Formally Embedding Existing High Level Synthesis Algorithms

Dirk Eisenbiegler and Ramayya Kumar

Forschungszentrum Informatik
(Prof. Dr.-Ing. D. Schmid)
Haid–und–Neustraße 10-14 76131 Karlsruhe, Germany
e–mail: eisen@fzi.de, kumar@fzi.de

Abstract. This paper introduces a general scheme for formally embedding high level synthesis by formulating its basic steps as transformations within higher order logic. A functional representation of a data flow graph is successively refined by means of generic logical transformations. Algorithms that are based on logical transformations guarantee "correctness by design". They not only construct an implementation but also derive the proof for its formal correctness, on the fly. An extra post-synthesis-verification step becomes obsolete. The logical transformations presented in this paper form a framework for formally embedding existing high-level-synthesis procedures.

1 Introduction

Guaranteeing functional correctness in hardware synthesis is an essential but demanding task. This is due to the complexity of synthesis tools and the underlying synthesis algorithms. Hence various forms of formal verification techniques are employed to prove the correctness of the implementations, resulting from the synthesis process [Melh93, ScKK93, Gupt92]. However the applicability of formal verification tools within the synthesis context is limited, since the proof of the goal "implementation $\Rightarrow$ specification" is very complex.

Post-synthesis verification is an exacting goal. Full automation can only be achieved for small sized circuits on 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.

Conventional synthesis algorithms just determine the implementation — the information on how the specification was refined into an implementation gets lost. The loss of this information is a major bottleneck for verification. The verification process gets just two logical formulae corresponding to the specification and the implementation. On the other hand, synthesis is split-up into a set of well-defined steps, namely scheduling, allocation and binding, and furthermore there exists a vast body of knowledge for solving these steps in an effective manner [CaWo91, Paul91, RoKr91]. We therefore propose a technique for "formal synthesis" which closely adheres to the steps of conventional synthesis and additionally exploits the knowledge available.

The idea of formal synthesis is in itself not new. One of the early attempts dealt with the conversion of regular expressions into hardware circuits [John84]. Later, a number of techniques were proposed for interactively refining the specifications into implementations [Lars94, HaLD89, JoWB89, AHL92, MaFo91, FoMa90]. All these above-mentioned techniques have one common drawback, namely they do not exploit the knowledge of the algorithms which abound in synthesis. 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 first briefly examine the synthesis problem and define the notations and scope of our work. Then we describe the formal techniques for scheduling, allocation and binding, respectively.

2 Basics of the "Formal Synthesis" Scenario

Starting from an algorithmic description, which does not incorporate timing explicitly, the overall aim of high level synthesis is to extract the data path and the controller. The major steps in synthesis are:

1. scheduling under restrained/unrestrained resource constraints
2. allocation
3. binding
4. determination of the RT-level implementation

2.1 Our Starting Point

The approach given in this paper deals with synthesis based on data-flow graph representations only. We represent the given data flow graph by a typed function g, and proceed with the various steps of synthesis. The sequential circuit corresponding to g will then repeatedly determine $g(x)$, for the various values of x. Since we use typed functions, a single input x is sufficient to represent any number of inputs corresponding to any type, since they can all be bundled together into a single x. The type definition uniquely determines the set of inputs and outputs. We will clarify this notion shortly.

2.2 An Example for g

Throughout this paper we shall illustrate the various steps of synthesis via an example named myg. myg maps a triple (a, b, c) onto the pair (x, y) as defined by the pseudo-procedural description in figure 1. Assuming that all the variables used are of the type natural numbers num, the overall type of the function is

$$\mathsf{num} \times \mathsf{num} \times \mathsf{num} \;\rightarrow\; \mathsf{num} \times \mathsf{num}$$

As basic operations there are the binary operations $+$, $-$ and $*$ and the unary operation inc. The operator inc maps some x to $x + 1$. Since the intermediate results are used within the succeeding expressions, they will be named explicitly. In our example they are named p, q, r, s and t. The data flow diagram which corresponds to the procedural code is given in figure 2.

```
Procedure myg(
  inputs: a,b,c:num;
  outputs: x,y:num)
begin
  p = a * b;
  q = inc(c);
  r = p * q;
  s = b + c;
  t = p - s;
  x = r + t;
  y = r * t;
end
```

Fig. 1. Pseudo procedural code for the example $g = \mathsf{myg}$

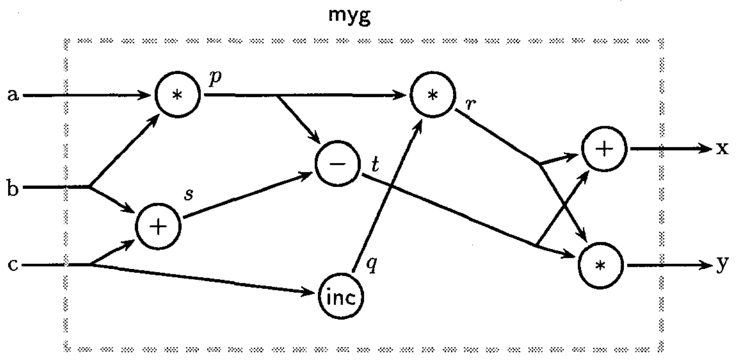

Fig. 2. Dataflow Diagram for myg

2.3 Remarks about the Notation

We will use λ-calculus expressions to denote functions (see [Davi89] for an introduction to the λ-calculus). let-terms will be used for representing β-redices. Let x be a variable, v be an arbitrary term having the same type as x and w denote an arbitrary term, where there may be free occurrences of x. In the expression

$$\text{let } x = v \text{ in } w$$

the variable x is used as an abbreviation for v in the expression w. The expression $w[v/x]$ is the expression, that can be obtained by substituting every occurrence of x in w by v. $w[v/x]$ is equivalent to the let-term above.

2.4 Formal Representation of g

Using let-terms to express the auxiliary variables in figure 2, g can be described by means of λ-abstraction over a tuple consisting of all inputs. In our example the input is a triple (a, b, c), there are 7 let-terms — one for each auxiliary variable — and there is a pair of outputs (x, y) (see equation (1)).

$$\boxed{\begin{array}{l} \vdash \ \mathsf{myg} = \\ \ \lambda(a,b,c). \\ \quad \mathsf{let}\ p = \ a * b \ \ \mathsf{in} \\ \quad \mathsf{let}\ q = \mathsf{inc}(c)\ \mathsf{in} \\ \quad \mathsf{let}\ r = \ p * q \ \ \mathsf{in} \\ \quad \mathsf{let}\ s = \ b + c \ \ \mathsf{in} \\ \quad \mathsf{let}\ t = \ p - s \ \ \mathsf{in} \\ \quad \mathsf{let}\ x = \ r + t \ \ \mathsf{in} \\ \quad \mathsf{let}\ y = \ r * t \ \ \mathsf{in} \\ \quad\ (x,y) \end{array}} \tag{1}$$

On comparing the pseudo-procedural code in figure 1 with the definition in equation (1), a direct one-to-one correspondence can be noticed. A little bit of formal syntactic sugaring yields the definition. This is true, if the pseudo procedural code consists of purely basic blocks.

2.5 The Formal Synthesis Scheme

Having defined the basics we will now proceed to give a gist of the overall formal synthesis scheme:

1. Convert the initial data flow graph into a functional representation.
2. Use an algorithm for scheduling, allocation or binding which performs the respective task on the data flow graph and gives us a schedule, allocation or binding, respectively.
3. Apply the pre-proven generic transformations for each task on the data flow graph along with the results of the algorithm.
4. Obtain a transformed function which is equivalent (in the logical sense) to the original description.
5. Derive the RT-level implementation from the transformed function.

Step 3 - the heart of the overall strategy, has been made possible by meticulous proofs of the generic transformations. They take in a function and the results of a specific synthesis step, and produce a new function which represents the end product of that specific synthesis step. Thus we are able to exploit all the optimizations that are offered by a particular algorithm and additionally, these transformations are automated and also not time-consuming. In the following sections, we shall show the transformed functions corresponding to the steps: scheduling, allocation and binding. The entire synthesis scheme will be implemented using the HOL theorem prover [GoMe93].

3 Scheduling

Scheduling determines the number of control steps (c-steps) for each calculation period[1] and assigns each operation to one particular c-step $0 \ldots n$. Given a specific n, the basic idea of the "schedule transformation" is to break up the original function g into a sequence of functions $g^0, g^1, \ldots g^n$, so that the composition of these functions yields the original function, i.e. $g = g^n \circ g^{n-1} \circ \ldots \circ g^0$.

In our example, there are 7 operations, whose outputs are the auxiliary variables p, q, r, s, t, x and y. We will also use the names of the auxiliary variables to denote the operation that produces this variable. Operation s, for example, is the first $+$ operation and the auxiliary variable s is its output.

Let us assume, that it is intended, that only two circuits are used: one multiplier and one multi-purpose unit for adding $+$, subtracting $-$ and incrementing inc. Under this hardware constraint several schedules are possible. Any arbitrary algorithm may determine the schedule.

In our example we will use the schedule sketched in figure 3: in c-step 0, s is processed, in c-step 1, p and q are processed, in c-step 2, r and t are processed and in c-step 3, x and y are processed. In this schedule n becomes 3. There are also schedules for myg with $n \neq 3$, but under the given restrictions, $n = 3$ is the minimum.

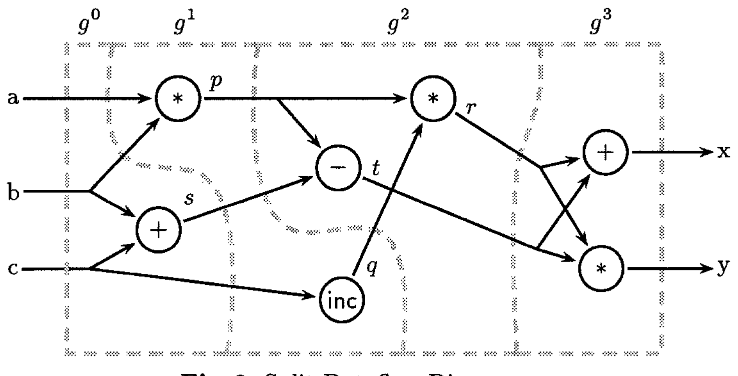

Fig. 3. Split Dataflow Diagram

The scheduled function myg will be described by means of a composition of four functions myg $= g^3 \circ g^2 \circ g^1 \circ g^0$, where the functions g^0, g^1, g^2 and g^3 perform the computations of c-step 0, 1, 2 and 3 respectively. The formal representation of the transformed function (theorem (2)) is derived by means of applications of the $\circ$ operator definition, expansion of let-expressions and by β-reductions.

It is easy to visualize that this transformation does not depend upon the scheduling algorithm itself, nor do we place any undue demands on the algorithm, except that it returns a schedule that obeys the data dependencies.

[1] It is also possible, that the number of c-steps is already given in the specification. Then only the assignment of operations has to be performed.

$$\begin{array}{ll} \vdash \mathsf{myg} = & \\ \mathsf{let} & \\ \quad g^0 = \lambda(a,b,c).\, \mathsf{let}\ s = b + c\ \mathsf{in}\ (a,b,s,c) & \mathsf{and} \\ \quad g^1 = \lambda(a,b,s,c).\, \mathsf{let}\ p = a * b\ \mathsf{in}\ \mathsf{let}\ q = \mathsf{inc}(c)\ \mathsf{in}\ (p,s,q) & \mathsf{and} \\ \quad g^2 = \lambda(p,s,q).\, \mathsf{let}\ t = p - s\ \mathsf{in}\ \mathsf{let}\ r = p * q\ \mathsf{in}\ (r,t) & \mathsf{and} \\ \quad g^3 = \lambda(r,t).\, \mathsf{let}\ x = r + t\ \mathsf{in}\ \mathsf{let}\ y = r * t\ \mathsf{in}\ (x,y) & \\ \mathsf{in} & \\ \quad g^3 \circ g^2 \circ g^1 \circ g^0 & \end{array} \tag{2}$$

4 Allocation of Registers

The register allocation determines the number of registers that is needed for the implementation. Usually the scheduling algorithms already take the functional resource constraints into account.

When functions $g^0, g^1, \ldots g^n$ are composed in the mathematical world, the output of a function g^j is the input of the function g^{j+1}, given $0 \leq j < n$. However, in the hardware context, registers are needed to store the values between two control steps. The total number of registers always equals the maximum number of outputs produced by any g^j, $0 \leq j < n$. This implies that when any of the functions g^j have lesser number of outputs, then some extra variables are added to the outputs of g^j and the inputs of g^{j+1}, so that the overall number is m. These variables can carry any arbitrary values since they are never used.[2]

In our example four variables have to be buffered after c-step 0, three after c-step 1 and two after c-step 2. Therefore, we add one auxiliary variable z^1 after c-step 1 and two auxiliary variables z^2 and z^3 after c-step 2 (see figure 4). The formal representation is given in theorem (3).

5 Binding of Registers

During register binding, variables are tied onto specific registers. The register binding is represented as the ordering of the variables within the tuples — i.e. register binding will be formally expressed by giving the variables a specific order.

In our example four registers are needed. They are named r^1, r^2, r^3 and r^4. After c-step 0 they are used for storing a, b, s and c, after c-step 1 they are used for storing p, s, q and z^1 and after c-step 2 they are used for storing r, t, z^2 and z^3. The mapping between the variables and the registers has to be optimized in order to avoid unnecessary variable transfers between registers. Such optimizations can be done by conventional synthesis algorithms outside the logic, and then be integrated within our formal synthesis environment.

The determination of register binding is performed again outside the logic. The result of register binding is a table describing the mapping between variables

[2] In general, there may be auxiliary variables with different types. Different sizes of registers will be needed to store them and optimization during register allocation may become more complex.

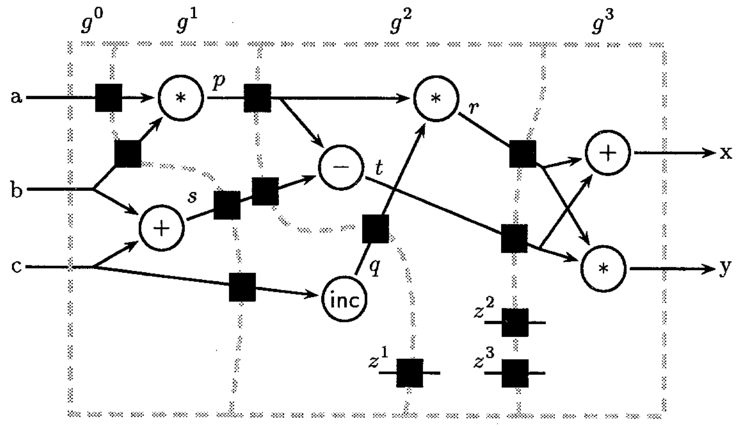

Fig. 4. Allocation of Registers

$$\begin{aligned}
&\vdash \mathsf{myg} = \\
&\mathsf{let} \\
&\quad g^0 = \lambda(a,b,c).\ \mathsf{let}\ s = b + c\ \mathsf{in}\ (a,b,s,c) \quad \mathsf{and} \\
&\quad g^1 = \lambda(a,b,s,c).\ \mathsf{let}\ p = a * b\ \mathsf{in}\ \mathsf{let}\ q = \mathsf{inc}(c)\ \mathsf{in}\ (p,s,q,z^1) \quad \mathsf{and} \\
&\quad g^2 = \lambda(p,s,q,z^1).\ \mathsf{let}\ t = p - s\ \mathsf{in}\ \mathsf{let}\ r = p * q\ \mathsf{in}\ (r,t,z^2,z^3) \quad \mathsf{and} \\
&\quad g^3 = \lambda(r,t,z^2,z^3).\ \mathsf{let}\ x = r + t\ \mathsf{in}\ \mathsf{let}\ y = r * t\ \mathsf{in}\ (x,y) \\
&\mathsf{in} \\
&\quad g^3 \circ g^2 \circ g^1 \circ g^0
\end{aligned} \tag{3}$$

and registers and this table is the basis for our next logical transformation step. Let us assume, that the register binding of table 1 is to be applied.

Registers	after c-step 0	after c-step 1	after c-step 2
r^1	a	p	r
r^2	b	q	t
r^3	s	s	z^2
r^4	c	z^1	z^3

Table 1. Register Binding

From now on, we will not use the auxiliary variable names $p, q, r, \ldots$ any more but replace them by register names $r^1, r^2, r^3, \ldots$. In each of the functions $g^0, g^1, \ldots$ the names $r^1, r^2, r^3, \ldots$ are used to represent the register values before the evaluation of the function and $r^{1'}, r^{2'}, r^{3'}, \ldots$ are used to indicate the register values after the evaluation of the function. Variable renaming is performed by α-conversion (see [Davi89]). The formal representation of the result of the register binding in theorem (4) is achieved by expansion of the $\circ$ operators and let-expressions and β-reductions.

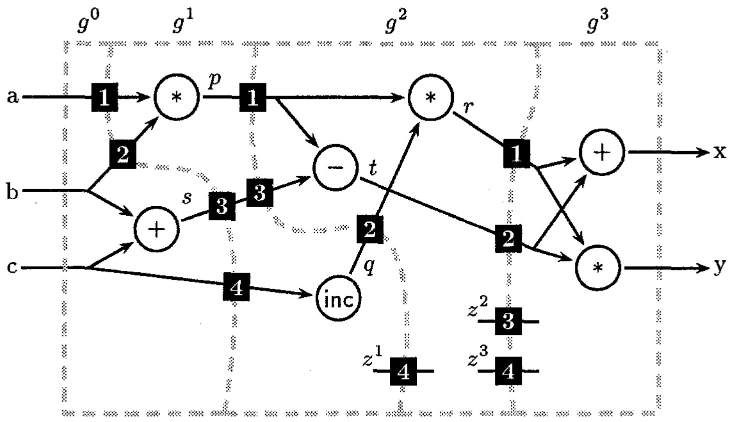

Fig. 5. Binding of Registers

$$
\begin{array}{l}
\vdash \mathsf{myg} = \\
\quad \mathsf{let} \\
\qquad g^0 = \lambda(a,b,c). \\
\qquad\quad \mathsf{let}\ r^{1'} = a\ \mathsf{and}\ r^{2'} = b\ \mathsf{and}\ r^{3'} = b + c\ \mathsf{and}\ r^{4'} = c\ \mathsf{in} \\
\qquad\qquad (r^{1'}, r^{2'}, r^{3'}, r^{4'}) \\
\quad \mathsf{and} \\
\qquad g^1 = \lambda(r^1, r^2, r^3, r^4). \\
\qquad\quad \mathsf{let}\ r^{1'} = r^1 * r^2\ \mathsf{and}\ r^{2'} = \mathsf{inc}(r^4)\ \mathsf{and}\ r^{3'} = r^3\ \mathsf{and}\ r^{4'} = z^1\ \mathsf{in} \\
\qquad\qquad (r^{1'}, r^{2'}, r^{3'}, r^{4'}) \\
\quad \mathsf{and} \\
\qquad g^2 = \lambda(r^1, r^2, r^3, r^4). \\
\qquad\quad \mathsf{let}\ r^{1'} = r^1 * r^2\ \mathsf{and}\ r^{2'} = r^1 - r^3\ \mathsf{and}\ r^{3'} = z^2\ \mathsf{and}\ r^{4'} = z^3\ \mathsf{in} \\
\qquad\qquad (r^{1'}, r^{2'}, r^{3'}, r^{4'}) \\
\quad \mathsf{and} \\
\qquad g^3 = \lambda(r^1, r^2, r^3, r^4). \\
\qquad\quad \mathsf{let}\ x = r^1 + r^2\ \mathsf{in}\ \mathsf{let}\ y = r^1 * r^2\ \mathsf{in} \\
\qquad\qquad (x, y) \\
\quad \mathsf{in} \\
\qquad g^3 \circ g^2 \circ g^1 \circ g^0
\end{array}
\tag{4}
$$

6 Allocation and Binding of Functional Units

In this step of the algorithm, we construct a compound functional unit *FU* providing the operators for implementing the operations of each c-step (allocation), and we use the compound functional unit *FU* to implement the operations of the dataflow graph (binding).

As already mentioned earlier, only two operation units are needed in our example: one multiplier (named multiplier) and one multi-purpose unit (named multipurpose) for adding, subtracting and incrementing. Their formal specifications are given as below. Such descriptions are assumed to be given in a library which defines the abstract RT-level components. The correctness of such components is beyond the scope of this paper and can be performed using conventional verification techniques.

$$\vdash \mathsf{multiplier}(a,b) = a * b$$

$$\begin{aligned}\vdash\ & \mathsf{multipurpose}((d,e),\mathsf{Add}) = d+e \\ & \mathsf{multipurpose}((d,e),\mathsf{Sub}) = d-e \\ & \mathsf{multipurpose}((d,e),\mathsf{Inc}) = d+1\end{aligned}$$

The multi-purpose unit has $((d,e),c)$ as input. d and e are data inputs and c is a control input for selecting the function. c may have one of the values Add, Sub and Inc and the corresponding output is $d+e$, $d-e$ and $d+1$, respectively.

In theorem (5) the functional unit *FU* is provided. It consists of one multiplier and one multi-purpose unit. It's input $(((a,b),(d,e)),c)$ consists of two parts: a data input $((a,b),(d,e))$ and a control input c. The result is a pair consisting of the product of a and b and the result of applying d and e to the multi-purpose unit, where the control of the multi-purpose unit is c.

In general there may be several operations of each type and optimizations in the binding between operations and functional units may reduce communication costs. In our small example the binding is unambiguous, since in each c-step there is always no more than one operation of each type.

Remark: In c-step 0, the multiplier unit remains unused. Arbitrary values z^5 and z^6 are its input and the output (named z^7) is not connected to the output of g^0. Since there is only one operand needed during the inc-operation in c-step 1, one of the data inputs becomes redundant. An arbitrary value z^8 is assigned to this input.

Theorem (5) is derived by applying the definitions of FU and the specifications of the multiplier and the multipurpose component.

7 Derivation of the RT-Level Implementation

This section describes, how the preprocessed algorithmic description is converted into a RT-level description. Before this step can be performed, we must describe the temporal relationships between the algorithmic and RT-levels, i.e. we must describe, how the circuit evaluating g interfaces with its environment. We will call these relations as communication schemes.

7.1 Communication Schemes

The datapath oriented synthesis algorithm described until now is an adequate basis for deriving implementations for different kinds of simple communication schemes. All hardware descriptions, that may be implemented by our approach must have a fixed number of c-steps $0 \dots n$ for each evaluation cycle. In c-step 0, the circuit reads x from the input i, in the succeeding c-steps it calculates $g(x)$ and at c-step n it assigns $g(x)$ to the output o.

We present two possible communication schemes describing the behavior of the circuit in an entirely different manner.

```
⊢ myg =
 let
  $FU = \lambda(((a,b),(d,e)),c).\ (\mathsf{multiplier}(a,b),\ \mathsf{multipurpose}((d,e),c))$
 in
 let
  $g^0 = \lambda(a,b,c).$
   let $(z^6, r^{3'}) = FU(((z^5,z^6),(b,c)),\mathsf{Add})$ and
      $r^{1'} = a$ and $r^{2'} = b$ and $r^{4'} = c$ in
      $(r^{1'}, r^{2'}, r^{3'}, r^{4'})$
 and
  $g^1 = \lambda(r^1,r^2,r^3,r^4).$
   let $(r^{1'}, r^{2'}) = FU(((r^1,r^2),(r^4,z^8)),\mathsf{Inc})$ and
      $r^{3'} = r^3$ and $r^{4'} = z^1$ in                                        (5)
      $(r^{1'}, r^{2'}, r^{3'}, r^{4'})$
 and
  $g^2 = \lambda(r^1,r^2,r^3,r^4).$
   let $(r^{1'}, r^{2'}) = FU(((r^1,r^2),(r^1,r^3)),\mathsf{Sub})$ and
      $r^{3'} = z^2$ and $r^{4'} = z^3$ in
      $(r^{1'}, r^{2'}, r^{3'}, r^{4'})$
 and
  $g^3 = \lambda(r^1,r^2,r^3,r^4).$
   let $(x,y) = FU(((r^1,r^2),(r^1,r^2)),\mathsf{Add})$ in
    $(x,y)$
 in
  $g^3 \circ g^2 \circ g^1 \circ g^0$
```

Specification A: self-starting evaluation The circuit starts its first computation cycle at time 0. It immediately restarts a new computation whenever the old calculation cycle has finished. Such a circuit will always be busy.
Formalization:

$$\mathsf{specA}(g,n,i,o) \ = \ (\ \forall x.\ o((n+1)*(x+1)-1) = g(i((n+1)*x))\ )$$

Remark: g s the function to be evaluated, n is the number of c-steps, i is the input and o is the output. Given such a formalization, the overall specification can be written as:

$$\exists n.\ \mathsf{specA}(g,n,i,o)$$

Specification B: event-driven evaluation At time 0, the circuit starts in a nonbusy state. The circuit begins a computation cycle whenever it is not busy and gets a specific stimulus from an input signal *start*.
Requirements:

– the circuit is not busy at time 0
– if at time t the circuit is not busy and there is no *start* signal at time t, then the circuit will not be busy at time $t+1$.
– if at time t the circuit is not busy and there is a *start* signal at time t, then the circuit will be busy during $[t+1, t+n]$, produce the required output at time $t+n$ and be ready for new input at time $t+n+1$.

Formalization:

$$
\begin{aligned}
&\mathsf{specB}(g,n,i,o) \; = \\
&\exists n. \\
&\quad (\, \neg busy(0) \,) \wedge \\
&\quad (\, \forall t.\, \neg busy(t) \wedge \neg start(t) \Rightarrow \neg busy(t+1) \,) \wedge \\
&\quad (\, \forall t.\, \neg busy(t) \wedge start(t) \Rightarrow \\
&\quad \forall m : t+1 \leq m \leq t+n.\ busy(m) \wedge \\
&\quad o(t+n) = g(i(t)) \wedge \\
&\quad \neg busy(t+n+1) \,)
\end{aligned}
$$

The overall specification for such circuits is:

$$\exists n.\ \mathsf{specB}(g,n,i,o)$$

7.2 Implementation Templates

For a given function g and a communication scheme such as $\mathsf{specA}(g,n,i,o)$ or $\mathsf{specB}(g,n,i,o)$ an RT-level implementation is to be derived. It is assumed that g has already been preprocessed according to the synthesis steps described in sections 3 through 6 so that g has the form given in (5).

It is not our intention to try and find an implementation and prove its correctness whenever we have successfully processed synthesis steps 3 through 6. Instead we use generic implementation descriptions for given communication schemes and prove a theorem stating that the generic implementation descriptions fulfills the communication scheme. During synthesis, this theorem is just instantiated and thereby the correct implementation is derived from the specification.

For lack of space, we cannot give a complete description on how the implementation is described within higher order logic and how the correctness proof is performed. Figure 6 gives a sketch of how a general implementation of the specification $\mathsf{specA}(g,n,i,o)$ looks like. It is assumed that g has the shape as in (5). The controller is a simple modulo-n-counter with n being the number of c-steps. In the middle there is the functional unit as described in section 6. The MUX-circuits on the left and right of the functional unit determine the data flow between input, output, registers and functional unit according to the c-step the circuit is in. Since there may also be multi-purpose units within the functional unit, the operations to be performed may also depend on the c-step and is therefore steered by the controller.

8 Conclusion

We have described, how high level synthesis can be performed by a sequence of logical transformations. Starting from the functional descriptions of data flow graphs, we have been able to successfully refine them into an RT-level hardware description. The novelty of our approach lies in the exploitation of the existing knowledge in synthesis in a logically correct manner.

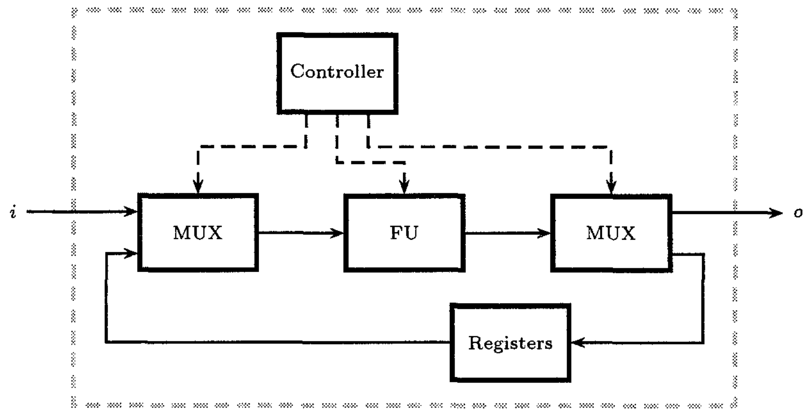

Fig. 6. Abstract implementation for general type A specifications

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. In the post-synthesis verification approach, however, the proof has to be "guessed" rather than constructed by derivation.

We have shown, that formal synthesis is an appropriate approach in high level synthesis and that it is possible to formally embed existing synthesis algorithms. We believe, that also in other areas of hardware design, formal synthesis can be a good alternative to the classical synthesis/post-synthesis-verification approach. Still, our approach of formally embedding the synthesis process has only been applied to particular synthesis algorithms in order to prove its applicability. It is our intention to provide a formal synthesis toolbox containing formally based synthesis procedures that cover the entire synthesis from the algorithmic level down to the logical level. We still have a long way to go, but we believe, that we have an interesting starting point.

References

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

[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.

[Day92] Nancy Day. A comparison between statecharts and state transition assertions. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 247–262, Leuven, Belgium, November 1992. North-Holland.

[FoMa90] M.P. Fourman and E.M. Mayger. Formally Based System Design - Interactive hardware scheduling. In G. Musgrave and U. Lauter, editors, *International Conference on Very Large Scale Integration*, pages 101–112. Elsevier Science Publishers (North-Holland), 1990.

[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. *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.

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

[JoWB89] S.D. Johnson, R.M. Wehrmeister, and Bhaskar Bose. On the interplay of synthesis and verification. In *IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, pages 385–404, Leuven,Belgium, 1989. Elsevier Science Publishers B.V.

[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.

[Loew92] Paul Loewenstein. A formal theory of simulations between infinite automata. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 227–246, Leuven, Belgium, November 1992. North-Holland.

[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.

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

[Paul91] P. G. Paulin. Global Scheduling and Allocation Algorithms in the HAL System. In R. Camposano and W. Wolf, editors, *High-Level VLSI Synthesis*, pages 255–281. Kluwer Academic Publishers, 1991.

[RoKr91] W. Rosenstiel and H. Krämer. Scheduling and Assignment in High Level Synthesis. In R. Camposano and W. Wolf, editors, *High-Level VLSI Synthesis*, pages 355–382. Kluwer Academic Publishers, 1991.

[ScKK93] K. Schneider, R. Kumar, and Thomas Kropf. Alternative proof procedures for finite-state machines in higher-order logic. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 213–226, Vancouver, B.C., Canada, August 1993. Springer.