Automatic Verification of Scheduling Results in High-Level Synthesis

Hans Eveking Holger Hinrichsen Gerd Ritter
Dept. of Electrical and Computer Engineering
Darmstadt University of Technology, D-64283 Darmstadt, Germany
{eveking/hinrichsen/ritter}@rs.tu-darmstadt.de

Abstract *A method for the fully automatic equivalence verification of a design before and after the scheduling step of high-level synthesis is presented. The technique is applicable to the results of advanced scheduling methods like AFAP and DLS, which work on cyclic control flows, as well as to pipelined designs.*

1 Introduction

Successful applications of automatic formal verification techniques are based on efficient decision procedures, e.g., OBDD's, fixed-point iteration techniques like model-checking, or provers for ground equational logic with uninterpreted functions. In the case of equivalence verification, the application of these decision procedures to realistic examples is even much more successful if the procedures are combined with domain-specific strategies which exploit *similarities* of the designs to be compared. Examples are the structural similarities of logic circuits after buffer insertion [11], and of finite-state machines after re-timing [17]. In the case of the verification of pipelined processors, the similarity of the pipelined version and the un-pipelined design is due to the fact that specific properties of functions like bit-vector addition are not exploited for pipeline-scheduling; thus, proof techniques for uninterpreted functions are applicable [4].

In the following, we argue that the equivalence verification of a design before and after the scheduling-step of high-level synthesis (HLS) is an instance of this type of comparison of "similar" designs, too. The decisions of HLS scheduling-techniques are based on abstract information concerning, e.g., data-dependencies or availability of functional units. The overall algorithm remains unchanged: HLS-scheduling is *not* able to transform a bubble-sort algorithm into a quick-sort algorithm. In many cases, the sets of transfer-operations and basic predicates are identical before and after scheduling.

While simple scheduling techniques like ASAP, ALAP, list-scheduling and forced-directed scheduling work on *acyclic* flow graphs and are, therefore, easily cross-checked by means of provers for ground-equational logic with uninterpreted functions, we present a method which is applicable to acyclic as well as to *cyclic* control-flows in the following. This is important since advanced scheduling techniques like path-based scheduling (AFAP [5]), dynamic loop scheduling (DLS [16]) or pipeline path-based scheduling (PPS [16]) modify cyclic control structures. In our method, the process of equivalence verification consists of a number of transformation steps which assimilate the original design to the scheduled design (Fig. 1). The given scheduled description defines the goal of transformation; it is, thus, not necessary to implement knowledge about the various scheduling algorithms in the proof procedure. The method is restricted to the proof of the computational equivalence: it is not checked if the scheduling goals (minimal number of steps or resources) are reached.

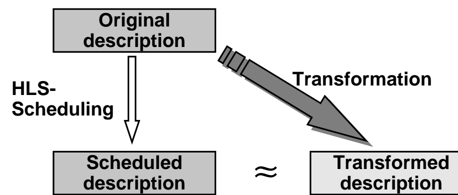


Fig. 1: Basic procedure of equivalence verification

Related work: According to [12], synthesis-verification maybe divided into *pre-synthesis* verification of synthesis-algorithms (e.g., [14]), *formal synthesis* where the constructive steps are embedded into a theorem prover (e.g., [3]), and *post-synthesis* verification where the synthesized results are verified afterwards (e.g., [13]). While most methods of post-synthesis verification use theorem proving techniques, a graph-based method was proposed in [8]. A technique for the verification of the register-allocation step by means

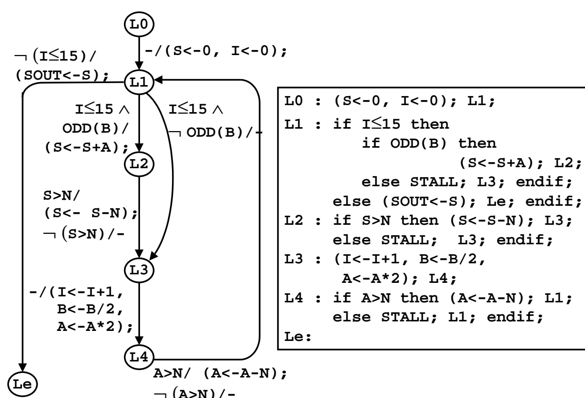


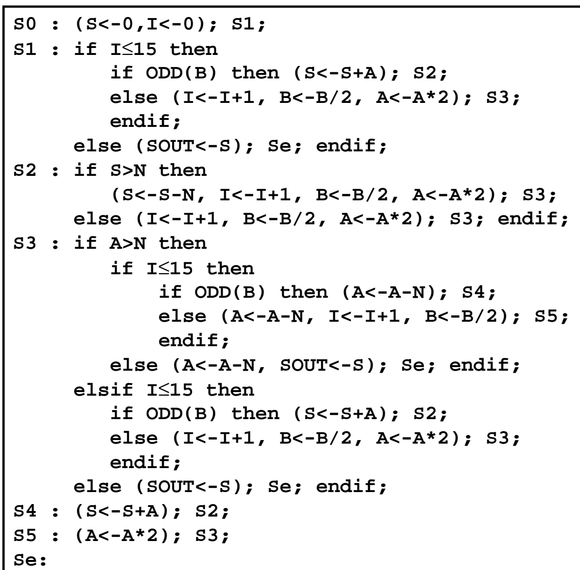
Fig. 2: Extended fsm and corresponding LLS-description

of model-checking is given in [1]. An automatic decision procedure for the verification of the scheduling step will be presented in the following.

In Sect. 2, our method of design representation is briefly surveyed. Sect. 3 presents the repertoire of basic transformations. The assimilation strategy is introduced in Sect. 4. Experimental results are given in Sect. 5.

2 Representation of designs

HLS-scheduling typically transforms an original VHDL or Verilog description into a data-flow or control-flow graph representation [16] in a first step. The scheduling result is often given as an extended fsm. We represent the original description as well as the scheduled result textually in our internal language LLS (Language of Labelled Segments). The rationale behind the introduction of another type of representation is that a number of rewrite-rules for the *textual manipulation and transformation* (Sect. 3) of LLS-descriptions exist. Fig. 2 shows an example adapted from [16] (the description calculates $a \cdot b \ mod \ n$) which illustrates the analogy between the extended fsm notation and our textual representation. *Labels* like `L0` correspond to control states, and are used to guide the flow of control. An *initial label* (`L0` in Fig. 2) has to be identified for each description. A LLS-description consists of a number of *segments* of the form `L:B` where `B` is called the *segment-body* associated with label `L`. The labels occurring in the segment-body are called *exit labels*, and are used to specify the flow of control, e.g., `L2`, `L3` and `Le` are the exit labels of segment `L1` (Fig. 2). The data-operations are specified in the segment body. Assignments to a variable like `x<-y` are



```
S0 : (S<-0,I<-0); S1;
S1 : if I≤15 then
        if ODD(B) then (S<-S+A); S2;
        else (I<-I+1, B<-B/2, A<-A*2); S3;
        endif;
     else (SOUT<-S); Se; endif;
S2 : if S>N then
        (S<-S-N, I<-I+1, B<-B/2, A<-A*2); S3;
     else (I<-I+1, B<-B/2, A<-A*2); S3; endif;
S3 : if A>N then
        if I≤15 then
            if ODD(B) then (A<-A-N); S4;
            else (A<-A-N, I<-I+1, B<-B/2); S5;
            endif;
        else (A<-A-N, SOUT<-S); Se; endif;
     elsif I≤15 then
        if ODD(B) then (S<-S+A); S2;
        else (I<-I+1, B<-B/2, A<-A*2); S3;
        endif;
     else (SOUT<-S); Se; endif;
S4 : (S<-S+A); S2;
S5 : (A<-A*2); S3;
Se:
```

Fig. 3: Example of Fig. 2 after DLS scheduling

called *transfers*. Parenthesis enclose synchronous parallel transfers. Hence, `(x<-y, y<-x)` exchanges the contents of `x` and `y` in a single step. Fig. 3 (taken from [16], too) shows the example of Fig. 2 after DLS-scheduling. The problem discussed in this paper is the automatic proof that, for example, the descriptions of Fig. 2 and Fig. 3 are computationally equivalent: if both are started with the same initial values then the same final values result when the end-labels `Le` and `Se`, respectively, are reached.

A more detailed introduction to LLS is given in [7]. For the purpose of this paper, it is sufficient to consider one more aspect of the language: while each control state and thus each label consumes exactly one step of time in Figs. 2 and 3, the sequential composition operator `;` can be used in LLS segments for consecutive transfers. Thus, if control reaches `M0` in the following example, then `x` is incremented in the first step of time; in the second step, `y` is incremented and control is transferred to `M1`:

```
M0: (x<-x+1);
    (y<-y+1); M1
```

Since loop-constructs are not provided in LLS a segment-body remains *acyclic*. A segment in which the serial composition of transfers does not occur is called a *single-step segment*. Descriptions in *state diagram form*, e.g., the descriptions of Figs. 2 and 3 consist only of single-step segments.

Two LLS segments are *computationally equivalent* $\simeq$ if both produce the same final values at the exit labels on the

same initial values.

Two LLS descriptions are *trace-equivalent* $\cong$ if all runs coincide step-by-step.

Two LLS descriptions are *computationally equivalent* $\simeq$ relative to pairs of initial/final labels if both produce the same final values at the final labels on the same initial values.

The two descriptions of Figs. 2 and 3, for instance, are computationally equivalent for the pair (L0,S0) of initial labels and for the pair (Le,Se) of final labels, but not trace-equivalent.

3 Basic transformations

Two distinct classes of basic transformations of LLS-descriptions are used in our technique:

1.: The well-known rules given by Bernstein [2] govern possible parallelizations of consecutive assignments. Let D denote the set of all variables on the destination side of an assignment and let S be the set of all source variables found in right-hand side expressions, e.g., for (x<-a+b, y<-x*a) we have $D = \{x, y\}$, $S = \{a, b, x\}$. The rules given by Bernstein permit the parallelization of D2<-S2 with D1<-S1:

```
(D1<-S1);              =>  (D1<-S1, D2<-S2);
(D2<-S2, D3<-S3);          (D3<-S3);
```

if the following conditions hold:

$$D1 \cap D2 = \emptyset, D1 \cap S2 = \emptyset, D2 \cap S3 = \emptyset$$

These rules can be extended to cope with *forwarding* and the *introduction of pipeline registers*: if the second condition is not satisfied then it is possible to *forward* the result computed for D1. If the third condition does not hold then *pipeline registers* have to be introduced in order to save previously computed values for S3. The following example illustrates these techniques:

```
(x<-a+b);          =>  (x<-a+b, y<-a+b, yp<-y);
(y<-x, z<-y);          (z<-yp);
```

2.: While the application of the first class of transformations preserves the computational equivalence $\simeq$ of a segment body, the second class is based on a number of rules which govern trace-equivalent $\cong$ transformations of LLS-descriptions. The equivalence-rules shown in Fig. 5 are independent of specific predicates and data-operations, and maybe viewed as a theory of "uninterpreted" control structures (the rules can be traced back to early work on theories of equivalent microprograms [9]). Rule R1 (Fig. 5) allows for the substitution of segment-bodies for labels. For example, we may replace the occurrence of label L1 in segment L of Fig. 4(a) with the segment-body of L1 yielding the segment (b) (only the modified segment L is shown).

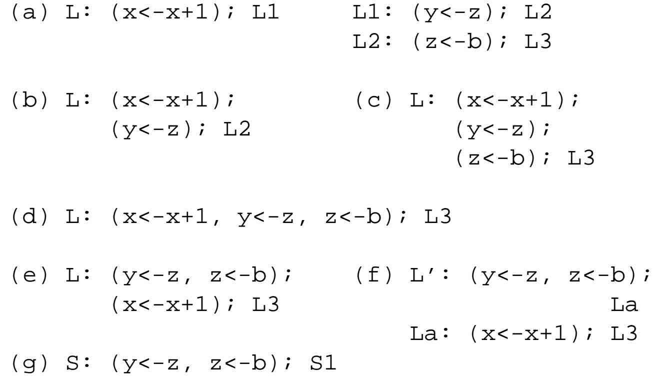


Fig. 4: Substitution of segment-bodies for labels

Substituting the segment-body of L2 in L results in (c). The segment-body of L consists now of a linear sequence to which we may apply parallelization techniques governed by Bernstein's rules yielding, e.g., (d).

Vice versa, transfers maybe serialized (e) and labels like La maybe introduced to cut off part of a segment-body (f).

Rules A1 to A5 govern straightforward relationships between nested if-clauses and boolean connectives. A1, for instance, is a straightforward rule for the negation of an if-condition.

Rules A6 and A7 refer to redundant conditions and operations, respectively.

Rules A8 and A9 allow for moving transfers into and out of if-clauses. If a transfer S in front of an if-clause is pushed into the if-clause (Rule A9) then the if-condition p has to be transformed into the condition $\{$S$\}$p. This type of predicate is called a *virtual predicate* since it is equivalent to the predicate p after a virtual execution of S. For instance if S is x<-x+1 and p is x=0 then $\{$S$\}$p becomes x+1=0. Of course, if the support of p and the set of destinations of S are disjunct then $\{$S$\}$p is equivalent to p. Virtual predicates have to be replaced by equivalent regular ones if the design has to be implemented.

Theorem: The application of the two classes of transformations above preserves the computational equivalence of LLS-descriptions.

The two classes form the basic repertoire of transformations employed by the assimilation algorithm given in the next section. In order to obtain at least *experimental* evidence of the completeness and power of the transformations, the extremely complex problem of the automated and formally correct synthesis of pipelined processors has been solved in [10]; a CPI of 1.1 has been obtained in the case of the DLX architecture.

```
A1: D(if p then S1 else S2 endif;) ≅
    D(if not p then S2 else S1 endif; )
A2: D(if p and q then S1
      else S2 endif;) ≅
    D(if p then if q then S1
                else S2 endif;
      else S2 endif;)
A3: D(if p or q then S1 else S2 endif;) ≅
    D(if p then S1
      elsif q then S1
      else S2 endif;)
A4: D(if p then if q then S1
               else S2 endif;
      else S3 endif;) ≅
    D(if p and q then S1
       elsif p and not q
         then S2 else S3 endif;)
A5: D(if p then S1 elsif q then S2
               else S3 endif;) ≅
    D(if p or q then
         if p then S1 else S2 endif;
      else S3 endif;)

A6: D(if p then S else S endif;)
    ≅
    D(S )
A7: D(if p then S1 else S2 endif;)
    ≅
    D(if p then if p then S1
                  else R endif;
      else S2 endif;)
A8: D(if p then S1 else S2 endif;
      S;)
    ≅
    D(if p then S1;S;
      else S2;S; endif;)
A9: D(S;
      if p then S1 else S2 endif;)
    ≅
    D(if {S}p then S;S1;
      else S;S2; endif; )

R1:        L: S
      ---------------
      D(L ) ≅  D(S )
```

Fig. 5: Rules of trace-equivalent transformations of LLS-descriptions

4 The assimilation algorithm

Rather than to give a proof of the computational equivalence of the original description and the scheduled result which may involve induction-steps, etc., our proof-method transforms the original description into a computationally equivalent description which is *bisimilar* ≈ to the scheduled description (Fig. 1). Informally, two descriptions `D` and `D'` in state-diagram form are bisimilar, `D` ≈ `D'`, iff for each segment `L` of `D` there exists a bisimilar segment `S` in `D'` and vice versa. Two single-step segments `L` and `S` are bisimilar, (`L`,`S`) ∈≈, iff (i) the same data-operations are performed, and (ii) control is transferred to bisimilar segments. If two descriptions are bisimilar then they are trace-equivalent ≅ (but there are trace-equivalent descriptions which are not bisimilar).

The proof is given in terms of a step-by-step breadth-first unfolding of both descriptions starting with the initial segments. Fig. 6 shows the incomplete tree of unfolding the two descriptions of Figs. 2 and 3. The unfolding starts with time-step 1, proceeds to time-step 2, etc. Proving the initial segments to be bisimilar in time-step 1 (e.g., `L0` and `S0` of Figs. 2 and 3, respectively), entails bisimulation proofs of consecutive segments ((`L1`,`S1`) ∈≈ in the example) in subsequent time-steps, etc.

We consider the case where the original description and the scheduled result have the same sets of transfers and basic predicates.

At each time-step, an attempt is made to adapt the segment(s) of the original description to the segment(s) of the scheduled result. Let `L` be a segment of the original description, and let `S` be a segment of the scheduled description. The proof of (`L`,`S`) ∈≈ proceeds as follows:

1.: `L` is manipulated in such a way that the data-operations of the segment `S` result. In detail, this requires the following steps:

i. In the *expansion-phase*, the set M of transfers lacking in `L` is identified. In the example of Fig. 4 (a) and (g), we have $M = \{$`(y<-z)`, `(z<-b)`$\}$. `L` is extended in order to move the missing transfers into segment `L` by means of the substitution of segment-bodies for labels. In Fig. 4 (b), the missing transfer `(y<-z)` is part of `L` after substitution of the segment-body of `L1`.
   Searching for transfers of M, the process of substitution stops if the same label was to be substituted a second time because this would not make additional transfers available. The procedure fails if transfers of M can not be found.
   Since the resulting segment-body is acyclic we can apply the parallelization techniques of Sect. 3 afterwards. The lacking transfers are moved into the first step of `L` observing Bernstein's rules (Fig. 4 (d)). It is important to parallelize the transfers with the smallest time-distance first: if in the example above, the transfer `(z<-b)` were parallelized before `(y<-z)` then the introduction of a pipeline-register for `z` would have been necessary.
   The expansion-procedure is more complex if if-clauses are involved requiring the application of rules A8 and A9, in particular.

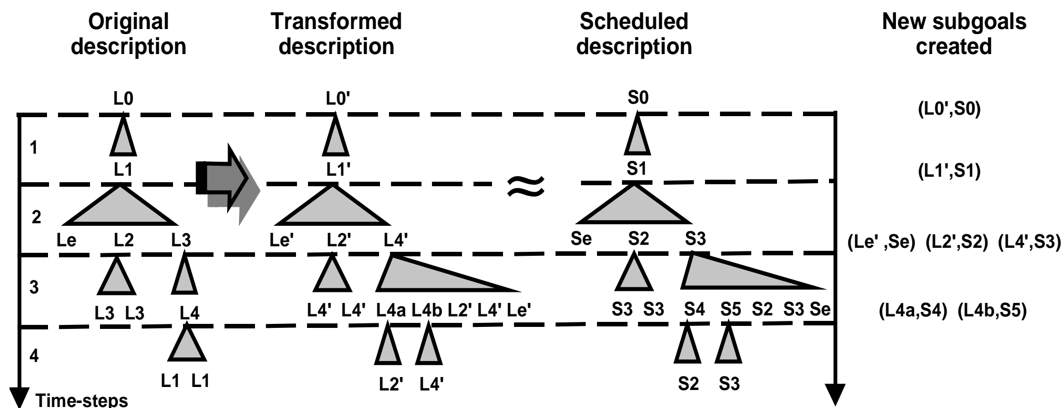


Fig. 6: Bisimulation proof by unfolding and assimilation

ii. In the *reduction-phase*, the surplus transfers are serialized, and new segments are introduced if necessary (Fig. 4 (e) and (f)). Comparing Fig. 4 (f) and (g), two bisimilar segments `L'` and `S` result provided that `La` and `S1` are bisimilar, too, i.e., (`La`,`S1`) $\in\approx$.

2.: If the first step was successful, then it is now proven that `L` can be replaced by a segment `L'` which is bisimilar to segment `S` provided the newly created subgoals can be satisfied. If the same problem (`L`,`S`) $\in\approx$ occurs again in the process of unfolding in a subsequent time-step then this does not create a new subgoal since the same procedure as before can be applied. In Fig. 6, for instance, the assimilation of `L1` and `S1` in time-step 2 created the new segment `L1'` with the new subgoals (`Le'`,`Se`) $\in\approx$, (`L2'`,`S2`) $\in\approx$, (`L4'`,`S3`) $\in\approx$. The solution of these problems reproduces (`L4'`,`S3`) $\in\approx$ three times in time-step 3, and once in time-step 4 which do not need to be considered further.
3.: A successful process of unfolding, therefore, results in a finite subtree where the leafs are subgoals which occurred in previous time-steps. The assimilation process stops if a fixed-point is reached and no new subgoals are created. The process will always terminate since the number of subgoals, i.e., the number of pairs of single-step segments with a finite repertoire of transfers and basic predicates is finite. The complete process of assimilating the description of Fig. 2 to the scheduled result of Fig. 3 creates 7 subgoals, and is shown in Fig. 6.

5 Experimental results

Our verification technique has been applied to several examples reported in the literature. The following table summarizes the results on a SUN Ultra2 300 MHz. PREFETCH is an example of AFAP scheduling taken from [5]. MODULO is the example of Fig. 2 originally given by [16] with results of cycle-modifying AFAP, DLS (Fig. 3) and PPS scheduling techniques.

Design	Scheduling method	Cpu-time (sec.)
PREFETCH	AFAP	0.06
MODULO	AFAP	0.10
	DLS	0.19
	PPS	0.10
FIR FILTER	feasible	1.35
	optimal	1.18
C1	3-stage pipe	0.08

Since forwarding and the introduction of pipeline registers is provided by the repertoire of basic transformations, the verification system is also able to verify *pipelined* scheduling results. The problem is more complex since the sets of transfers and predicates is distinct due to pipeline-registers and forwarding techniques. Two pipelined schedules of an FIR FILTER by the Sehwa system [15] were verified. C1 [6] is a three-stage pipeline originally verified by means of PVS, and includes forwarding as well as pipelined read/write-operations of a register-file.

The examples demonstrate that the fully automatic verification of HLS scheduling results is feasible with our technique within a few seconds.

References

[1] P. Ashar, S. Bhattacharya, A. Raghunbathan, and A. Mukaiyama. Verification of RTL generated from scheduled behavor in a high-level synthesis flow. In *Proc. ICCAD'98*, 1998.

[2] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. Computers*, pages 757–763, 1966.

[3] C. Blumenröhr, D. Eisenbiegler, and R. Kumar. Applicability of formal synthesis illustrated via scheduling. In *Proc. IWLAS'96*, 1996.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. Computer Aided Verification '94*. Springer LNCS 818, 1994.

[5] R. Camposano. Path-based scheduling for synthesis. *IEEE Trans. on CAD*, 10(1):85–93, 1991.

[6] D. Cyrluk. Inverting the abstraction mapping: a methodology for hardware verification. In *Proc. FMCAD '96*. Springer LNCS 1166, 1996.

[7] H. Eveking, H. Hinrichsen, and G. Ritter. Formally correct construction of pipelined processors. Technical Report 98-6-1, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 1998.

[8] F. Feldbusch and R. Kumar. Verification of synthesized circuits at register transfer level with flow graphs. In *Proc. EDAC'91*, 1991.

[9] V.M. Glushkov. Automata theory and formal microprogram transformations. *Kibernetika*, 1(5):1–9, 1965.

[10] H. Hinrichsen, H. Eveking, and G. Ritter. Formal synthesis for pipeline design. In *Proc. DMTCS+CATS'98*. Springer DMTCS, 1998.

[11] A. Kuehlmamm and F. Krohm. Equivalence checking using cuts and heaps. In *Proc. DAC'97*, pages 263–268, 1997.

[12] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design - a classification and survey. In *Proc. FMCAD '96*. Springer LNCS 1166, 1996.

[13] M. Mutz. Automatic post synthesis verification support for a high level synthesis step by using the HOL theorem proving system. In *Proc. CHARME '97*. Chapman & Hall, 1997.

[14] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. In *Proc. ICCD'98*, 1998.

[15] N. Park and A.C. Parker. Sehwa: a software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on CAD*, 7(3):356–370, 1988.

[16] M. Rahmouni and A.A. Jerraya. Formulation end evaluation of scheduling techniques for control flow graphs. In *Proc. EuroDAC'95, Brighton*, 1995.

[17] C. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. DATE'98*, 1998.