Implementation Issues about the Embedding of Existing High Level Synthesis Algorithms in HOL *

Dirk Eisenbiegler[1], Christian Blumenröhr[1] and Ramayya Kumar[2]

[1] Institute for Circuit Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid), University of Karlsruhe, Germany
[2] Forschungszentrum Informatik (Prof. Dr.-Ing. D. Schmid), Karlsruhe, Germany
e–mail: eisen@ira.uka.de, blumen@ira.uka.de, kumar@fzi.de

Abstract. This article describes the embedding of high level synthesis algorithms in HOL. For given standard synthesis steps, we describe, how its data can be mapped to terms in HOL and the synthesis process be expressed by means of a logical derivation. In contrast to post-synthesis verification techniques our approach is constructive in a sense that the proof is derived during synthesis rather than "guessed" afterwards. Therefore one does not get into the hardship of NP-completeness or undecidability. Our approach ensures correctness based on the HOL system and is also performed fully automatically.

1 Introduction

During the hardware design process of digital circuits, more and more complex tools are involved. Due to their complexity, guaranteeing the correctness of synthesis software is crucial. Bugs in the software may lead to incorrect hardware implementations.

One approach towards proving the correctness of implementations is by post-synthesis verification. An excellent overview of verification techniques is given in [Gupt92, Melh93]. However, full automation is only achievable for comparatively small sized circuits at lower levels of abstraction. For large sized circuits, verification algorithms either run into space/time hurdles or the user has to interact and perform some proofs by hand.

Formal synthesis is another approach towards hardware correctness. We consider formal synthesis as a derivation of the implementation from the specification by logical refinements.

We are developing a formal synthesis toolbox called HASH (Higher order logic Applied to Synthesis of Hardware) which exploits standard synthesis algorithms and is applicable to different abstraction levels. It is based on the HOL system, i.e. hardware is represented by means of HOL terms, and only rule applications are used to transform hardware descriptions. As opposed to conventional

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

synthesis tools, where there is no restriction on how to compute the implementation, our approach can only produce correct hardware implementations. The reliability of our synthesis conversions only depends on the correctness of the implementation of the HOL core and is independent from the complexity of the conversions. In this article, we will present the high level synthesis component of HASH.

Other approaches in the area of formal synthesis are [Lars94, AHL92, HaLD89, John84, JoSh90]. All these above-mentioned techniques have one common drawback, namely they do not exploit the knowledge of the algorithms which abound in synthesis. Additionally, the interactions to be performed during synthesis are at the schematic level or from a logician's point of view. The novelty of our current approach is that no new synthesis algorithms (either formal or informal) are proposed, but a general scheme for logically embedding various existing synthesis algorithms within a formal set-up is presented.

The outline of this paper is as follows: We will first describe the high level synthesis procedure in an informal manner (section 2). Then the logical representations and the logical transformations corresponding to the synthesis process are introduced in sections 3, 4 and 5. Afterwards, we will present some experimental results (section 6) and finally discuss the embedding of existing high level synthesis techniques (section 7).

2 Our High Level Synthesis Process

The starting point of our approach is a so called basic block. Basic blocks are data flow graphs describing the input/output relation by a composition of atomic operations. The timing of the atomic operations is static in a sense that they can be executed in fixed time (see figure 1). The functional relation represents a pure algorithmic description without any timing information.

The result of high level synthesis is a structure at the RT-level. Our synthesis process consists of the following steps: scheduling, register allocation and binding, allocation and binding of functional units. Interface synthesis will not be considered in this paper.

Our implementation[1] does not yet allow pipelining, instead all hardware resources (functional units as well as registers) will be reused during different clock ticks of one evaluation period.

Also the synthesis approach currently does not support any control flow. For more details on high level synthesis see [GDWL94, CaWo91].

Scheduling

Scheduling determines the number of control steps k needed for the evaluation of the algorithm and assigns each operation to one particular control step $0, 1, \ldots, k$

[1] Currently, no chaining of functional units and no multi-cycle operation units are used.

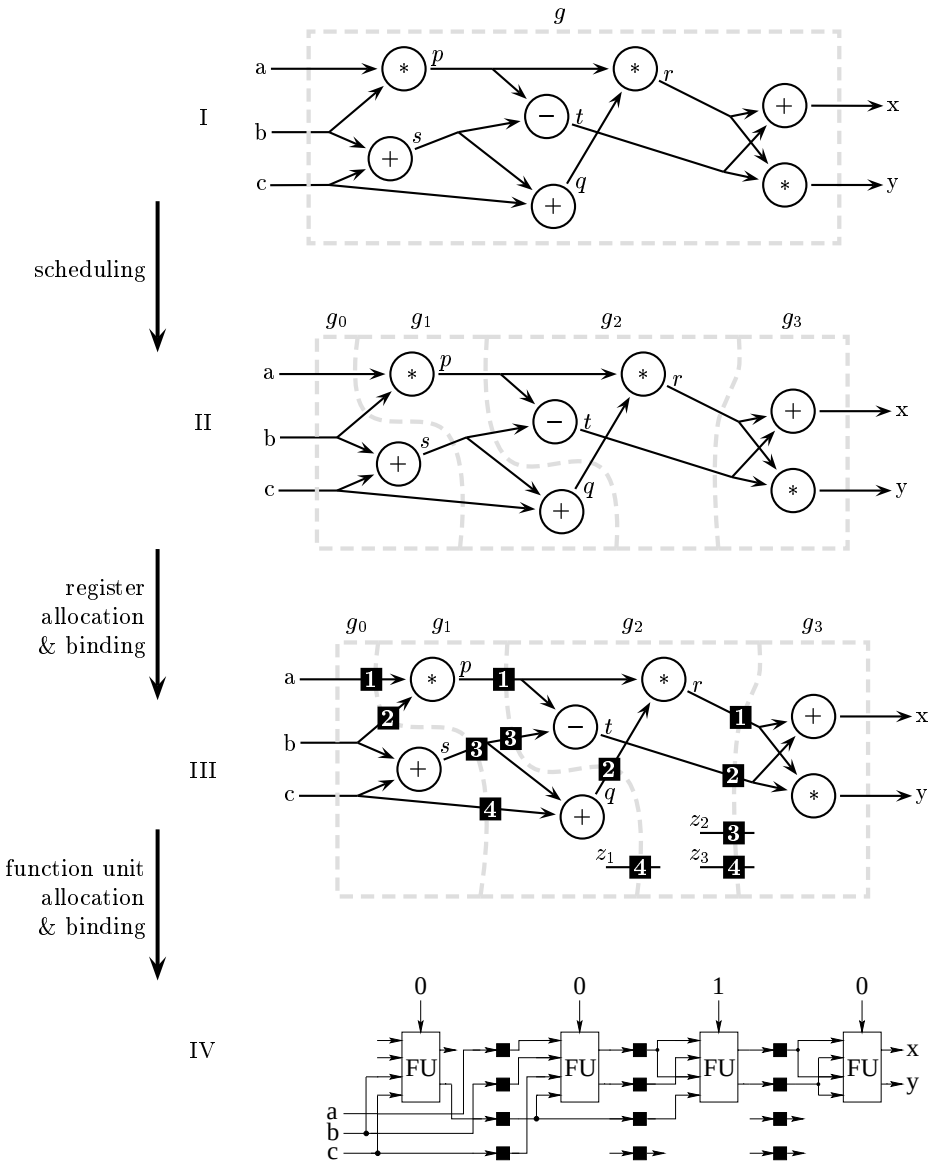

Fig. 1. High Level Synthesis Process

(see figure 1). There are mainly two costs, that have to be considered: the number of control steps k and the hardware resources required for implementing the operations. During scheduling, a trade off between the number of control steps k (speed of the implementation) and the hardware requirements (size of the implementation) has to be found.

Mainly there are two kinds of scheduling algorithms: ones with pregiven hardware constraints for the operation units and others with pregiven timing constraints. However, the implementation at the RT-level not only consists of operation units but also of communication units. The cost for these units can only be roughly estimated during the scheduling process. There are also advanced synthesis algorithms with their cost functions covering timing aspects as well as different hardware constraints. Such algorithms can be used to also handle sophisticated synthesis tasks. A schedule algorithm that is suitable for control flow paths is e.g. path-based scheduling [Camp91], whereas [PaKn89] introduces a possible schedule technique named force-directed only applicable to data flow graphs.

Register Allocation and Binding

The register allocation determines the number of registers needed for storing intermediate results between two control steps. The register binding determines a mapping between registers and auxiliary variables (intermediate results) for every control step.

In case there is only one single data type for all auxiliary variables, register allocation becomes trivial. The number of registers needed equals the maximum number of auxiliary variables between two control steps. In general, there may be auxiliary variables with different types. Different sizes of registers will be needed to store them. This makes register allocation more complex.

Register allocation and binding have an impact on the size needed for the communication parts between function units and registers. Good register bindings and allocations avoid additional hardware.

Function Unit Allocation and Binding

In this step, we construct a compound functional unit FU providing the operators for implementing the operations of each control step (allocation), and we use the compound functional unit FU to implement the operations of the data flow graph (binding). The function units are assumed to be given in a library. The library describes the mapping between its components and the operation(s) they can perform. There may be function units that are implementations of single operations as well as multi-purpose units with control input signals for selecting different operations. In our example, the function units consists of a multiplier implementing the $*$-operation and a multi-purpose unit implementing the $+$ and $-$ operation, where the operation is selected by a control signal having one of the values 0 and 1, respectively. Besides the functional aspects, the library also contains cost information such as area and power consumption.

3 Formal Representation of Data Flow Graphs

The efficiency of software strongly depends on the underlying data structures. In synthesis tools, suitable hardware representations have to be found. This also holds for our formal synthesis approach, where hardware is represented by means of HOL terms. In our approach, data flow graphs are represented as follows:

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

The above structure describes its input/output function in terms of its basic operations. $x_1, x_2, \ldots, x_m$ 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. let-terms are only used for a better readability of β-redices. 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 a specific arity of its input and output tuple.

Since these terms represent pure data flow graphs, i.e. no cycles are present, 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 dependency between the nodes is respected.

The following term gives an example for a data flow graph representation in HOL. The synthesis state in figure 1/I is formally represented as follows:

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

A constructor function named `mk_dfg` and a destructor function `dest_dfg` have been implemented. In ML, dfg's are represented with the following type:

```
type dfg =
  {
   inputs:term list,
   outputs:term list,
```

```
    operations :
      {operator:term, invars:term list, outvars:term list} list
  };
```

`mk_dfg` maps ML terms of type `dfg` to the corresponding HOL `term`. `dest_dfg` is the inverse function.

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. The synthesis states described in figures 1/II and 1/III are formally represented as follows:

$$\langle dfg_k \rangle \circ \ldots \circ \langle dfg_2 \rangle \circ \langle dfg_1 \rangle$$

During the allocation and binding of the function units, a compound function unit FU is introduced as an abbreviation. This abbreviation is described by means of a β-redex. The synthesis state described in figure 1/IV is represented as follows:

$$\begin{array}{l} \mathsf{let}\ FU\ =\ \langle dfg \rangle\ \mathsf{in} \\ \ \langle dfg'_k \rangle \circ \ldots \circ \langle dfg'_2 \rangle \circ \langle dfg'_1 \rangle \\ \mathsf{end} \end{array}$$

In this representation, each data flow graph $\langle dfg'_i \rangle$ consists of a single FU operator.

4 Transforming the Data Flow Graphs within HOL

This section describes, how the synthesis 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, the register-allocation table, etc.). 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 7.

The approach is based on a conversion for normalizing functions. We will first describe this conversion and then describe, how the synthesis steps are realized using this conversion.

Function Normalization

All HOL representations corresponding to figure 1 are 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 with the definition of $\circ$ (if there are any) and the function unit abbreviation is expanded (provided there is one)
3. β-reductions and paired β-reductions are performed wherever possible

In all cases, the result looks as follows:

$$\lambda(x_1, x_2, \ldots x_m).v[x_1, x_2, \ldots, x_m]$$

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

A Universal Conversion

We will now introduce a simple conversion which is applicable to all synthesis steps (figure 2).

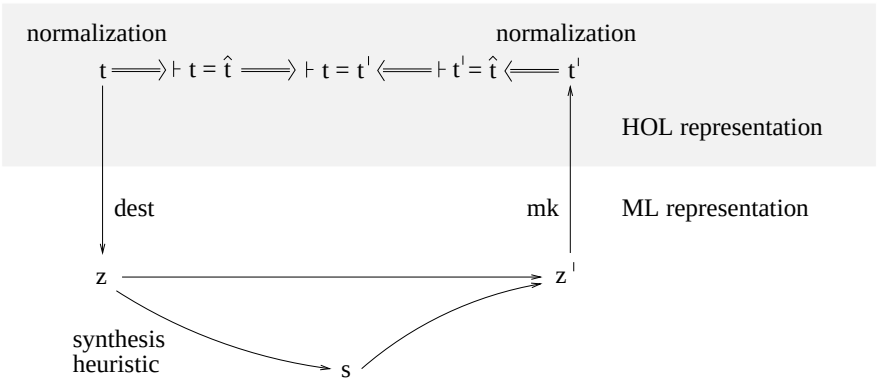

Fig. 2. Universal Conversion for all Synthesis Steps

1. The HOL term representation t is switched to its ML representation z. This is performed by applying some dest-function, which is based on `dest_dfg` (see section 3).
2. For the next step some external control information s (schedule, register allocation table, etc.) is required, which is produced by some arbitrary heuristic. According to s, z is then mapped to some new ML data structure z' corresponding to the result of the synthesis step under consideration. Step 2 is performed completely outside the logic.
3. The data structure z' is translated back to its HOL representation t'. This is performed by applying some mk-function, which is based on `mk_dfg` (see section 3).

4. Both t and t' are normalized by means of applying a normalization conversion. The results should be the same: $\vdash t = \hat{t}$ and $\vdash t' = \hat{t}$.
5. The equations $\vdash t = \hat{t}$ and $\vdash t' = \hat{t}$ are combined to $\vdash t = t'$ (symmetry and transitivity of equivalence).

The major drawback of this universal conversion is the complexity of step 4 when dealing with dfgs 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. Since such β-redices can be nested, the term size and time consumption in step 4 may grow exponentially with the depth.

The universal conversion not only works for single synthesis step, but it is also possible to combine several of our synthesis steps within step 2 of the conversion. Applying the universal conversion mechanism to the entire synthesis process reduces the time consumption since step 4 has to be performed only once rather than thrice (scheduling, register allocation & binding and FU allocation & binding).

5 An Advanced Conversion

The universal conversion is comparable to post-synthesis verification and does not exploit any knowledge about how the synthesis step was performed. In this section, we will describe an advanced conversion, where synthesis is performed by a sequence of conversions which are optimized for a *specific synthesis step*. Thereby, one can exploit the knowledge corresponding to this specific synthesis step. In principle, each of these conversions is similar to the universal conversion except that steps 2 and 4 are tuned towards a specific synthesis transformation. Although the advanced conversion is performed in several small parts, and therefore the technique described in section 4 has to be applied more often, the overall cost is reduced due to the remarkably lower cost for step 4 within each part.

The Scheduling Conversion

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 4 is avoided.

Figure 3 shows a HOL session performing the scheduling step applied to the example of figure 1. 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 (see section 7). For sake of readability, we used let-expressions rather than β-redices. `EXPAND_LETS_CONV` and `ABBREVIATE_LETS_CONV` have been applied to convert let-expressions to β-redices and vice versa.

```
- (
 EXPAND_LETS_CONV THENC
 (SCHEDULING_CONV force_directed) THENC
 UNEXPAND_LETS_CONV
)
   (--‘\(a,b,c).
      let p = a*b in
      let s = b+c in
      let q = s+c in
      let r = p*q in
      let t = p-s in
      let x = r+t in
      let y = r*t in
        (x,y)
      ‘--);
= = = = = = = = = = = = = = val it =
  |- (\(a,b,c).
       let p = a * b
       in
       let s = b + c
       in
       let q = s + c
       in
       let r = p * q
       in
       let t = p - s in let x = r + t in let y = r * t in x,y) =
     ((((\(r,t). let x = r + t in let y = r * t in x,y) o
       (\(p,q,s). let r = p * q in let t = p - s in r,t)) o
      (\(a,b,c,s). let p = a * b in let q = s + c in p,q,s)) o
     (\(a,b,c). let s = b + c in a,b,c,s) : thm
-
```

Fig. 3. HOL session performing a scheduling step

The Register Allocation and Binding Conversion

Register allocation and binding have one thing in common: they only have an effect on the interfaces between the slices. In our register allocation and binding conversion, the interfaces are changed step by step rather than all at once. The interfaces between $\langle dfg_i \rangle$ and $\langle dfg_{i+1} \rangle$ are changed by applying the universal synthesis conversion to $\langle dfg_i \rangle \circ \langle dfg_{i+1} \rangle$. Therefore in each step our universal synthesis conversion only has to be applied to a small subterm — the rest of the

term remains unchanged. Again $k-1$ applications are needed to do the job, but it pays out since the data flow graph considered is significantly smaller.

To be able to apply the interface changing conversion to all subterms $\langle dfg_i \rangle \circ \langle dfg_{i+1} \rangle$, the associative law of the $\circ$-operation has to applied. The number of the associative law rule applications needed in our implementation is $2(k-2)$.

The Function Allocation and Binding Conversion

Function allocation and binding only convert slices to equivalent ones and the *FU* abbreviation is performed. Therefore, besides expanding the *FU* abbreviation, one can apply our general synthesis conversion scheme to each slice separately. k steps are needed rather than one, but again the data flow graphs considered have a smaller depth.

6 Experimental Results

We used a scalable data flow graph as a benchmark. It 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 4.

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. In simplified terms, q controls the depth of the data flow graph whereas p determines the width.

We applied both the simple conversion presented in section 4 as well as the advanced conversion described in section 5. The runtimes[2] for the conversions

[2] All experiments have been run on a SUN ULTRA SPARC with 128MB.

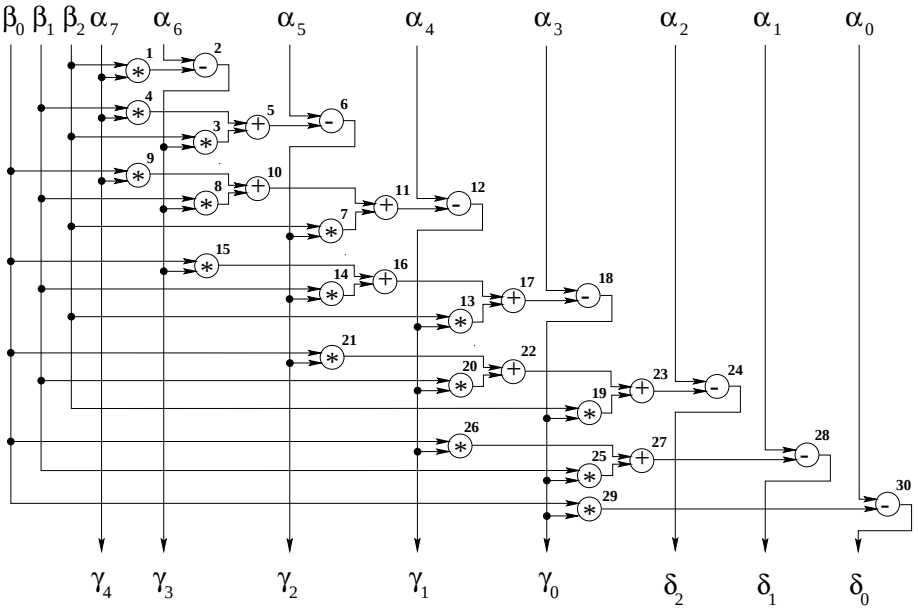

Fig. 4. A data flow graph with p=3 and q=4

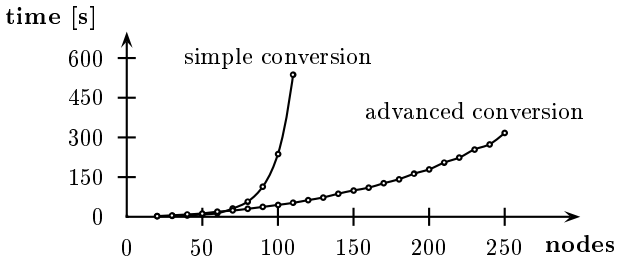

Fig. 5. Comparison simple/advanced conversion, $p = 5$, $q = 1, 2, \ldots$

are displayed in figure 5. It shows, that it pays out to interleave synthesis and logical derivation thereby exploiting the knowledge on how the implementation was derived, i.e. which synthesis steps have been applied and how they have been performed. The idea behind the technique of the simple conversion is pretty close to what one could do when performing post-synthesis verification. As can be seen in figure 4, some intermediate results ($\gamma_0 \ldots \gamma_q$) are used more often, which leads to an exponentially growth of β-redices in the universal conversion as shown in

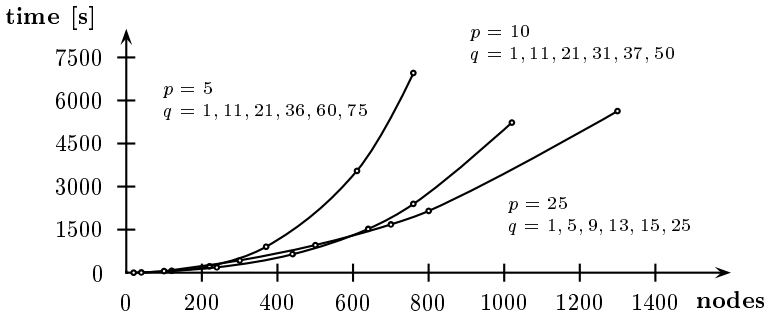

Fig. 6. Advanced conversion applied to DFGs with different p and q

section 4. During the conversion this results in an exponential consumption of both time and data storage. Therefore, the simple conversion is applicable only to very small sized circuits. In our example, the execution failed for bigger data flow graphs due to a lack of memory. The advanced conversion, however, did not run into space hurdles and could therefore also be applied to considerably bigger data flow graphs (see figure 6).

7 Embedding Existing High Level Synthesis Algorithms

The conversions described in the sections 4 and 5 are our basis for implementing synthesis tools in HOL. They are controlled by parameters telling them, how to perform the synthesis step (the schedule, the mapping between registers and variables etc.). Arbitrary heuristics can be invoked to compute this control information.

The heuristics invoked in section 6 have all been very primitive. For scheduling, a simple ASAP algorithm was used. Since the operands and results in all operations are of the same logical type, register allocation became trivial. The register binding was generated randomly — optimization aspects were not considered.

However, we also invoked more sophisticated synthesis heuristics. Table 7 shows different schedules achieved by different scheduling techniques. The schedules describe how the nodes (as numbered in the DFG in figure 4) are mapped to control steps. There are mainly two optimization goals for these algorithms: the number of control steps required and the number of operation units needed for the implementation.

In general, implementations with a big number of control steps can be realized with a small number of operation units whereas being restricted to a small number of control steps leads to a big number of operation units. There are mainly two kinds of scheduling algorithms: ones with hardware constraints and others with timing constraints. For a given restriction on the number of

operation units, scheduling algorithms with hardware constraints try to find a schedule with a minimal number of control steps. Scheduling algorithms with timing constraints are the other way around: for a given limitation on the number of control steps, the algorithm tries to find a schedule with a minimal number of hardware requirements.

The ASAP/ALAP algorithm (as soon/late as possible) assigns the nodes to the earliest/latest control step according to the restrictions given by the data dependencies. The force directed heuristic [PaKn89] tries to minimize the hardware by distributing it uniformly over the control steps. The heuristic is modeled after the calculation of the equilibrium for a set of springs and weights which obey the Hooke's law. The ASAP, the ALAP and the force directed scheduling algorithm do not place any restriction on the hardware and produce a schedule with a minimal number of control-steps. The (static) list scheduling heuristic [JMSW91] has a given restriction on the hardware consumption and tries to minimize the number of control steps needed according to a precalculated priority list.

In our example, the ASAP produced a schedule with a total of 7 operation units (3 multipliers, 2 adders and 2 subtractors), the result of the ALAP required 8 operation units (3 multipliers, 2 adders and 3 subtractors), and the force-directed algorithm required 7 operation units (2 multipliers, 2 adders and 3 subtractors). For the list scheduling algorithm, the number of multipliers was limited to 1, the number of subtractors was limited to 2 and the number of adders was also limited to 2. However, it required two extra control steps compared to the other techniques. According to our experiments, the time for the logical transformation is independent from the synthesis algorithm invoked: 5.97s for the ASAP, 5.72s for the ALAP, 5.78s for the force-directed and 6.15s for the list scheduling algorithm.

In our approach, a synthesis step can be divided into two parts: computation of the control information and execution of the transformation within the logic (figure 8). 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 is 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.

The optimization tasks corresponding to high level synthesis steps are very complex and mutually depend on one another. Thus heuristics have to be involved. The major advantage of our approach is, that we can exploit the existing

C-Step	Heuristics			
	ASAP	ALAP	Force-Directed	List Scheduling
1	1,4,9	1	1	1
2	2	2	2,4	2,4
3	3,18,15	3,4	3	9
4	5,10	5	5,15	3
5	6	6,8,9	6,8,9	5,8
6	7,14,21	7,10	7,10,14	6,10,15
7	11,16	11	11,16	7
8	12	12,14,15	12	11,14
9	13,20,26	13,16	13	12,16,21
10	17,22	17	17,21	13
11	18	18,20,21	18,20,26	17,20
12	19,25,29	19,22,25,26	19,22,25	18,22,26
13	23,27,30	23,27,29	23,27,29	19
14	24,28	24,28,30	24,28,30	23,25
15	–	–	–	24,27,29
16	–	–	–	28,30

Fig. 7. Control information derived by different scheduling algorithms, p=3, q=4

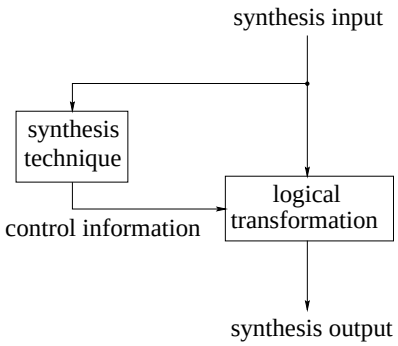

Fig. 8. The concept of our high level synthesis process

techniques. Our synthesis conversions offer the interface for embedding arbitrary conventional high level synthesis algorithms dedicated to the corresponding synthesis task. This has the effect, that – in contrast to most formal synthesis approaches — we do not have to invent new synthesis algorithms.

Although the conversions described in section 5 have to be performed in the given order, there is no restriction on how to compute the corresponding control information. It is possible to determine it step by step as sketched in the left side of figure 9 and one can as well determine it all at once as in the right side of figure 9. What really matters is that the control information is delivered to

the conversions in the given order — the order in which they are computed is ambiguous. Therefore, it is possible to embed arbitrary external synthesis algorithms. This aspect is of big importance since there is no limit as to the achievable quality of synthesis tools based on our approach.

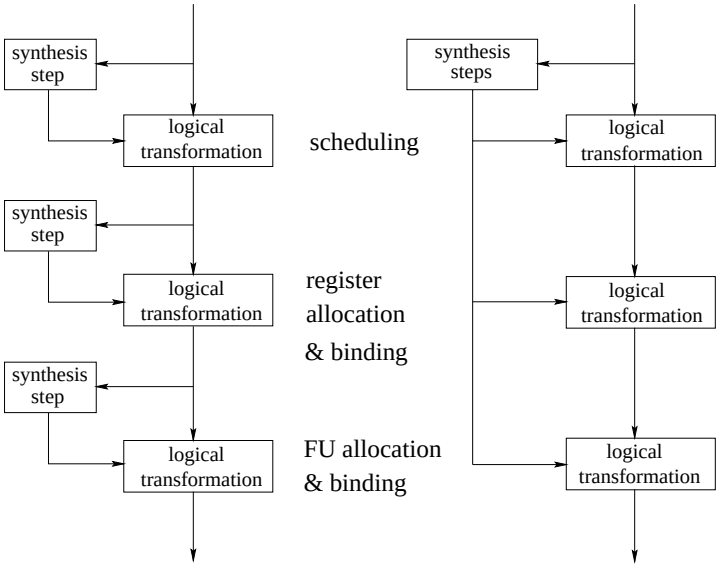

Fig. 9. Possibles schemes for the using of our synthesis conversions

8 Conclusion

We have described how high level synthesis can be performed by a sequence of logical transformations. The novelty of our approach lies in the exploitation of the existing knowledge in synthesis in a logically correct manner. As in conventional synthesis programs, finding suitable hardware representations and corresponding algorithms is essential for the efficiency. We have shown that it is possible to map algorithms and data of standard synthesis tools to logical conversions and representations in HOL.

Due to the expressiveness of HOL, general verification is an exacting goal. In our approach, however, the proof is constructed rather than "guessed" as in post-synthesis verification. Since our approach does not lead to NP-complete or undecidable problems, we believe, that formal synthesis is a well suited application for the HOL system.

In our recent work it turns out, that also in other abstraction levels of hardware design, formal synthesis can be a good alternative to the classical

synthesis/post-synthesis verification approach [EiKu95]. It is our intention to provide a formal synthesis toolbox called HASH containing formally based synthesis steps that cover the entire synthesis from the algorithmic level down to the logical level.

For the hardware designer, there is no difference between using synthesis tools based on HASH and conventional synthesis tools. However, formal synthesis guarantees correctness, implicitly. This style of formal synthesis will be acceptable to most users since they can proceed with their designs in a customary manner and yet have correctness without getting into the hardship of logic.

References

[AHL92] AHL. *Lambda Reference Manual*, 1989.

[Camp91] R. Camposano. Path-based scheduling for synthesis. *IEEE Transactions on Computer Aided Design*, 10(1):85–93, January 1991.

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

[EiKu95] D. Eisenbiegler and R. Kumar. An automata theory dedicated towards formal circuit synthesis. In *Higher Order Logic Theorem Proving and Its Applications*, Aspen Grove, Utah, USA, September 1995. Springer.

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

[Gupt92] A. Gupta. Formal hardware verification. *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 *IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, pages 532–548, Leuven,Belgium, 1989. Elsevier Science Publishers B.V.

[JMSW91] R. Jain, A. Mujumdar, A. Sharma, and H. Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *DAC '91*, pages 210–215, 1991.

[John84] S. Johnson. Synthesis of Digital Designs from Recursion Equations. MIT Press, 1984.

[JoSh90] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. North-Holland, 1990.

[Lars94] M. Larsson. An engineering approach to formal system design. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 300–315, Valetta, Malta, September 1994. Springer.

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

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

This article was processed using the LaTeX macro package with LLNCS style