An Efficient Representation for Formal Synthesis [*]

Christian Blumenröhr, Dirk Eisenbiegler

Institute for Circuit Design and Fault Tolerance
(Prof. Dr.-Ing. D. Schmid)
University of Karlsruhe, Germany
blumen,eisen@ira.uka.de

Abstract

In the last years, performing synthesis by logical refinement has become an interesting alternative towards post-synthesis verification. This paper gives a case study about the complexity of formal synthesis programs within a given calculus. For a simple synthesis step, it is discussed, how one can efficiently implement circuit transformations with respect to the complexity of the logical transformations of the underlying calculus.

1. Introduction

Nowadays, more and more complex and sophisticated synthesis tools are involved during the design of digital circuits. However, even a fully automated synthesis process does not necessarily mean “correctness by construction”. In general, bugs in synthesis tools lead to faulty implementations, and due to their complexity, it is almost impossible to formally verify synthesis tools. Therefore, formal methods have to be applied to ensure that the implementation satisfies the specification. A possible approach is post-synthesis verification [Gupt92]. However, fully automated post-synthesis verification is only achievable for comparatively small sized circuits at lower levels of abstraction, and theorem proving that requires interactive steps is not very much accepted by circuit designers.

Formal synthesis is a complementary approach to hardware verification, since formal averment is an integral part of the synthesis process. In [KBES96] a survey of and a classification scheme for formal synthesis approaches can be found. In the last years, formal synthesis has become a new research topic and several systems have been introduced such as T-Ruby [ShRa95], Lambda/Dialog [MaFo91], Veritas [HaLD89], DDD [JoBo91] or HASH [EiKB97]. [Busc92] and [BaFr96] present reimplementations of the Veritas formal synthesis approach for different theorem provers (LAMBDA and ISABELLE, respectively). Other research activities have been started by [GrMT94], [Wang92] or [Lars95]. They all have one thing in common: they are based on some calculus, i.e. some small core of basic logical transformations. 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.

This paper investigates efficiency aspects of implementing formal synthesis transformations in the theorem proving environment HOL [GoMe93] illustrated by a basic circuit transformation step. In general, there are various ways to perform such transformations. However, the complexity very much depends on which basic logical transformations are used and on the order in which the logical transformations are applied. Therefore, when implementing such synthesis transformations, one has to consider the time complexity of the basic logical transformations involved. We will introduce a simple algorithm and discuss, how its performance can be improved.

In order to optimize the performance of formal synthesis, it is also possible to modify the core of the theorem prover such that the considered circuit transformations can be implemented more efficiently. However, one has to be careful, since modifying the core of a theorem prover may violate the consistency of the calculus. We will introduce a modification of the HOL theorem prover and discuss the impact towards safety and efficiency.

At the end of the paper, we will discuss the extra costs for formal synthesis as compared to conventional synthesis. In our approach HASH (Higher order logic Applied to Synthesis of Hardware), which exploits standard synthesis algorithms, circuit transformations are strictly divided from the design space exploration parts (calculating the scheduling table, determining the state encoding, etc.). The design

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

space exploration parts are the same as in conventional synthesis programs [TLWN90, GDWL94]. The extra cost for formal synthesis arise from the circuit transformations, that are to be performed by rule applications within a theorem prover.

2. Scheduling transformation

In this paper, we concentrate on the transformation in HASH for performing the scheduling task within high-level synthesis. High-level synthesis converts an algorithmic description of the circuit 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.

The scheduling task assigns a control step (c-step) to each operation in the algorithmic specification. There exist various heuristic scheduling algorithms which try to minimize the number of control steps or the hardware requirements [CaWo91, GDWL94]. A large number of them start from data flow graphs that correspond to the basic blocks in the algorithmic description. Although certain scheduling algorithms start from control/data flow graphs, we shall restrict ourselves to pure data flow graphs in this paper. The handling of control flow is the topic of our current research. In figure 1, the scheduling task is performed for a small data flow graph.

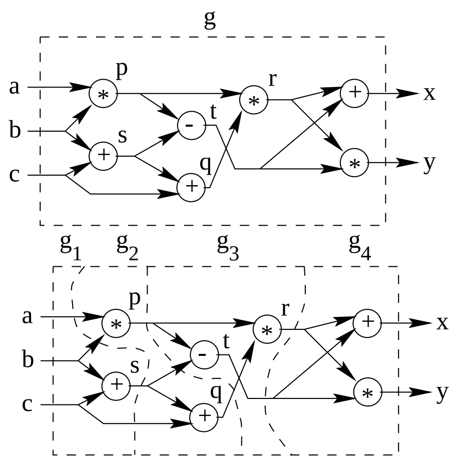

Figure 1. Example for a scheduling step

2.1. Formalizing data flow graphs

In order to transform the data flow graphs, they first have to be formalized suitably. 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 [Davi89].

The following term shows, how the data flow graph in figure 1 is represented in HOL:

$$
\begin{array}{l}
\lambda(a,b,c). \\
\quad \mathsf{let}\ p = a * b\ \mathsf{in} \\
\quad \mathsf{let}\ s = b + c\ \mathsf{in} \\
\quad \mathsf{let}\ q = s + c\ \mathsf{in} \\
\quad \mathsf{let}\ r = p * q\ \mathsf{in} \\
\quad \mathsf{let}\ t = p - s\ \mathsf{in} \\
\quad \mathsf{let}\ x = r + t\ \mathsf{in} \\
\quad \mathsf{let}\ y = r * t\ \mathsf{in} \\
\quad\quad (x,y)
\end{array}
$$

The above expression describes an input/output function in terms of its basic operations. The function maps some input triple (a,b,c) to an output pair (x,y). Each let-term describes the connectivity of one operation. let-terms stand for β-redices, where $\mathsf{let}\, x = y\, \mathsf{in}\, z$ means $(\lambda x.z)\, y$. Using let-expressions improves readability.

2.2. 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 1).

This section describes, how the scheduling process described in figure 1 is implemented as a conversion in HOL. Our high level synthesis conversion is steered by external control information (the schedule table). The implementation of other high level synthesis conversions (allocation and binding of storage and (multi-purpose) functional units) is described in [EiBK96]. 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 discussed in section 4.

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.

2.3. Function normalization

The HOL representations corresponding to figure 1 are both nothing but simple compositions of the same basic functions. In principle, 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, provided there are any

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, which looks like $\lambda(x_1, x_2, \ldots x_m).v[x_1, x_2, \ldots, x_m]$. In $v[x_1, x_2, \ldots x_m]$ there are no β-redices left and there is nothing but pure function applications.

The most significant step performed in this normalization, is (paired) β-reduction. β-reduction means turning some expression $(\lambda x.P[x])\, a$ to $P[a/x]$, where x is substituted by a in P. Paired β-reduction means turning some expression $(\lambda(x_1, ..., x_n).P[x_1, ..., x_n])\,(a_1, ..., a_n)$ to $P[a_1/x_1, , ..., a_n/x_n]$.

2.4. 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 $\vdash\, g = g'$ (symmetry and transitivity of equivalence).

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.

2.5. An advanced conversion

The universal conversion does not exploit any knowledge about *how* the synthesis step was performed. However, one can think of an advanced conversion, where synthesis is performed by a sequence of conversions which are optimized for the scheduling step.

In principle, this conversion is similar to the universal conversion except that step 2 is tuned towards scheduling. The idea of our scheduling conversion is to split the data flow graph step by step rather than doing it all at once, as in the universal synthesis conversion. β-reduction is only applied to those variables whose corresponding nodes have been assigned to the current control step. Although some β-redices will remain, the terms achieved after normalization will be equal.

Other than in the universal synthesis conversion, $k-1$ conversions (k – number of control steps) have to be applied successively rather than applying one single conversion. Hence, the exponential complexity associated with step 2 is avoided and the overall cost is reduced.

To demonstrate the differences between the simple and the advanced conversion, we will show some experimental results in the following section.

2.6. Experimental results

We consider two scalable data flow graphs. As a first example, we use a data flow graph, which realizes the division of two polynomials (PD) with the given coefficients α_i and β_i:

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

The coefficients γ_i and δ_i should be computed. To facilitate the calculation, we assume that the divisor is normalized with respect to β_p. After a few algebraic transformations we get the following two formulas for computing the coefficients:

$$\begin{aligned} \gamma_i &= \alpha_{i+p} - \sum_{k=i+1}^{min\{i+p,q\}} \beta_{i+p-k} \cdot \gamma_k \qquad && i = 0 \ldots q \\ \delta_j &= \alpha_j - \sum_{k=0}^{min\{j,q\}} \beta_{j-k} \cdot \gamma_k \qquad && j = 0 \ldots p-1 \end{aligned}$$

The data flow graph consists of $p+q$ subtractors, $p(q+1)$ multipliers and $q(p-1)$ adders, so there is a total of $2pq+2p$ nodes. The critical path has a length of $3q+2$ nodes.

Another scalable data flow graph is realized in our second example. It calculates the discrete cosine transform (DCT), which is popularly 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 most applications, the parameters are set to $N = M = 8$. The number of additions is $2N^3 - N^2 - N$ and there are $2N^3 - N + 2$ multiplications. So there is a total of $4N^3 - N^2 - 2N + 2$ nodes. The length of the critical path is $2N + 1$. A more detailed description of this data flow graph can be found in [BlEK96].

In figure 2, the runtimes for the simple and advanced conversion are shown for the polynomial division. We set p to 25 and increased q. Due to the exponential memory consumption of the simple conversion, the computer's capacity of 1.2 GB was exceeded at about 600 nodes.

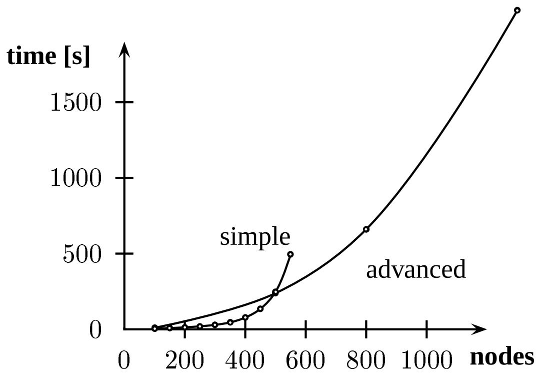

Figure 2. Simple and advanced conversion applied to PD

Figure 3 shows the runtimes for the two above mentioned data flow graphs. We applied different schedules that were derived by different scheduling algorithms namely ALAP and FD (force-directed [PaKn89]). For PD, p was again set to 25, and for the DCT, we increased N from 2 to 8. The structures of the data flow graphs PD and DCT differ in the depth and the number of reused intermediate results. As can be seen, the runtime depends on the structure of the DFG but is almost independent of the schedule table.

3. Changing the core of the theorem prover

In the following, we will present a modified HOL theorem proving system, named HOL', where we changed the core in order to increase efficiency. There are two modifications: the term representation is changed and also the core of basic logical transformations is enlarged.

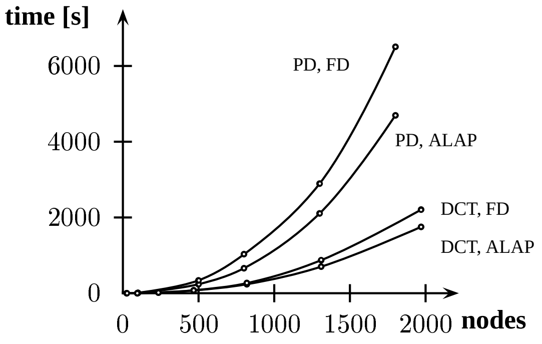

Figure 3. Advanced conversion for different data flow graphs and schedules

3.1. Changing the term representation

In the HOL theorem prover, terms are implemented in a so called *deBrujin*-style, which means that free and bound variables are represented differently. Free variables are stored with their name and type, whereas bound variables are only represented by a number, linking to the λ-abstraction it refers to. The number tells, how many λ-abstractions apart the referred λ-abstraction stands. Example: The variables x and y in the body of the term $\lambda x.(\lambda y.x\,y)$ are internally represented by 1 and 0, respectively. One advantage of the deBrujin representation is, that checking the α-equivalence of two terms can be performed in constant time.

However, the deBrujin term representation has a decisive drawback. Whenever one traverses a term while constructing or destructing it and reaches a bound variable, one has to go back to find the corresponding λ-abstraction for identifying the variable. Especially, if, as in our approach, variables are bound over a large range, this leads to a quadratic complexity. Therefore deBrujin term representation is not well suited for our application. However, this problem does not only arise in our formal synthesis approach. Whenever large circuits have to be represented, variables, which correspond to signals, are bound over a large range, and this means that deBrujin term representation becomes very inefficient.

Therefore we implemented HOL with a so called *name-carrying* term representation. Here, also the bound variables are stored with their name and type. It has the advantage that terms, which correspond to circuit descriptions, can be handled in a by far more efficient manner. On the other hand, some basic operations (e.g. α-equivalence check) become a bit more inefficient when using this representation style. Furthermore, it also increases memory consumption for representing bound variables, since instead of just a number, a string (for the name) and a type expression have to be stored.

3.2. Introduction of more efficient functions

As mentioned in section 2.3, β-reduction is applied very often during the normalization of the terms. HOL only allows one single β-reduction at a time. However one could 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 improve the efficiency of the theorem prover. This advanced β-conversion is equipped with a filter for selecting the bound variables that are to be expanded. This is important for our advanced conversion, where we only expand variables, that occur within the considered control step (see section 2.5). Based on the advanced β-conversion, paired β-reduction can be performed within one traversal. For the implementation of the advanced paired β-conversion we added a conversion, which changes paired β-redices into an unpaired representation. Example: $(\lambda(x, y).t)\,(a, b)$ is turned to $(\lambda x.(\lambda y.t))\,a\,b$. Afterwards, all the β-redices can be expanded in one traversal using the already mentioned advanced β-conversion.

The additional conversions are not only tailored to our application, but is of general interest for other users of the HOL system. However, it is to be noted, that enlarging the core is safety critical. Therefore, one has to consider carefully, which functions should be added to the core.

Figure 4 shows runtimes for a scheduling of the PD data flow graph scheduled with an ALAP technique. It shows, that both the simple and the advanced conversion runs faster under the modified HOL system (indicated by HOL'). Again, due to nested β-redices, the simple conversion ran into memory problems for larger data flow graphs.

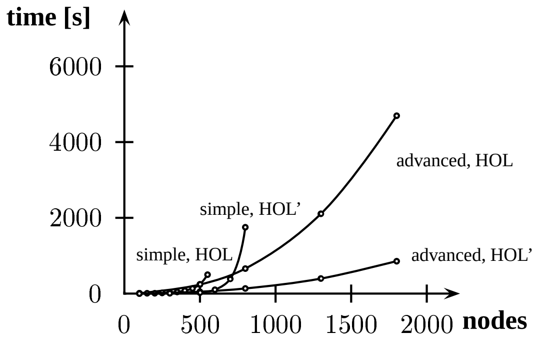

Figure 4. Simple and advanced conversion in different theorem provers

4. The formal synthesis scenario

Figure 5 demonstrates the underlying idea of our formal synthesis approach HASH illustrated by the scheduling step. Given a data flow graph, some scheduling heuristic is started. This heuristic step has nothing to do with logic. The heuristic returns a scheduling table which maps each operation in the data flow graph onto a control step. This scheduling table is now used by the formal logical transformation in HASH to produce a scheduled data flow graph.

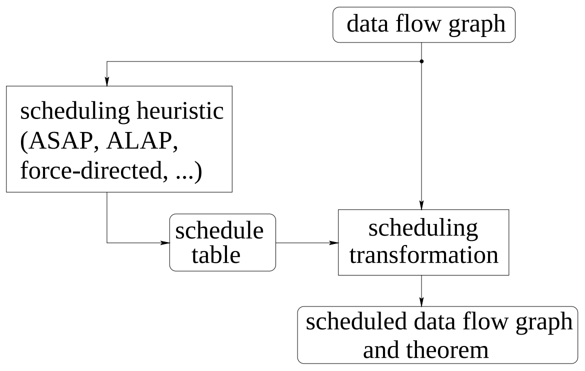
data flow graph
scheduling heuristic (ASAP, ALAP, force-directed, ...)
schedule table
scheduling transformation
scheduled data flow graph and theorem

Figure 5. Invoking synthesis heuristics within formal synthesis

The split between design space exploration (i.e. different schedule tables for different heuristics) and the logical transformation is the core idea in HASH. This core idea is applicable to most of the synthesis steps, e.g. allocation/no. of resources available, retiming/split in the combinational logic, etc.

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.

Since the entire synthesis process is nothing but a HOL conversion, correctness is guaranteed implicitly. Faulty implementations *cannot* be achieved even if the control information produced by the external program is flawed, such as a schedule where the data dependencies are disregarded. In such cases, the transformation cannot be performed within the logic and an exception will be raised. In conventional synthesis programs, such bugs could lead to faulty implementations. 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 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. In our scheduling step, for example, it is pretty easy to count the number of control steps and to check, whether it exceeds some given limit.

Figures 6 and 7 show the runtimes for both heuristics and transformational part of the formal synthesis step for the PD data flow graph and the DCT data flow graph, respectively (notice the different scales!). For determining the schedule table, we applied both the force-directed and the ALAP program. The circuit transformation was performed in HOL' (the modified HOL system) by applying the advanced conversion. As can be seen, the runtime for sophisticated design space exploration techniques such as force-directed scheduling can exceed 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.

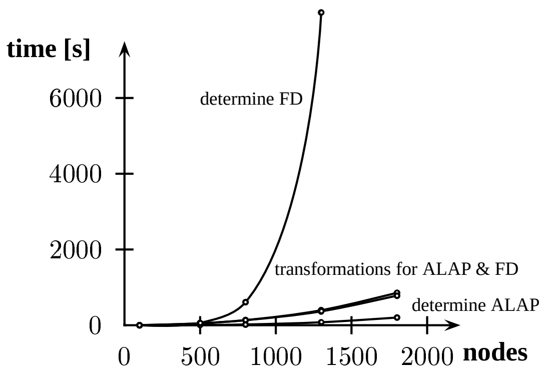

Figure 6. Time consumption for heuristic and transformation for PD

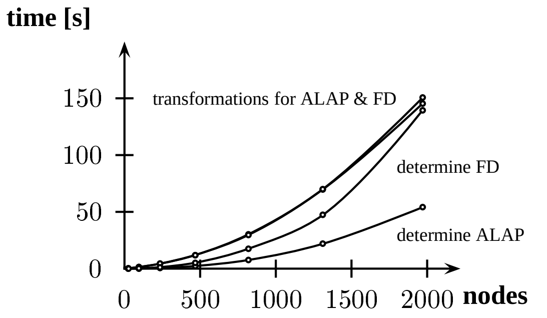

Figure 7. Time consumption for heuristic and transformation for DCT

5. Conclusions

For a given simple synthesis step, we have illustrated the efficiency problem when performing the 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 can be worthwhile to modify the basic logical transformations themselves. We have demonstrated, that formal refinement techniques for hardware synthesis can only be applicable in practice when considering these efficiency aspects. Efficient formal synthesis implementations, however, are applicable even for large sized circuits and the extra-costs for it, which are independent of the design space exploration part, are reasonable.

References

[BaFr96] D. Basin and S. Friedrich. Modeling a hardware synthesis methodology in isabelle. In [HOL96], pages 33–50.

[BlEK96] C.Blumenröhr, D. Eisenbiegler, and R.Kumar. Applicability of formal synthesis illustrated via scheduling. In *Workshop on Logic and Architecture Synthesis*, Grenoble, France, December 1996. Institut National Polytechnique de Grenoble.

[Busc92] H. Busch. Transformational design in a theorem prover. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design*, volume A-10, pages 175–196, Nijmegen, The Netherlands, June 1992. IFIP TC10/WG10.2 International Conference, North-Holland.

[CaWo91] R. Camposano and W. Wolf. *High-Level VLSI Synthesis*. Kluwer, Boston, 1991.

[Davi89] R. E. Davis. *Truth, Deduction and Computation: Logic and Semantics for Computer Science*. Computer Science Press, New York, 1 edition, 1989.

[EiBK96] D. Eisenbiegler, C. Blumenröhr, and R. Kumar. Implementation issues about the embedding of existing high level synthesis algorithms in HOL. In [HOL96], pages 157–172.

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

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

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

[GrMT94] W. Grass, M. Mutz, and W. Tiedemann. High level synthesis based on formal methods. In *Proc. EUROMICRO*, pages 83–91, Liverpool, 1994.

[Gupt92] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, 1992.

[HaLD89] F.K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In Luc J. M. Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, volume 2, pages 532–548. IMEC-IFIP, Elsevier Science Publishers, 1989.

[HOL96] Joakim von Wright, Jim Grundy, and John Harrison, editors. *Theorem Proving in Higher Order Logics:9th International Conference, TPHOLs'96*, number 1125 in Lecture Notes in Computer Science, Turku,Finland, August 1996. Springer-Verlag.

[JoBo91] S.D. Johnson and B. Bose. DDD - A system for mechanized digital design derivation. In *Workshop on Formal Methods in VLSI Design*, Miami, Florida, January 1991. ACM/SIGDA.

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

[Lars95] M. Larsson. An engineering approach to formal digital system design. *The Computer Journal*, 38(2):101–110, 1995.

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

[PaKn89] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer Aided Design*, 8(6):661–679, June 1989.

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

[TLWN90] 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.

[Wang92] L. Wang. Deriving a correct computer. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, volume A-20, pages 449–458, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland.