (\mathsf{FOR\_N}\ n\ (\mathsf{PARTIALIZE}\ (\lambda x.\ \mathsf{MUX}\ (c\ x, a\ x, x)))))\end{array} \quad (3)$$

$$\begin{array}{l}\vdash\ \mathsf{FOR\_N}\ 1\ A\ =\ A\ \wedge \\ \quad \mathsf{FOR\_N}\ (\mathsf{SUC}\ n)\ A\ =\ A\ \mathsf{THEN}\ (\mathsf{FOR\_N}\ n\ A)\end{array} \quad (4)$$

Within the HOL theorem proving system, we have proven the SPT theorems and the OPT theorems by hand. These theorems are powerful enough to derive SLFs in different ways. A given program is first optimised by applying some optimisation theorems. The OPT theorems can either be applied by hand or one needs to invoke some heuristics. Afterwards the SPT theorems are applied and to produce a SLF. This can be performed by pure rewriting, which is fully automated by the HOL system.

Converting a program to a SLF corresponds to conventional scheduling, allocation and binding techniques. In our approach, design goals such as hardware consumption and execution speed are reached by selecting suitable theorems among the OPT-theorems. Other than with conventional synthesis techniques, the schedule (the assignment between operations and clock cycles) is not calculated explicitely but is implicitly derived during the theorem applications. Allocation and binding are performed after the SLF-transformation within the basic blocks of the SLF. Performing allocation and binding within basic blocks in formal synthesis has already been presented in [3].

5. Conclusion

In this paper, we have presented a new methodology for deriving RT-level structures from circuit descriptions at the algorithmic level. It differs from other high-level synthesis approaches in three aspects. First, it is formal. The implementation is derived by applying basic logical transformations within a theorem prover thus guaranteeing correctness implicitly. Second, it provides a new synthesis concept. The implementation is derived by applying program transformations rather than extracting a control and data flow graph and analysing an exponential number of control paths. Thirdly, the input language for our high-level synthesis supports design reuse by using interface patterns rather then mingling algorithmic aspects and interface behaviour.

Our hardware description language Gropius can be used to describe circuits at different abstraction levels (see [4]). It provides a consistent concept for a correctness-by-design synthesis style ranging from the algorithmic level down to the gate level.

References

[1] C.Blumenröhr and D. Eisenbiegler. Deriving structural RT-implementations from algorithmic descriptions by means of logical transformations. In *GI/ITG/GME Workshop '98*.

[2] D. Eisenbiegler and C. Blumenröhr. Gropius – a hardware description language for the reuse of designs. to be published in REUSE'98.

[3] D. Eisenbiegler, C. Blumenröhr, and R. Kumar. Implementation issues about the embedding of existing high level synthesis algorithms in HOL. In *TPHOLs'96*.

[4] R. D. Eisenbiegler and C. Blumenröhr. A constructive approach towards correctness of synthesis-application within retiming. In *EDTC'97*.

[5] D. Gajski et al. *High-Level Synthesis, Introduction to Chip and System Design*. 1994.

[6] 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 '91*.

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

[8] P. Middelhoek et al. A methodology for the design of guaranteed correct and efficient digital systems. In *IEEE Int. High Level Design Validation and Test Workshop '96*.

[9] R. Camposano. Behavior-preserving transformations for high-level synthesis. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, 1989.

[10] R. Kumar , C. Blumenröhr, D. Eisenbiegler, and D. Schmid . Formal synthesis in circuit design-A classification and survey. In *FMCAD'96*.

[11] Z. Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer implementations. *TCAD*, 13(2), Feb. 1994.