Applicability of Formal Synthesis Illustrated via Scheduling *

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

Ramayya Kumar

Forschungszentrum Informatik
(Prof. Dr.-Ing. D. Schmid),
Karlsruhe, Germany
kumar@fzi.de

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

Abstract

This paper describes a novel technique for formal synthesis and exemplifies the main ideas using the high level synthesis task — scheduling. The novelty of the approach is based on the fact, that arbitrary scheduling algorithms can be embedded within a formal framework to automatically achieve guaranteed correct implementations. Two realistic examples are used to emphasize its applicability and it can be seen that the additional costs for formal synthesis are almost negligible in practice. We achieve the same quality for the implementations as conventional synthesis plus the proof of their correctness.

1 Introduction

Although high level synthesis is based on a sequence of algorithms which conform to the "correctness by construction" paradigm, its implementation may be error-prone. This is due to the complexity of the programs [1] which implement these algorithms.

One approach towards proving the correctness of implementations is by post-synthesis verification. An excellent overview of verification techniques is given in [Gupt92, Melh93]. One of the important correctness criterions is to show that the implementation implies the specification. Two of the most important reasons for the complexity of these proofs are:

1. the existence of the major gap between the abstraction levels of the specification and the implementation and
2. the obliviousness of the information used in refining a specification into an implementation.

Therefore, full automation can only be achieved for comparatively small sized circuits at lower levels of abstraction. For large sized circuits, hardware verification specialists are mandatory. They have to either provide appropriate structuring and abstraction of the proofs, while using automatable logics, or perform logical interactions with the underlying theorem prover, while using complex logics.

Formal synthesis is a complementary approach to hardware verification, since formal averment is an integral part of the synthesis process. However, it is a specialized technique, which is only tailored towards the proof of synthesized implementations. Verification is nevertheless needed for validating specifications which can be achieved by checking properties such as safety and liveness.

We are developing a formal synthesis toolbox called HASH (Higher order logic Applied to Synthesis of Hardware) which is applicable to different abstraction levels. It contains one universal transformation per synthesis step, e.g. scheduling, allocation, retiming, state-minimization, etc. Each transformation is guided by the results of corresponding standard synthesis algorithms, that abound in literature [TLWN90, GDWL94]. Hence, no new synthesis

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

[1] The programs implementing the synthesis algorithms are mostly imperative in nature, and the correctness of large imperative programs is nearly impossible to prove.

algorithms (either formal or informal) are proposed, rather a general scheme for logically embedding various existing synthesis algorithms within a formal set-up is presented [EiKu95b]. In contrast to conventional synthesis approaches, only correct hardware implementations can be produced or no implementation is derived, when the results of the synthesis algorithms are faulty. The quality with respect to costs of the fully automatically generated implementations is dictated only by the conventional synthesis algorithms. The implementations therefore have a higher quality than those of conventional synthesis from an overall perspective , since they are proven to be correct. This concept will be elaborated with respect to the scheduling task in the sections to follow.

There are also other approaches in the formal synthesis domain. An overview is given in [KBES96]. But all other techniques do not exploit the results of the sophisticated algorithms which abound in synthesis ([FoMa89, HaLD89, JoBB88, ShRa95]). Therefore, the quality of their implementations is normally worse than that of conventional synthesis algorithms. In contrast to HASH which supports fully automated synthesis, all other approaches need interaction either at the schematic level or from a logician's point of view.

The major contributions of this paper are two-fold:

1. formal synthesis within HASH is applicable to realistic circuits, and
2. the additional costs for formal synthesis are reasonable.

The above-mentioned contributions are exemplified via the scheduling task in high-level synthesis.

The outline of this paper is as follows: in the next section, we briefly introduce our approach and define the notations and scope of our work. In section 3, we will show the results with two realistic examples and section 4 concludes the paper.

2 Our Formal Synthesis Approach towards Scheduling

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 algorithms for solving this task [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 underlying idea behind the scheduling transformation in HASH is illustrated in figure 1. 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 c-step. This scheduling table is now used by the formal logical transformation in HASH to produce a scheduled data flow graph. 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.

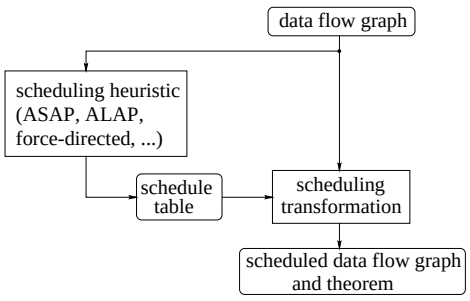

Figure 1: The concept of HASH as applied to scheduling

All the logical transformations in HASH have been implemented within the HOL theorem

prover [GoMe93]. Each transformation takes the current design state and the result of some synthesis heuristic and returns the new design state along with the correctness theorem, stating that the old design state is equivalent to (or implied by) the new design state.

Returning to the scheduling task, the formalization of the current design state, i.e. the data flow graph, is achieved by using λ-expressions [Davi89]. The data flow graphs are represented as follows:

$$\begin{array}{l}\lambda(x_1,\ldots,x_m).\\ \quad \mathsf{let}\ \langle outvars_1\rangle = op_1\langle invars_1\rangle\ \mathsf{in}\\ \quad \mathsf{let}\ \langle outvars_2\rangle = op_2\langle invars_2\rangle\ \mathsf{in}\\ \qquad\qquad\vdots\\ \quad \mathsf{let}\ \langle outvars_l\rangle = op_l\langle invars_l\rangle\ \mathsf{in}\\ \qquad (y_1,\ldots,y_n)\end{array}$$

The above structure describes the input/output function in terms of the basic operations in the data flow graph. $x_1, x_2, \ldots, x_m$ are the inputs, $y_1, y_2, \ldots, y_n$ the outputs and op_1, $op_2, \ldots, op_l$ the operations of the data flow graph. Each let-term describes the connectivity of one operation. For all i, $\langle invars_i\rangle$ and $\langle outvars_i\rangle$ denote the inputs and outputs of operation op_i, respectively. The inputs and outputs of operations are tuples, with each operation having the specific arity of its input and output tuple. This formal representation is however not unique, since the ordering of the operations is ambiguous. Nevertheless, the data dependencies between the operations must be respected.

The scheduling transformation in HASH takes the formalized data flow graph g and the schedule table and produces g' which is a composition of functions $g_1, g_2, \ldots, g_k$ such that $g' = g_k \circ \ldots \circ g_2 \circ g_1$ and k is the number of c-steps. Each g_i $(i = 1, \ldots, k)$ represents a slice in the original data flow graph g and corresponds to those operations that are executed in the i^{th} c-step. Additionally, the transformation produces the correctness proof stating the equivalence between g and g'. If the heuristic produces a false result (e.g. a schedule table where the data dependencies are violated or some operations are unscheduled), then the transformation fails and returns some constructive feedback to the user which reflects the cause of the failure.

In figure 2, a simple example is shown which illustrates the invocation of the scheduling transformation in HASH. In this example, a well-known heuristic called force-directed scheduling has been applied [PaKn89]. For better readability, the data flow graphs are shown in a schematic manner and not by their formal representation. If in this example the heuristic schedules operation 3 before operation 2, an exception will be raised during the transformation giving the constructive feedback that g' (figure 2) cannot be built with this schedule table.

It is also possible to combine several synthesis steps into one complex step. Then the corresponding logical transformations have to be performed one after another. The cost for this complex logical transformation is just the sum of the costs of the individual transformations (see [EiBK96] for more details about the transformations).

3 Experimental Results

In this section, we demonstrate that our formal synthesis scenario works with realistic examples. We therefore consider two scalable data flow graphs and compare the runtimes for calculating the schedule using various algorithms with the runtimes for the transformations, which produce a correct implementation. We cannot compare our work with any other verification results, since to our knowledge, no one has formally verified the scheduling task.

The scheduling algorithms we applied are ASAP (As Soon As Possible), ALAP (As Late As Possible), list-scheduling and two versions of force-directed scheduling (without/with look-ahead).

ASAP, ALAP and the two versions of force-directed scheduling do not enforce any constraints on the number of resources used. However, they always produce the shortest possible schedule. List-scheduling on the other hand works with a constrained number of resources but produces a schedule which is usually slower than those of the former approaches. The main idea behind the force-directed heuristic is to use the slack between the ASAP and ALAP schedules so as to distribute the operations in a better manner so that the resource utilization is also minimized in addition to the number of c-steps [PaKn89].

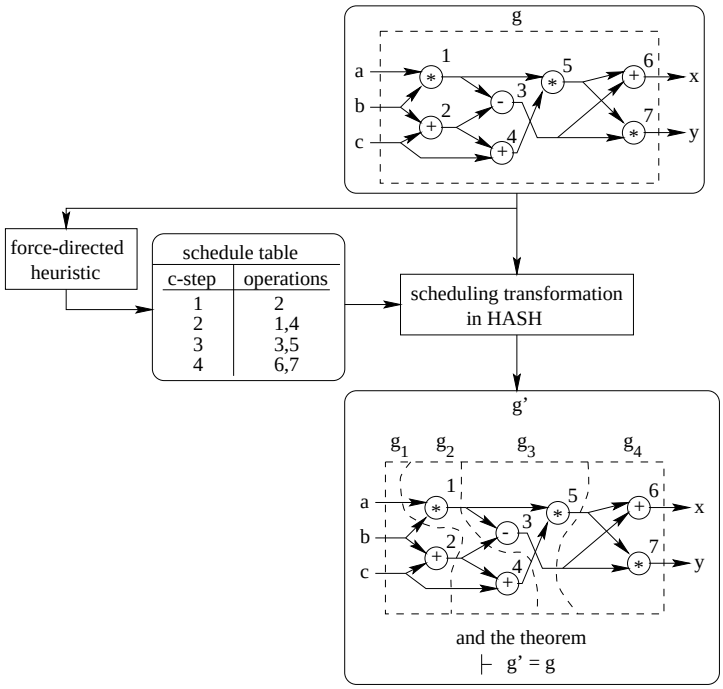

Figure 2: A simple example for the scheduling transformation in HASH

3.1 Division of two Polynomials

As a first example, we used a scalable data flow graph, which realizes the division of two polynomials with the given coefficients α_i and β_i:

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

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 the demanded coefficients:

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

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

Using these formulas, the data flow graph can be realized very quickly. To illustrate the underlying structure, a data flow graph with $p = 3$ and $q = 4$ is shown in figure 3.

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.

The runtimes [2] for the heuristics are shown in figure 4. The parameter p was always set to 25 and q was set to $1, 9, 15, 25, 35$. FD1 and FD2 correspond the two versions of the force-directed algorithm, and LS stands for list scheduling.

Irrespective of the variations in q, ASAP always needed 24 adders, 25 multipliers and 24 subtractors. ALAP always required 24 adders and 25 subtractors but the number of multipliers varied between 25 and 48. The two versions of the force-directed algorithm delivered either 24 adders, 24 multipliers and 25 subtractors or 24 adders, 25 multipliers and 24 subtractors.

[2] All experiments have been run a SUN ULTRA CREATOR with 196MB.

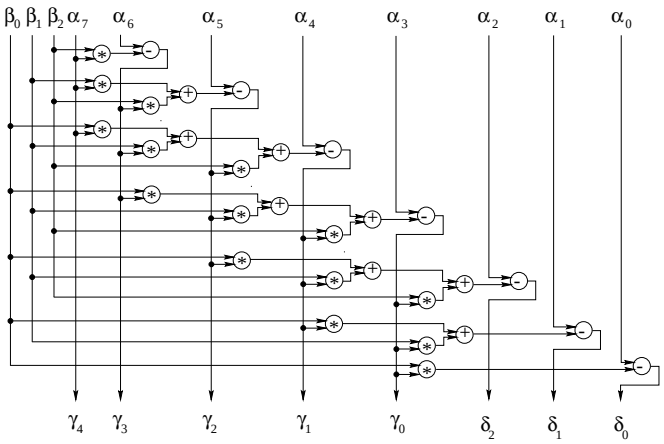

Figure 3: A data flow graph with p=3 and q=4

#Nodes	Heuristics				
	ASAP	ALAP	FD1	FD2	LS
100	0.1	0.1	0.2	0.3	0.3 / 5 + 10
500	2.0	5.3	60.5	67.7	19.2 / 29 + 36
800	6.8	20.3	610.5	641.2	74.2 / 47 + 55
1300	25.5	78.4	8260.8	8343.8	303.5 / 77 + 88
1800	64.0	203.3	47116.9	47289.3	763.0 / 107 + 120

Figure 4: Time for the heuristics

Although force-directed scheduling is a complicated algorithm which usually requires a lesser number of resources than ASAP or ALAP, it does not perform better in this example. This is because there is no better schedule, if the number of c-steps are minimized. On closer examination, one can detect, that one always needs $p-1$ adders and either $p-1$ multipliers and p subtractors or vice-versa (cf. from figure 3). The list-scheduling algorithm was restricted to 4 adders, 4 multipliers and 4 subtractors. The number of resulting c-steps is shown as sum of the c-steps for unconstrained scheduling and the additional c-steps for list-scheduling.

In figure 5 the runtimes for the transformations after the heuristics can be seen. The most interesting fact is that the runtime for the force-directed heuristic grows exponentially, whereas the runtime for its transformation does not; instead it grows in a polynomial fashion. Furthermore, the transformation is even faster than the heuristic for higher number of nodes and the intersection lies at about 1000 nodes. So it can be seen that the additional costs for formal synthesis can be negligible for large data flow graphs when compared with sophisticated heuristics! Additionally, it turns out that the runtime for the transformation is almost independent of the heuristic used. The only thing that matters is, how the heuristic distributed the nodes in the c-steps, not how long it took for that.

#Nodes	Heuristics				
	ASAP	ALAP	FD1	FD2	LS
100	12.1	9.8	11.6	11.7	19.4
500	247.9	238.4	339.6	333.2	443.2
800	647.0	660.4	1026.9	1035.9	1140.0
1300	1978.0	2106.9	2894.0	2881.5	3347.3
1800	4324.6	4693.4	6503.8	6442.7	7630.8

Figure 5: Time for the transformations

3.2 Discrete Cosine Transform (DCT)

Another scalable data flow graph is realized in our second example. It calculates the discrete cosine transform, 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 most cases, $N = M = 8$ is used. The data flow graphs are built as follows: The $N \cdot M$ pixels of the image are used as inputs. Furthermore, in order to ease the data flow graph, the cosine - terms are considered as additional inputs due to the complexity of the cosine - operation. In order to minimize the number of these additional inputs, one can exploit the periodicity of the cosine function. So the arguments can be restricted to the interval $[0, \pi]$. A restriction to the interval $[0, \frac{\pi}{2}]$ would also be possible, but then additional inverters will be necessary. If $N = M$, the following formula for the additional inputs due to cosine functions can be given as:

$$f(N) = \begin{cases} 0 & : \quad N = 1 \\ 5 & : \quad N = 3 \\ N + f(\frac{N}{2}) & : \quad N = 2, 4, \ldots \\ 2N + 1 & : \quad N = 5, 7, \ldots \end{cases}$$

If $N \neq M$, a formula cannot be given in a general manner. An additional reduction could be achieved, if $cos(\frac{\pi}{2})$ would be omitted, but then the data flow graph could not be built in a regular manner anymore.

Due to the definition of the DCT, there are still two factors to consider: $\frac{2}{\sqrt{N \cdot M}}$ and $\frac{1}{\sqrt{2}}$. The latter can be regarded as $cos(\frac{\pi}{4})$. So if N is even, this coefficient is already introduced as input. All in all, one has $N^2 + 1 + (N \, mod \, 2) + f(N)$ inputs for the data flow graph, if $N = M$. The number of outputs is $N^2 \quad [N \cdot M$, if $N \neq M]$.

To achieve a compact representation of the data flow graph, as many intermediate results as possible were reused. This leads to a total number of $2N^3 - N^2 - N \quad [N^2(M+1) + N(M^2 - 2M - 2) + M]$ additions and $2N^3 - N + 2 \quad [N^2(M+1) + N(M^2 - M - 1) + 2]$ multiplications. So there is a total of $4N^3 - N^2 - 2N + 2 \quad [2N^2(M+1) + N(2M^2 - 3M - 3) + M + 2]$ nodes. The length of the critical path is $2N + 1 \quad [N + M + 1]$.

To give a better idea of the structure, the data flow graph for $N = M = 2$ is shown in figure 6.

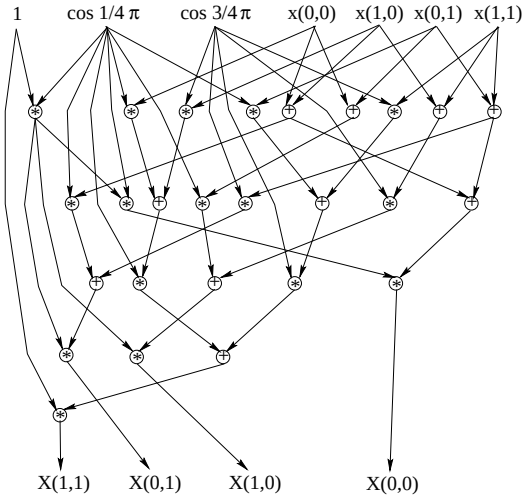

Figure 6: A data flow graph with N=2, M=2

In figure 7 the runtimes and required resources for the different heuristics are displayed. It should be noted that in this example, the number of resources required for force-directed

scheduling is always better than that of ASAP or ALAP. For the list-scheduling algorithm, we restricted the number of resources used to 8 adders and 8 multipliers. The number of resulting c-steps is shown as sum of the c-steps for unconstrained scheduling and the additional c-steps for list-scheduling.

#Nodes	Heuristics													
	ASAP			ALAP			FD1			FD2			LS	
	Time	#+	#*	Time	#+	#*	Time	#+	#*	Time	#+	#*	Time	#csteps
26	0.01	4	5	0.01	6	7	0.02	3	4	0.03	3	4	0.02	5+ 0
95	0.03	12	19	0.09	12	16	0.24	9	12	0.26	9	12	0.22	7+ 1
234	0.20	20	49	0.53	21	33	1.3	20	25	1.5	20	25	1.5	9+ 7
467	0.63	30	101	2.3	37	56	5.2	33	47	6.4	33	44	7.8	11+ 20
818	2.3	42	181	7.6	57	85	17.5	48	65	22.5	48	62	33.8	13+ 41
1311	6.1	56	295	21.9	81	120	47.3	66	86	64.0	66	86	112.9	15+ 71
1970	15.4	72	449	54.1	109	161	139.6	87	131	160.3	87	128	322.5	17+111

Figure 7: Time and resources for the heuristics

We investigated 7 data flow graphs by setting $N = M$ and varying their numbers from 2 to 8. One can see that the force-directed heuristic does not have an exponential behaviour, as in the previous example. This can be explained by a closer look at the data flow graphs. If we compare e.g. the DCT with $N = M = 5$ and the polynomial division with $p = 25, q = 9$, which have both nearly 500 nodes, one can see that 34% of the nodes in the DCT are placed immediately, since there is no difference between ASAP and ALAP (cf. brief description of force-directed scheduling in the introduction to section 3). In the polynomial division, only 19% are placed. Furthermore, the average movability of the remaining nodes is 2.2 for the DCT and 8.5 for the polynomial division. The maximal movability for the DCT is 8 and for the polynomial division it is 18. So it can be concluded that the operations in the division have more choices and the scheduling algorithm takes much longer.

In figure 8 the runtimes of the scheduling transformation for the different heuristics are shown. The conversions for ALAP, FD1 and FD2 are of the same magnitude. A special case is the transformation for the ASAP algorithm. Due to the nature of the data flow graph, many operations can be scheduled in the first c-steps by the ASAP, which can also be seen from the extremely high number of required resources in figure 7. This special constellation is very disadvantageous for the transformation algorithm. The transformation of the data flow graph with 1970 nodes was not possible due to space problems. But in most cases, especially when ingenious algorithms are used, the operations are better distributed in the schedule. Generally, one can see again that the runtime for the transformation is fairly independent from the heuristic, if the number of c-steps is equal. For list-scheduling the transformation takes longer due to the larger number of c-steps required.

4 Conclusions and Future Work

We have shown, that formal synthesis is not simply an academic dream, but can also be applied to realistic circuits. Additionally, the costs for formal synthesis are acceptable and are almost independent from the heuristics involved. In certain cases the design space exploration part can take much longer than performing the actual logical transformation, which in turn not only yields an implementation but also the proof of its correctness.

The novelty of HASH rests on the fact that in contrast to post-synthesis verification or other approaches for formal synthesis, we exploit the abundance of knowledge within the synthesis domain. The quality of the synthesis results produced, in terms of area, timing and power, are the same as that of conventional approaches. However, the correctness proof is an added quality! Yet another plus point in HASH is that, although a theorem-prover is used in the background, the entire procedure is automatic and no formal background is required on the part of the designer.

The major consequences that can be drawn from this work are that immense amounts of simulation/verification time can be saved and hence verification can be restricted to property checking. The time required for formal synthesis can be reduced even further, if the transformations are run either in the background or as a batch-process while the circuit designer concentrates on his job – the task of design exploration.

We have just discovered the tip of the iceberg and we still have a long way to go. In the future we shall concentrate on finding transformations for control-flow based scheduling algorithms, chaining of operations, pipelining, memory mapping, etc. We shall also provide

#Nodes	Heuristics				
	ASAP	ALAP	FD1	FD2	LS
26	0.5	0.5	0.5	0.5	0.5
95	3.6	3.8	3.7	3.7	3.9
234	19.4	17.8	18.8	18.7	26.9
467	91.5	73.0	77.2	75.9	154.6
818	365.5	237.4	265.5	254.7	668.9
1311	1259.5	702.0	874.0	882.8	2523.4
1970	–	1751.7	2204.8	2139.1	8130.8

Figure 8: Time for the transformations

links between the different levels of abstractions for the design of hardware (see [EiKu95] for application of HASH at RT-level).

References

[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 Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics:9th International Conference, TPHOLs'96*, number 1125 in Lecture Notes in Computer Science, pages 157–172, Turku,Finland, August 1996. Springer-Verlag.

[EiKu95] D. Eisenbiegler and R. Kumar. An automata theory dedicated towards formal circuit synthesis. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, number 971 in Lecture Notes in Computer Science, pages 154–169, Aspen Grove, Utah, USA, September 1995. Springer-Verlag.

[EiKu95b] D. Eisenbiegler and R. Kumar. Formally embedding existing high level synthesis algorithms. In Paolo E. Camurati and Hans Eveking, editors, *Correct Hardware Design and Verification Methods*, number 987 in Lecture Notes in Computer Science, pages 71–83, Frankfurt/Main, Germany, October 1995. IFIP WG10.5 Advanced Research Working Conference, Springer-Verlag.

[FoMa89] M.P. Fourman and E.M. Mayger. Formally Based System Design - Interactive hardware scheduling. In G. Musgrave and U. Lauther, editors, *Very Large Scale Integration*, pages 101–112, Munich, Federal Republic of Germany, August 1989. IFIP TC 10/WG10.5 International Conference, North-Holland.

[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.

[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.

[JoBB88] S.D. Johnson, B. Bose, and C.D. Boyer. A tactical framework for digital design. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383, Boston, 1988. Kluwer Academic Publishers.

[KBES96] 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, FMCAD'96*, Palo Alto, USA, 1996.

[Melh93] T. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, 1993.

[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 *CHDL95*, 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.