On the Efficiency of Formal Synthesis – Experimental Results

Christian Blumenröhr, Dirk Eisenbiegler, Detlef Schmid

Abstract— **Formal synthesis has become an interesting alternative towards post-synthesis verification. Formal synthesis means integrating formal validation within the synthesis process by performing synthesis via rule applications. The practical applicability of formal synthesis very much depends on the efficiency of the underlying rules. This paper gives a case study about the complexity of formal synthesis programs. Experiments with two realistic-sized benchmark circuits were performed using the formal synthesis system HASH (Higher order logic Applied to Synthesis of Hardware). HASH provides means for representing and transforming circuits in a secure and logically sound manner. Furthermore, arbitrary synthesis procedures can be invoked to achieve high quality of designs. In this paper, the implementation of a formal scheduling step is used to illustrate efficiency considerations related to formal synthesis.**

Keywords— **Formal methods, formal verification, theorem prover, high-level synthesis, dataflow synthesis, scheduling**

I. Introduction

With the enormous size of today's digital circuits and with the increasing complexity of the synthesis process applied for deriving them, guaranteeing the correctness of hardware synthesis is becoming an important matter. By correctness we mean that the synthesis result (implementation) satisfies the synthesis input (specification), in a formal mathematical sense. Due to the fact that most synthesis steps are nowadays performed automatically, the correctness of hardware implementations strongly depends on the correctness of the synthesis tools being involved. In general, bugs in the programs of those synthesis tools lead to faulty implementations. The "correctness by construction" paradigm, that has been propagated in the synthesis domain, is questionable, since an automated synthesis process is not correct by definition. Due to their complexity, the programs can only be tested partially and it is almost impossible to formally verify them. Actually, the designers do not trust in the synthesis tools and validate the synthesis results by simulation. However, with the increasing complexity of circuits, exhaustive simulation of the implementation is not possible. The solution is that formal methods have to be applied to ensure that the implementation satisfies the specification.

A. Definition and Delimitation of Formal Synthesis

The correctness of the synthesis process can be guaranteed at three different phases: before, during or after the synthesis process. These approaches towards formal correctness of synthesis are called pre-synthesis verification, formal synthesis and post-synthesis verification, respectively [1]. All approaches use some kind of logic for proving the correctness of the implementation with respect to the specification. However, in pre-synthesis verification, the proof is derived even before synthesis happens, in formal synthesis the proof is formally derived during synthesis, and in post-synthesis verification, the proof is derived after each synthesis run.

Pre-synthesis verification would be favorable, since it means that the correctness is proven once and for all by means of software verification. Since this is extremely tedious especially for large sized programs such as synthesis tools, only few activities have been started in this area [2], [3].

At the moment, post-synthesis verification approaches [4], [5] like model checking and theorem proving are the most frequently used formal methods. First synthesis is performed in a conventional, non-formal manner. Afterwards, the correctness of the synthesis output is proven with respect to the synthesis input. During verification, the information on *how* the synthesis process was performed is no longer available. This is a significant drawback of this method. Since implementations of real-world specifications often comprise a large state space, verification for large sized circuits is tedious or even impossible. When performing model checking [6], increase in design size results in a combinatorial explosion in the number of global states. Although research has gone very far in the model checking area and there exist large circuits that can be verified efficiently, there are several important classes of circuits for which no efficient model checking techniques exist (counters, data paths with a large bitwidth, complex units like multipliers). Theorem proving on the other hand is not limited by the design size, but requires a lot of interaction. This is unacceptable for circuit designers, since performing complex correctness proofs by hand requires strong logical skills and a profound knowledge of the theorem prover.

In *formal synthesis* the synthesis process and the logical validation are strongly interwoven. Other than in conventional synthesis, where hardware is represented by arbitrary data structures, formal synthesis requires a mathematical hardware representation. In contrast to conventional synthesis, where the specification is refined or optimized by arbitrary procedures, formal synthesis is restricted to the application of a small core of basic mathematical rules within a theorem prover. The correctness of a formal synthesis process only depends on the correct implementation of these basic transformation rules. Therefore, formal synthesis programs are an extremely safe alternative

Institute for Circuit Design and Fault Tolerance, University of Karlsruhe, Germany. E-mail: {blumen,eisen,schmid}@ira.uka.de

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

to conventional synthesis programs.

B. State of the Art

In the last years, formal synthesis has become a new research topic and several systems have been introduced such as T-Ruby [7], Lambda/Dialog [8], Veritas [9], DDD [10] or HASH [11]. [1] gives a survey on different approaches in the formal synthesis domain.

There is also a broad range of scientific work for deriving implementations in a transformational design style. Synthesis is performed by applying a fixed set of basic circuit transformations, that are described in a more or less mathematical manner. Paper & pencil proofs are performed to prove the correctness of the circuit transformations [12]. However, the correctness of the implementation of the circuit transformations is not considered. In the CAMAD system [13] the algorithmic description is given in a Pascal-like notation. For transforming the program, it is first translated to a timed Petri-net representation. Both the transformations from Pascal to Petri-nets and the transformations within the Petri-nets are pieces of software that are both complex and crucial to soundness and correctness. However, there is no explicit proof for the correctness of the implementation of these critical parts. The TRADES system [14], [15] uses an approach, where the elementary transformations are performed by graph-rewriting. This representation style is based on a formal semantics, but there is no explicit proof for the correctness of their implementation.

In formal synthesis, the implementation of the theorem provers core is not formally verified either. However, this implementation only consists of few hundred lines of code. This core, which is the only crucial part for correctness, is fixed. No matter, how big and complex the formal synthesis program may become, the size of the core remains fixed.

C. Objective of the Paper

In the following section we briefly introduce our formal synthesis system HASH. In the rest of the paper, we illustrate efficiency considerations at some specific synthesis step during high-level synthesis: scheduling.

The efficiency of formal synthesis approaches depends on how efficient hardware can be represented and on how fast circuit transformations can be realized based on the given set of logical transformations. In general, there are various ways to realize such transformations. However, the complexity very much depends on the basic set of logical transformations being used and on the order in which the logical transformations are applied. This paper investigates efficiency aspects of implementations of formal synthesis transformations. Although there is a broad range of formal synthesis implementations, there are some very general efficiency considerations. The paper discusses the implementation of the scheduling step in the HOL theorem proving environment [16]. This leads to a general discussion about the efficiency of the implementation of formal synthesis transformations (section III).

Besides optimizing the circuit representation style and the order of the rule applications, it is also possible to modify the implementation of the underlying calculus (section IV). We will introduce a modification of the HOL theorem prover and discuss the impact towards soundness and efficiency. In section V, we will discuss the extra costs for formal synthesis as compared to conventional synthesis.

II. The Formal Synthesis Approach HASH

In order to get along with the complexity of large formal synthesis programs, we believe that it is absolutely necessary to make a split between basic circuit transformation steps and the design space exploration steps. Due to this split, the heuristics for exploring the design space can be performed outside the logic and the results imported into the design transformation. Therefore standard synthesis algorithms that abound in literature [17], [18] can be exploited. It shows, that this split can be successfully applied to many synthesis steps and that for each synthesis step, e.g. scheduling, state minimization, retiming, etc., there must exist a unique transformation which performs this synthesis step. It is to be noted here, that these transformations are specific to a synthesis step but independent of the design space exploration technique. The logical transformations are implemented as a sequence of basic logical rule applications. Given an input circuit description (specification), some synthesis heuristic is started calculating the control information. Then the output circuit description (implementation) is generated according to the control information and a theorem is derived that the implementation implies or is equivalent to the specification. This basic concept is shown in figure 1.

// Insert figure 1 here //

Two important points are met independently with this strategy: quality and correctness of the implementation. The quality only depends on the algorithm that calculates the control information, whereas the correctness aspect is guaranteed due to the transformation being based on the HOL system.

The HOL system [16] is a higher order logic theorem prover environment. The core of HOL consists of five axioms and eight primitive inference rules. The only way to derive new theorems from existing ones is via these rules. False theorems cannot be proven. Since the entire formal synthesis process is nothing but a conversion in HOL, correctness is guaranteed implicitly. Faulty implementations cannot be achieved no matter how the design space exploration step was implemented. If the control information produced by the heuristic is flawed, HOL will at some point produce an exception, but it will never derive a false result. Our formal synthesis program either leads to correct implementations or to no implementation but an exception. In case of an exception, an information is produced telling the user in which synthesis step the error occurred. In conventional synthesis programs, however, such bugs might lead to faulty implementations.

In our approach, circuit transformations guarantee that the functional behavior is preserved. Besides functional correctness, there may be further requirements for the implementation such as timing and area constraints. However, checking, whether such constraints are fulfilled, can easily be done by non-formal methods. Therefore, it is not necessary to formally represent and verify such constraints in logic.

III. Scheduling Transformation

As mentioned before, our formal synthesis methodology can be applied to many synthesis steps. In this paper we want to restrict ourselves to the scheduling task within high-level synthesis to demonstrate the applicability of our approach.

High-level synthesis converts an algorithmic circuit description into a structure at the Register-Transfer (RT) level. The major steps in high-level synthesis are scheduling, allocation of storage, functional and interconnection units, binding the allocated hardware onto some library components and interface synthesis.

For a better understanding, the starting point for scheduling in this paper is a basic block of some algorithmic description. It represents a pure data flow graph. However, this is no restriction of our approach. The handling of mixed control/data flow graphs is described in [19].

The scheduling task assigns a control step (c-step) to each operation in the algorithmic specification. There are various heuristic scheduling algorithms trying to minimize the number of control steps and the hardware requirements [18], [20]. Figure 2 gives an example of some scheduling step. The input data flow graph represented by the function g is split into a sequence of functions $g_4 \circ g_3 \circ g_2 \circ g_1$, where g_i corresponds to c-step i. During scheduling, the number of c-steps has to be determined and each basic operation within g has to be assigned to one of the c-steps.

// Insert figure 2 here //

Our approach allows pipelining and to some extend chaining. Chaining is not possible, if it leads to combinatorial cycles, which cannot be expressed by our representations style (see section III-A). Furthermore, our approach currently does not support multi-cycle operations. However, if those operations can be performed by pipelining, i.e. they can be expressed by a sequence of partial operations, it is possible to integrate them. On the other hand we have currently no solution for multi-cycle operations that are not pipelined and are performed by units with internal controller and registers.

A. Formalizing Data Flow Graphs

In general, there are various ways to formalize hardware descriptions in logic. The representation style may have a significant impact on the efficiency of the hardware transformations. However, there are common problems. When describing circuit structures, one needs (local) variables to indicate interconnections. This paper will discuss efficiency aspects when transforming such structures in logic.

In our approach, data flow graphs are represented by means of functions that are nothing but simple compositions of basic operations. They are formalized using λ-expressions [21]. The following Backus-Naur form shows the syntactical structure of data flow graphs:

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

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

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

The two terms illustrated in figure 3 show, how the original and the scheduled data flow graph in figure 2 are represented in HOL.

// Insert figure 3 here //

The expressions in figure 3 describe input/output functions in terms of their basic operations. The functions map some input tuple to an output tuple. Each let-term describes the connectivity of one operation, i.e. a node of the data flow graph. Since these terms represent pure data flow graphs, i.e. there are no cycles, a partial ordering on the set of nodes is induced. This partial order corresponds to the fact, that some operation A must be executed before B if the output of A happens to be an input to B. This partially ordered data flow graph is represented as an arbitrarily ordered list, whereby the data dependencies between the nodes are respected.

Let-terms are used for a better readability of β-redices [21], where let $x = y$ in z is equivalent to the β-redex $(\lambda x.z)\, y$. A β-redex $(\lambda x.z)\, y$ expresses that the argument y is applied to the λ-abstraction $(\lambda x.z)$ that maps some x onto some z. One of the basic rules of functional algebra asserts that a λ-expression can be expanded where it is applied (β-reduction); and conversely, that a term can be extracted to the application of a λ-expression (β-expansion):

$$(\lambda x.z)\, y \Leftrightarrow z[y/x],$$

where $z[y/x]$ denotes the expression, where every occurrence of x in z is replaced by the argument y. If x and y are tuples, these conversions are called paired β-reduction/expansion.

B. Transforming the Data Flow Graphs within HOL

During scheduling, the function g is split into a concatenation of functions $g_1, g_2, \ldots, g_k$ with $g = g_k \circ \ldots \circ g_2 \circ g_1$, and each function again represents a data flow graph (see figure 3).

This section describes, how the scheduling process described in figure 2 is implemented as a conversion in HOL. Our conversion is steered by an external control information, the schedule table. The schedule table indicates, which operation has to be performed in which control step.

The conversion derives a theorem stating that the original and the scheduled data flow graph are equivalent. In this section we will only describe the logical aspects of formally deriving the synthesis result from the input data flow graph. The computation of the control information and invocation of the external heuristics will be demonstrated in section V.

The approach is based on a conversion for normalizing functions. We will first describe this conversion and then describe, how scheduling can be realized based on this conversion.

C. Function Normalization

Both the input and the output of the scheduling process are nothing but simple compositions of the same basic operations (see figure 3). Normalizing such representations is pretty simple. The general algorithm looks as follows:
1. the original term g is converted to $\lambda(x_1, x_2, \ldots, x_m).\ g(x_1, x_2, \ldots, x_m)$ by applying a paired η-reduction in the inverse direction
2. the $\circ$-operations are expanded by rewriting
3. β-reductions and paired β-reductions are performed wherever possible

Given some function g and the scheduled function $g' = g_k \circ \ldots \circ g_2 \circ g_1$, this algorithm leads to the same normalized representation: $\lambda(x_1, x_2, \ldots x_m).v[x_1, x_2, \ldots, x_m]$. In $v[x_1, x_2, \ldots x_m]$ there are no β-redices left and there are only pure function applications.

D. A Universal Conversion

We will now introduce a simple conversion which is based on this normalization scheme. Given some data flow graph representation g and some schedule table, the following steps have to be performed:
1. produce $g' = g_k \circ \ldots \circ g_2 \circ g_1$ according to the schedule table
2. derive $\vdash g = \hat{g}$ and $\vdash g' = \hat{g}$ via normalization
3. The equations $\vdash g = \hat{g}$ and $\vdash g' = \hat{g}$ are combined to $\vdash g = g'$

The major drawback of this universal conversion is the complexity of step 2 when dealing with data flow graphs with a big depth, i.e. maximum number of operations on a path from some input to some output. Data flow graphs whose intermediate nodes have larger fanouts, i.e. the output of a node is used by many successor nodes as inputs, lead to a number of duplications during β-reduction. Due to the fact that during β-reduction one has to traverse the entire term and since such β-redices can be nested, the term size and time consumption in step 2 may grow exponentially with the depth.

E. An Advanced Conversion

The universal conversion does not exploit any knowledge about how the synthesis step was performed. The advanced conversion exploits this knowledge. In the advanced scheduling conversion the data flow graph is split step by step rather than all at once. Let k be the number of control steps. The scheduling transformation is performed by a sequence of $k \Leftrightarrow 1$ transformations, each of them splitting apart one function g_i. Each step transformation is similar to the universal conversion except that step 2 is modified. β-reduction is only applied to those variables, whose corresponding nodes have been assigned to the considered control step. Although this modified normalization step does not eliminate all β-redices, the terms achieved after each step will be equal. Hence, for data flow graphs with larger fanouts the exponential complexity associated with step 2 is avoided and the overall cost is reduced. Since the advanced conversion performs step 2 several times, it is to be expected that for small data flow graphs it is slower, but faster for larger ones.

In the following section we present some experimental results that demonstrate the differences between the simple and the advanced conversion.

F. Experimental Results

We consider two scalable data flow graphs. As a first example, we use a data flow graph, that realizes the discrete cosine transform (DCT), which is frequently used for image compression. The DCT of an image with pixels $x(n, m)$ is defined by:

$$X(u,v) = \frac{2}{\sqrt{N \cdot M}} \cdot c(u) \cdot c(v) \cdot \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} x(n,m) \cdot \cos[\frac{\pi \cdot u}{2N} \cdot (2n+1)] \cdot \cos[\frac{\pi \cdot v}{2M} \cdot (2m+1)]$$

with

$$c(u), c(v) = \begin{cases} \frac{1}{\sqrt{2}} & : \quad u, v = 0 \\ 1 & : \quad \text{otherwise} \end{cases}$$

In the following, we assume that $N = M$. In the data flow graph, there is a total of $2N^3 \Leftrightarrow N^2 \Leftrightarrow N$ additions and $2N^3 \Leftrightarrow N + 2$ multiplications. The length of the critical path is $2N + 1$. A more detailed description of this data flow graph can be found in [22].

The second example implements a division of two polynomials (PD) according to the following equation.

$$\frac{\sum_{i=0}^{p+q} \alpha_i \, x^i}{\sum_{i=0}^{p} \beta_i \, x^i} = \sum_{i=0}^{q} \gamma_i \, x^i + \frac{\sum_{i=0}^{p-1} \delta_i \, x^i}{\sum_{i=0}^{p} \beta_i \, x^i}$$

α_i and β_i indicate the coefficients of dividend and the divisor, respectively. γ_i and δ_i indicate the coefficients of the result and the rest. The following equations describe the data flow graph for mapping the coefficients α_i and β_i to γ_i and δ_i.

$$\gamma_i = \alpha_{i+p} \Leftrightarrow \sum_{k=i+1}^{min\{i+p,q\}} \beta_{i+p-k} \cdot \gamma_k \qquad i = 0 \ldots q$$

$$\delta_j = \alpha_j \Leftrightarrow \sum_{k=0}^{min\{j,q\}} \beta_{j-k} \cdot \gamma_k \qquad j = 0 \ldots p \Leftrightarrow 1$$

The data flow graph consists of $p+q$ subtracters, $p(q+1)$ multipliers and $q(p \Leftrightarrow 1)$ adders. The critical path has a length of $3q+2$ nodes.

The structures of the two data flow graphs differ both in the depth and the number of reused intermediate results. In contrast to the DCT, the PD data flow graphs comprise nodes with larger fanouts, which are nested, additionally. In figure 4, an example is shown, where these nodes are marked.

// Insert figure 4 here //

Figure 5 shows the runtime for transforming the PD data flow graph with the simple and the advanced conversion. We set p to 25 and increased q thus varying the size of the benchmark. For this data flow graph, the use of the advanced conversion leads to far better results. Due to the exponential memory consumption of the simple conversion, the computer's capacity of 1.2 GB was exceeded at about 600 nodes.

// Insert figure 5 here //

When applying the same conversions to the DCT example, however, it shows, that the simple conversion seems to be superior (see figure 6). This is due to the fact, that both the length of the critical path and the fanouts are smaller than with the PD example. However, for even larger data flow graphs, it is to be expected that the simple conversion grows faster and finally consumes more time than the advanced conversion.

// Insert figure 6 here //

IV. Changing the Core of the Theorem Prover

In this section we present an optimized HOL theorem proving system named HOL'. HOL' differs from HOL in two points: the term representation is changed and two basic rules are added.

A. Changing the Term Representation

In the HOL theorem prover, terms are implemented in a so-called *deBrujin*-style [21], which means that free and bound variables are represented differently. A variable x is said to be a bound variable, iff it is within the scope of a λ-abstraction whose identifier is x; otherwise x is called free. In the deBrujin representation, free variables are stored with their name and type, whereas bound variables are only represented by a number, linking to the λ-abstraction they refer to. The number describes the distance between the bound variable and the λ-abstraction in terms of λ-abstractions. Example: Consider the term $\lambda x.(\lambda y.\, x+y+z)$. In the subterm $x+y+z$, the two bound variables x and y are internally represented by 1 and 0, respectively. One advantage of the deBrujin representation is, that checking the α-equivalence of two terms (only the names of bound variable are changed) can be performed in linear time.

However, the deBrujin term representation has a major drawback. During the construction and destruction of terms, one has to switch between bound and free variables. In the deBrujin-style term representation, this means traversing the entire term. For big terms with local variables having a large range — such as circuit structures — this leads to a quadratic complexity when building, destructing and transforming such terms.

Therefore, in HOL' a so-called *name-carrying* term representation was implemented. Here, also the bound variables are stored with their name and type. It has the advantage that many terms, such as circuit descriptions, can be handled in a far more efficient manner. However, there are also some basic logical operations such as α-equivalence check that became less efficient. Furthermore, this approach also increases memory consumption due to the fact that bound variables are not any more represented by a single number but are represented by their name and type.

B. Introduction of More Efficient Functions

The original HOL system only allows one single β-reduction at a time. One can increase the efficiency of the scheduling transformation by performing several β-reductions in a single term traversal step. By adding this conversion to the core, we improved the efficiency of the theorem prover. This advanced β-conversion is equipped with a filter, i.e. a characteristic function, for selecting the bound variables that are to be expanded. This is useful for our advanced conversion expanding only those variables, that occur within the considered control step (see section III-E).

In the original HOL system, a paired β-reduction with an n-tuple needs n term traversal steps. However, based on the advanced β-conversion, paired β-reduction can be performed within two traversals. For the implementation of the advanced paired β-conversion we added a second conversion, that first switches some paired β-redex into an equivalent unpaired representation. Example: $(\lambda(x,y).t)\,(a,b)$ is turned to $(\lambda x.(\lambda y.t))\,a\,b$. In a second traversal, the advanced β-conversion is applied at the locations indicated by the filter function.

These two extra conversions are not only tailored to our application, but are of general interest for other users of the HOL system. However, it has to be noted, that enlarging the core is crucial to soundness and correctness, since faults within the implementations of the modified rules may violate the consistency of the calculus. Therefore, one has to be extra careful with the implementation and one should be very restrictive with the size of the core.

Figures 7 and 8 compare runtimes for scheduling conversions within HOL and HOL'. It shows, that both the simple and the advanced conversion runs extremely faster with HOL'. Again, due to nested β-redices in the PD data flow graphs, the simple conversion ran into memory problems for larger data flow graphs.

// Insert figure 7 here //

// Insert figure 8 here //

V. The Formal Synthesis Scenario Applied to Scheduling

Figure 9 demonstrates, how the scheduling step is performed by means of the small example in figure 2. Given a data flow graph, some scheduling heuristic (here: force-directed [23]) is started. The heuristic returns a schedule table assigning each operation in the data flow graph to a control step. This heuristic step has nothing to do with logic. The scheduling table is then handled to the logical transformation, that maps the input data flow graph to the output data flow graph according to the schedule table. Additionally, a correctness theorem is derived.

// Insert figure 9 here //

Figure 10 shows a HOL session performing this scheduling step. The HOL conversion `SCHEDULING_CONV` accomplishes the scheduling transformation according to the schedule which is determined by the scheduling heuristic. `SCHEDULING_CONV` gets the scheduling heuristic as a parameter. In this example, we applied the force-directed scheduling heuristic. Any other scheduling heuristic can be embedded as well.

// Insert figure 10 here //

Our formal synthesis system HASH can be combined with every synthesis tool, provided that the synthesis tool can deliver the control information such as the schedule table to the HASH transformation. The circuit designer then can select the synthesis heuristic and HASH performs the transformation according to the result of the heuristic. This can even be done in the background, while the designer is busy with the next synthesis step.

The overall cost for performing a single formal synthesis step is the sum of performing the synthesis step conventionally by some heuristic and the cost for the logical transformation. Therefore, the ratio between these parts is quite interesting.

Figures 11 and 12 show the runtimes for both design space exploration and transformational part of the formal synthesis step for the DCT data flow graph and the PD data flow graph, respectively. For determining the schedule table, we applied both the force-directed (FD) and the ALAP program. The circuit transformation was performed in HOL' (the modified HOL system) by applying the simple conversion for the DCT and the advanced conversion for the PD.

// Insert figure 11 here //

As can be seen, the runtime for sophisticated design space exploration techniques such as force-directed scheduling exceeds the runtime for the transformational part by far. Even for simple design space exploration techniques, the ratio between design space exploration and circuit transformation seems reasonable. Additionally, it shows, that even large-sized data flow graphs can be synthesized with our method and that the runtime for the transformations depends on the structure of the data flow graph but is independent of the design space exploration technique involved.

Of course, there may be circuits and synthesis steps, where the extra-cost for formal synthesis is much higher than the cost for conventional synthesis. However, one has to be aware of the fact that the alternative to formal synthesis is a conventional synthesis with subsequent exhaustive simulation or post-synthesis verification, which is in general extremely costly.

In this paper, we have restricted ourselves to the scheduling problem. Nevertheless, also formal synthesis steps for binding, allocation and interface synthesis have been realized, but are not discussed here. Thus we are able to perform a complete formal high-level synthesis for data flow graphs.

VI. Conclusions

In this paper we have presented a technique to derive correct implementations in a secure manner. This technique is not restricted to a certain abstraction level (see [11] for performing formal synthesis at the RT-level).

For the scheduling task, we have illustrated the efficiency problem when performing the synthesis step by means of a logical transformation. It shows, that it is very important to be aware of the complexity of the basic logical transformation when constructing formal synthesis programs. On the other hand it may be worthwhile to modify the basic logical transformations themselves.

Experimental results have demonstrated that formalizing the synthesis process as a sequence of logical transformations may not be a significant performance factor. Our experiences with formal synthesis programs at the algorithmic level and the RT-level confirm us to claim that efficient formal synthesis implementations are applicable even for large sized circuits and that the extra-costs as compared to conventional synthesis are reasonable or at least less than the extra-costs for post-synthesis verification.

Acknowledgments

The authors are grateful to the anonymous referees whose constructive comments have improved the quality of the paper.

References

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

[2] M. Aagaard and M. Leeser, "Verifying a logic synthesis algorithm and implementation: A case study in software verification," *IEEE Transactions on Software Engineering*, Oct. 1995.

[3] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri, "Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis," in *International Conference on Computer Design 1998*, Austin, USA, 1998, IEEE-Press.

[4] A. Gupta, "Formal hardware verification methods: A survey," *Formal Methods in System Design*, vol. 1, no. 2/3, pp. 151–238, 1992.

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

[6] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking," 1996, vol. 152 of *Nato ASI Series F*, Springer-Verlag.

[7] R. Sharp and O. Rasmussen, "The T-Ruby design system," in *IFIP Conference on Hardware Description Languages and their Applications*, 1995, pp. 587–596.

[8] E.M. Mayger and M.P. Fourman, "Integration of formal methods with system design," in *International Conference on Very Large Scale Integration*, A. Halaas and P.B. Denyer, Eds., Edinburgh, Scotland, Aug. 1991, IFIP Transactions, pp. 59–70, North-Holland.

[9] F.K. Hanna, M. Longley, and N. Daeche, "Formal synthesis of digital systems," in *Applied Formal Methods For Correct VLSI Design*, Luc J. M. Claesen, Ed. IMEC-IFIP, 1989, vol. 2, pp. 532–548, Elsevier Science Publishers.

[10] S.D. Johnson and B. Bose, "DDD: A system for mechanized digital design derivation," in *International Workshop on Formal Methods in VLSI Design*, Miami, Florida, Jan. 1991, ACM/SIGDA, Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).

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

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

[13] Z. Peng and K. Kuchcinski, "Automated transformation of algorithms into register-transfer implementations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 2, pp. 150–166, Feb. 1994.

[14] P.F.A. Middelhoek, *Transformational Design: An Architecture Independent Interactive Design Methodology for the Synthesis of Correct and Efficient Digital Systems*, Ph.D. thesis, Universiteit Twente, NL, 1997.

[15] C.Huijs, "Transformational design of digital systems based on graph rewriting," in *Proceedings of ProRISC/IEEE Workshop*, Mierlo, The Netherlands, Nov. 1996.

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

[17] D.E. Thomas, E.D. Langnese, R.A. Walker, J.A. Nestor, J.V. Rajan, and R.L. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer Academic Publishers, 1990.

[18] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis, Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

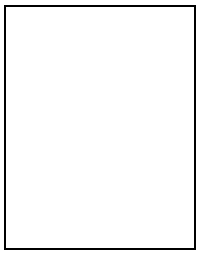
[19] C.Blumenröhr and D. Eisenbiegler, "Performing high-level synthesis via program transformations within a theorem prover," in *Digital System Design Workshop at the 24th EUROMICRO 98 Conference*, Vaesteraas, Sweden, 1998, pp. 34–37, IEEE-Press.

[20] R. Camposano and W. Wolf, *High-Level VLSI Synthesis*, Kluwer, Boston, 1991.

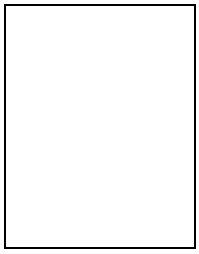
[21] H. P. Barendregt, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 7: Functional Programming and Lambda Calculus, pp. 321–364, Elsevier, 1992.

[22] C.Blumenröhr, D. Eisenbiegler, and R.Kumar, "Applicability of formal synthesis illustrated via scheduling," in *Workshop on Logic and Architecture Synthesis*, Grenoble, France, Dec. 1996, Institut National Polytechnique de Grenoble.

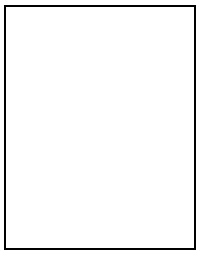
[23] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer Aided Design*, vol. 8, no. 6, pp. 661–679, June 1989.



Christian Blumenröhr studied electrical engineering at the Universität Karlsruhe and received his diploma in 1995. Since 1995 he is a research assistant at the Institute for Computer Design and Fault Tolerance at the same university. There he first was a member of the machine learning group. Since 1996 he is with the formal synthesis group and now works on a research project called *formal circuit design*. His research interests include formal methods, high-level synthesis and system level synthesis.



Dirk Eisenbiegler studied computer science at the Universität Karlsruhe and received his diploma in 1993. From 1993 until 1996 he was research assistant at the department "automation of circuit design" (ACID) at the Forschungszentrum Informatik (FZI) in Karlsruhe. Since 1996 he is with the Institute for Circuit Design and Fault Tolerance at the Universität Karlsruhe. Since 1996 he works on a research project called *formal circuit design*. His research interests include formal methods and synthesis.



Detlef Schmid is the head of the Institute for Computer Design and Fault Tolerance at the Universität Karlsruhe. In 1972/73, he was the first head of the department of the new faculty for computer science in Karlsruhe, which is now one of the most successful faculties for computer science in Europe. In 1974, he was the founder of the "Fakultätentag Informatik", a federation of all german universities working in the field of computer science. From 1985 to 1996 he was the head of the ACID-group at the Forschungszentrum Informatik (FZI) in Karlsruhe. Since 1989 he is member of the board of curators at the Institute for Microelectronics in Stuttgart, and since 1992 he is elected reviewer of the Deutsche Forschungsgemeinschaft for the area technical science.