International Journal of Computer Aided VLSI Design 3, 9–41 (1991)

Transformational Design Methodology for Parameterized VLSI Modules

D. Verkest, L. Claesen, and H. De Man
Interuniversity Micro Electronics Center (IMEC)–VSDM
Belgium

A new method is presented to be used for either *guided synthesis* or formal correctness verification of *parameterized* digital hardware modules. The method is based on the concept of correctness preserving transformations, formalized by means of *transformation descriptions*. A parameterized description of the module is used as the specification. This specification can then be manipulated by the designer by using transformation descriptions. After a number of transformation steps, the designer has a parameterized description of the implementation of the module. In this method, manipulations are done *directly* on an existing hardware description language, instead of using derived formalisms, as is done in other approaches. Starting from the original structure description and a transformation description, the structure description corresponding with the transformed design can be generated *automatically* using the techniques described in this paper. The algorithms to support this design methodology are presented and examples are elaborated to illustrate the important concepts. This design method has been applied to actual VLSI designs, such as a pipelined and parameterized multiplier-accumulator module and a systolic implementation of a finite impulse response filter.

design of VLSI modules, transformational design, guided synthesis, formal verification, correctness preserving transformations, hardware description languages

1 GUARANTEED CORRECT VLSI DESIGNS

Current capabilities of VLSI technology allow that larger systems are being integrated on one chip. Therefore, the design of these systems has become very complex. Formerly used design methodologies, such as top-down or bottom-up, no longer are affordable, because of the large *design iteration cycles* and the fact that only a limited number of people in the world are still able to manage the complete design trajectory from high-level specifications down to MOS transistor techniques. The currently emerging methodology of custom VLSI design is more like a *meet-in-the-middle* [1] approach, as shown in Figure 1, where two different design teams meet each other at the level of functional modules, such as arithmetic and logic units (ALUs), multipliers, and so forth. To be reused frequently by system designers, these modules must be flexible. This is achieved by the ability to generate modules in a parameterized way [2].

During this design activity, there is the problem of guaranteeing or verifying the correctness of the design. The CAD support for this aspect is currently very

Correspondence and requests for reprints should be sent to D. Verkest, IMEC–VSDM, Kapeldreef 75, B-3001 Leuven, Belgium.

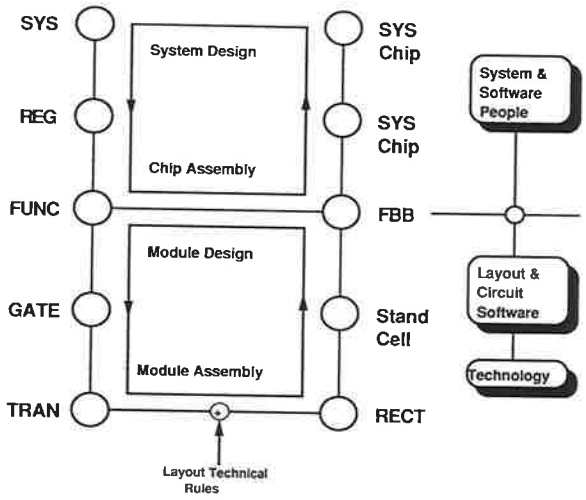


Figure 1. Meet-in-the-middle strategy.

poor. Only for specific classes of applications, automatic synthesis (silicon compilation) from high-level behavioral specification to chip layout can be done, resulting in *correctness by construction*. For the class of digital filters and general digital signal processing (DSP) systems, the feasibility of such an approach has been demonstrated [3, 4].

The Cathedral II silicon compiler [4] is organized according to this meet-in-the-middle strategy. An automatic synthesis is done from the high-level specification language Silage [5] into a number of controllers and execution units that are predesigned as parameterized modules. Examples of parameters are the word-length of the inputs, the number of input buses, and Booleans to indicate the presence or absence of an accumulator section or a pipeline. The set of required parameterized module generators is designed by circuit designers as a set of Lisp procedures [2] automatically generating the circuit layout for the module. While the high-level synthesis is correct by construction, the correctness of the module generators is still being verified by classical verification techniques such as logic and circuit simulation.

For several other design classes, automatic synthesis is not yet feasible, and "manual" system design is still used for most applications. This is the case for full custom design as well as for highly optimized circuits, e.g., video or radar applications. The only tools available for correctness verification are simulators. However, simulation suffers from a number of well-known drawbacks: covering all input combinations is not feasible for realistic VLSI designs, and detection of design errors depends on the choice of appropriate input signals and the correct interpretation of output results. Another complication in the case of modules is that correctness must be guaranteed not only for one instance, but for the whole allowable parameter domain of the generator procedures. Because of the multiplicity of possible instances that can be generated, verification methods that act on instances no longer are appropriate. Methods to *formally prove* the correctness would be very useful here.

This paper describes a new technique that can be used to guarantee the correctness of such parameterized hardware modules. In Section 2, a short overview will be given of different approaches taken to tackle the verification problem and how they relate to the problem of verifying parameterized modules. The transformational design methodology will be explained in Section 3. To make this methodology accessible to VLSI designers, we have decided in favor of the use of an existing hardware description language (HDL) for performing transformations. In Section 4, this HDL and the language to describe the transformations are presented. In Section 5, it is shown how such an existing HDL can be used for transformational design. The algorithms underlying this methodology will be discussed in Section 6. Next, two actual VLSI design examples will be discussed in the light of this methodology. In Section 7, the design of the multiplier-accumulator module from the module library of the Cathedral II silicon compiler [4] is discussed. Finally, in Section 8, the design of a finite impulse response (FIR) filter, starting from a straightforward implementation and resulting in a systolic implementation, is presented.

2 APPROACHES FOR FORMAL CORRECTNESS VERIFICATION

Several approaches have been developed to solve the problem of formal verification. A good survey of the field can be found in [6]. Most of these formal verification methods could also be used to verify the correctness of parameterized modules. However, in general, verification after the design facts, i.e., comparing a highly optimized implementation with a high-level specification, is not feasible without a lot of assistance from the designer. From the experience of researchers using general theorem provers to perform such proofs (e.g., [7, 8]), one can conclude that the construction of the proof takes a lot of effort. In addition, the proof has to be well understood in advance.

As opposed to verification after the design facts, one can adopt a *transformational design methodology*: correctness preserving transformations are used to refine designs toward efficient implementations. This approach has the advantage that the "verification" can proceed in small steps—in fact, it is not really verification, but a constructive proof or guided synthesis. Following this course, it is the designer who, while designing his system, constructs the proof, i.e., the sequence of transformation steps. A verification effort after the design facts is no longer needed.

A CAD tool supporting this methodology first should check whether the transformation is applicable in a specific situation by checking a number of constraints; second, execute the transformation to produce the description of the transformed design. Special formalisms to support such a design methodology have been developed by Sheeran [9, 10] and Subrahmanyam [11]. Milne [12] and Eveking [13]

stressed the role of constraints in this context. In principle, also a general theorem prover such as HOL [14] could be used as a transformational design system.

The key role of the designer in a transformational design methodology was already mentioned; it is the designer who must decide what transformations to apply. This implies that the formalism underlying the transformational design must be *accessible to the designer* by using his familiar design environment and hardware description methods. The formalisms mentioned earlier are lacking on exactly that point.

To make tranformational design accessible to VLSI designers, we have opted for the use of an *existing HDL*. Instead of developing a separate formal system, we develop a system to support transformational design using this existing HDL.

3 TRANSFORMATIONAL DESIGN METHODOLOGY

The transformational design methodology is illustrated in Figure 2. It consists of a step-by-step transformation of the specification into an efficient implementation, based on the concept of correctness preserving transformations. In the figure, two alternative paths are shown: formal verification is the path going up from implementation to specification and synthesis is the path going down. Before going into detail, we will define some terms (see Figure 2).

- *transformation* (the large arrow in the figure): the process of altering the specification to obtain the implementation.
- *transformation step*: one step out of the transformation. A transformation step consists of an intermediate design step and a transformation description.
- *intermediate design description* (the rectangular boxes): the structure description that describes the result of a transformation step and also serves as input for the next transformation step. The structure descriptions can be parameterized.
- *transformation description* (the rounded boxes): describes which primitive equivalence transformation is used in the transformation step and to which part of the design it is applied. The transformation descriptions can be parameterized.
- *primitive equivalence transformation*: the correctness preserving transformation used in the transformation description. Examples of these can be found in Section 4.2.

During the synthesis path, the transformation steps are just formal descriptions of the optimizations the designer has in mind, while in the verification process the transformation descriptions would be exactly the inverse of these optimizations. It is our experience that it is easier to describe these transformation steps during the synthesis phase; therefore, we will concentrate on the guided synthesis. Note, however, that this is not a restriction of the applicability of the techniques described in this paper; they are just as well applicable to both alternatives. In the remainder of this paper, transformational design will be regarded as a synonym for the guided synthesis alternative of Figure 2.

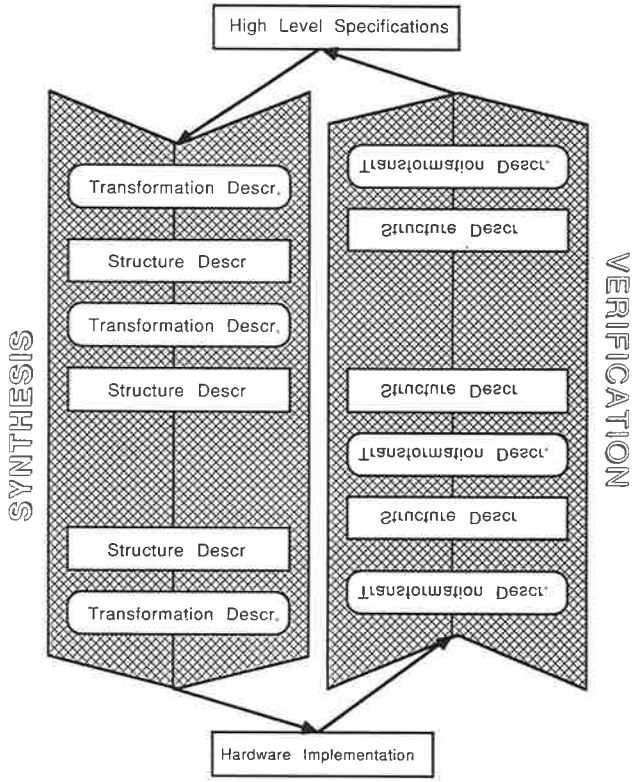


Figure 2. Transformational design methodology.

A transformational design system takes as input a high-level specification of the design. Starting from that initial specification, the designer manipulates the hardware description using the transformation descriptions until he achieves the desired implementation in terms of primitives that can be realized as circuits in hardware. These primitives are usually small blocks of up to 20 transistors, such as gates, or small functional blocks, such as full adders. The correctness of these basic primitives with respect to their layout realization can be verified using circuit extraction from the (symbolic) layout. The logic equations can then be extracted [15, 16] from the obtained transistor netlists and compared [17, 18] to the logic equations that originate from the behavioral specification of the primitive cell. Given the parameterized structure description of the implementation, the actual module generator procedure can be created by the module designer [2].

The *high-level specification* consists of an interconnection of predefined cells

together with the don't care behavior and corresponds to a naive implementation of the required behavior. The HDL used for this description (and which is also used for the description of the intermediate design steps) is defined in Section 4.1. An example is the multiplication algorithm, which can be implemented naively by the shift-add algorithm as learned in primary school. This naive description, although easy to understand, is never used as implementation because of considerations on speed, area, throughput, and so forth. Instead, a better suited implementation is used, e.g., an implementation of the Booth [19] algorithm. To obtain this more efficient implementation starting from the implementation of the naive shift-add algorithm, the designer must supply a number of transformation descriptions.

The *transformation descriptions* are, as stated earlier, used to describe the transformation steps. The language used for these descriptions is defined in Section 4.2. The designer has a number of primitive equivalence transformations available that could be used in the transition descriptions. These are transformations that, when applied to an arbitrary design, yield a functional equivalent design if certain conditions are met (see Section 4.2).

Thus the method is a step-by-step correctness preserving transformation under the full guidance of the designer, who makes all of the design decisions. This is in contrast to automatic synthesis and verification techniques, where the equivalence between implementation and specification is conserved, respectively proved, by the system without any information from the designer. Automatic systems suffer from the drawback that they must determine all of the design transformations on their own. This is especially difficult for the general case of complex parameterized hardware designs. By letting the designer formally express his elementary design transformations, the synthesis or verification problem becomes more manageable.

By recording the transformation steps, the designer can maintain a history of the evolution of the design. Owing to this good documentation of the design, later modifications can be easily achieved and proved correct, starting from an intermediate design step.

4 LANGUAGES

Before going into detail on the technique developed to calculate a new hardware description given a transformation description and the original hardware description, we present the languages used herein. First, the HDL, called Hilarics, is presented, and then the transformation description language is discussed. More details on the transformational design strategy and on these languages and their semantics can be found in [20, and 21].

4.1 HDL Hilarics

We have chosen to use directly the existing Hilarics [22] parameterized structure description language to perform the manipulations. This is in contrast to the methods mentioned in Section 2, that try to model hardware based on functional models,

first-order or higher-order logic. This choice is motivated by the fact that hardware designers are traditionally much more familiar with such a language than with the functional or logical formalisms mentioned earlier.

Hilarics starts from the concept that the structure (composition) of a circuit should be described completely independent from other design views. Hilarics is a purely applicative language, in which designs can be described hierarchically and in a parameterized way. As such, the language is currently in use for the description of the structure part for register-transfer descriptions, for circuit-level simulation, for timing verification, for switch- and logic-level simulation, and as a definition input for the module generation environment. Hilarics has been used for the description of several hierarchical and parameterized VLSI designs and parts thereof.

In this paper, the main emphasis is on net-oriented descriptions in Hilarics. The interconnection part of Hilarics consists of a number of connections or nets. The nets are the sets of interconnected terminals of the components and of the cell. A net is defined by the two terminals that are connected, separated by an equal sign. All terminals defined in the definition and declaration part must be connected, and no other terminals can be used. Input terminals can be connected only once; output terminals can be connected to more than one other (input) terminal.

To exploit repetitivity, iterativity, and to be able to deal with conditionality, certain controlling constructs can be used: *for-loops and if-then-else* constructs.

For the specific domain of digital signal processing and for arithmetic applications, the following additional view information is required.

- The *delay potential* indicates for synchronous systems the relative shift in time between nets (more precisely between the signals on the nets). This concept allows one to perform delay management and retiming transformations [23]. It is obvious that no time shift can occur between the two terminals of the same net, so the delay potential is a property of the net and not of the terminals.
- The *arithmetic weight factor* indicates the place of a terminal in a bit vector. Two consecutive bits in a binary number have a different weight factor (by a factor 2). Weight factors are a property of the terminals and not of the nets. For example, in a hardwired shifter, realizing a multiplication by 2, the two terminals of a net will have different weight factors by a factor 2.

The Hilarics language is used to describe all the intermediate design descriptions of the transformation. An example of a structure description with Hilarics is given subsequently. It describes the interconnection pattern of the carry inputs and outputs of the structure at the top in Figure 6, which is the adder matrix and the accumulator section of the Booth multiplier module explained in more detail in Section 7. The figure has been made for an 8×8 instance of the module, but all of the descriptions are parameterized. The C terminals in the structure description are the carry inputs (the east terminals of the full adder cells). The K terminals are the carry outputs (the west terminals of the full adders). The characters inside brackets (i, j *in* $C(i, j)$) are indices, whereas the character after the bracket (j in $C(i, j)j$) denotes the weight factor.

```
for i = 1 to m + 1 do
 for j = 0 to n + 2 * m - 1 do
  if i ≤ m - 1 then if j ≥ 2 * (i - 1) then if j ≤ 2 * i - 1 then C(i, j)j = 0
                                                             else C(i, j)j = K(i, j - 1)j
                                                  fi
                                        else nonet
                    fi
               else if i = m then if j ≥ 2 * (i - 1) then C(i, j)j = 0
                                                     else nonet
                                  fi
                             else if j > 0 then C(i, j) = K(i, j - 1)j
                                           else C(i, j)j = 0
                                  fi
                    fi
  fi
 rof
rof
```

4.2 Transformation Descriptions

This section contains several topics. First, the language used for the transformation descriptions is defined. In the second paragraph, the different classes of primitive equivalence transformations are explained.

4.2.1 Transformation Description Language

A transformation description indicates which primitive equivalence transformation is used, and to which part of the design this primitive transformation is applied. The transformation language is very similar to Hilarics, but instead of net definitions we find primitive equivalence transformations.

The primitive equivalence transformations are correctness preserving transformations. If they are applied to a specific structure, a new structure is obtained that is functionally equivalent to the original one.

The transformation description language (TDL) is applicative in order to facilitate the formal manipulations. Only one primitive equivalence transformation may be used in a single transformation description, but, of course, it can be applied to several components of the structure. This restriction stems from the fact that the transformation description is applicative. If more than one primitive transformation were allowed, then the transformation description would become procedural, which would substantially complicate the proof techniques.

4.2.2 Classes of Primitive Equivalence Transformations

In the DSP-like applications considered so far, the following classes of primitive equivalence transformations can be distinguished: arithmetic transformations, Boolean transformations, and flow-graph transformations. For each primitive transformation, there are a number of constraints that must be met by the structure

description and its components before the primitive transformation may be validly applied. Because of this constraint mechanism, the exact function of the basic cells is of little importance. If the basic cells satisfy the constraints imposed by the primitive transformation, then the transformation description is valid.

1. *Primitive arithmetic equivalence transformations*. These can be applied for arithmetic building blocks, such as full adders. The *comm* transformation belongs to this class. In Figure 3, the function of *comm*(a, b) is to make a new net consisting of the terminal a and the terminal to which b is connected. Since terminal b is no longer connected after the transformation, it must be reconnected by another *comm* transformation, such as *comm*(b, a). This means that every terminal involved in a *comm* operation has to occur once as the left-hand-side argument and once as the right-hand-side argument of *comm*. This is a valid equivalence transformation if the components concerned allow commutativity. In addition, the terminals concerned must have the same delay potential and the same weight factor.
2. *Primitive Boolean equivalence transformations*. These properties act on Boolean operators. The classical rules of Boolean algebra can be used here, and the validity of these transformations can be checked by a tautology checker [17, 18]. A good example of a transformation belonging to this class is given in Figure 4. Here two full adder cells are replaced by one dedicated 2-bit adder cell to optimize the critical path in the adder matrix of the Booth module. The conditions for this operation are that the logic functionality and the interconnection with the surrounding cells must be the same for both substituents.

Notice that, for the equivalence proof of the logic function of these smaller

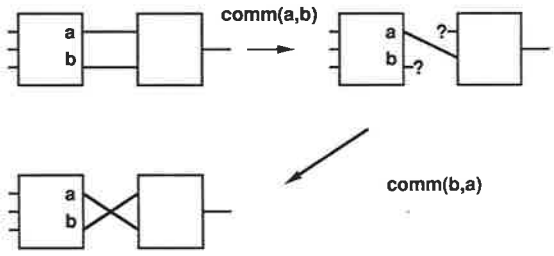


Figure 3. Primitive arithmetic transformation: switching terminals using *comm*.

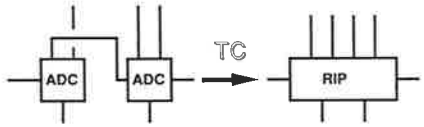


Figure 4. Two Boolean equivalent cells that can be checked by a tautology checker.

blocks, no parameterized descriptions are necessary and a tautology checker can indeed be used.

3. *Primitive flow-graph equivalence transformations*. Examples of these transformations are delay management and retiming operations [23], as shown in the upper part of Figure 5. These properties are used, for example, in the reorganization of the internal pipeline of the multiplier.

   A trivial operation eliminates superfluous components, as depicted in the lower part of Figure 5. The obvious conditions are that the two components must be instances of the same cell and that their inputs must be identical.

An example of a transformation description is given in Figure 6. In the figure, a transformation step out of the transformation of the Booth module is shown. It uses the primitive *comm* operation to exchange carry inputs and outputs in order to transform the adder matrix of the module from a carry-propagate structure into a carry-save structure. All the full adder cells with terminals that are affected by the transformation are "shadowed."

4.3 Formal Manipulations on Hilarics and on the TDL

As will be explained in Section 5, a lot of calculations must be done on the descriptions. As a result of these calculations, the descriptions tend to become much larger than necessary. To obtain a more concise description, some language manipulation rules can be applied. These rules can be compared with the generic conversion rules, as defined in [24]. They preserve meaning with respect to the behavioral (and the structural) model irrespective of the interpretation of the constants in the language. This means independent of the function of the components of which the terminals are defined in the net. Analogous operations are also available in applicative languages such as FP [25] and in HDLs derived from this approach [9].

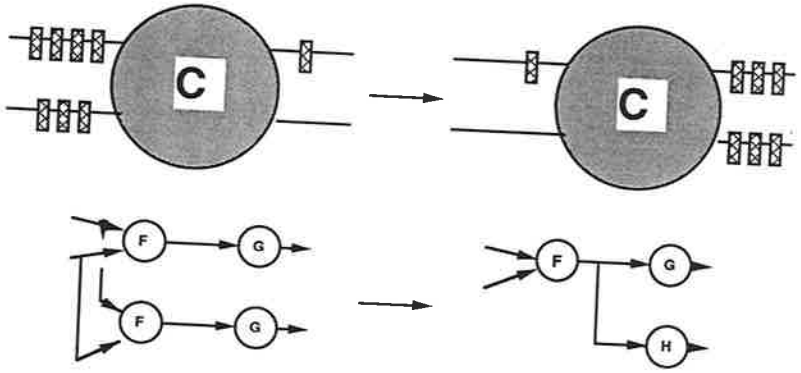
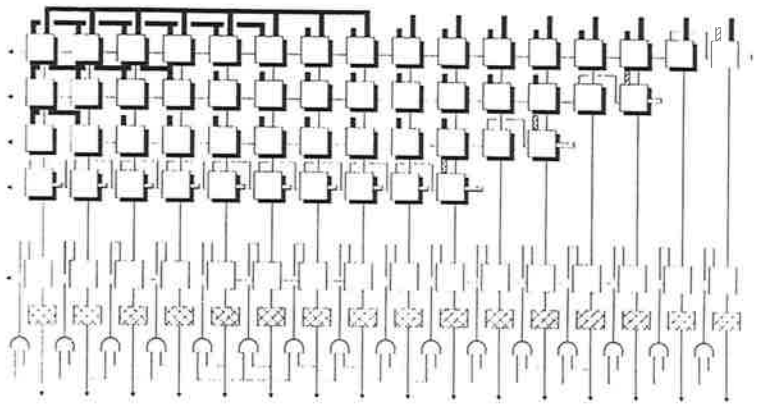


Figure 5. Retiming equivalence transformation and "remove" equivalence transformation.



```
FOR i=1 TO m DO
  FOR j=2*(i-1) TO n+2*m-1 DO
    IF i<>1 THEN COMM(C(i,j),C(i-1,j))
            ELSE IF j>=2*(m-1) THEN COMM(C(i,j),C(m,j))
                              ELSE COMM(C(i,j),C(j div 2 +1,j))
                 FI
    FI
  ROF
ROF
```

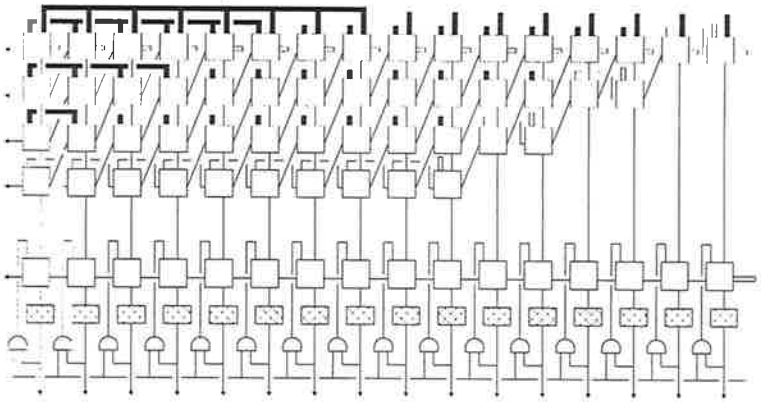


Figure 6. Transformation step from carry-propagate to carry-save structure.

The primitive equivalence transformations defined in the preceding sections can be viewed as language specific conversion rules, as defined in [21]. They preserve meaning with respect to the behavioral model depending on the semantics of the constants in the language.

5 TRANSFORMATIONAL DESIGN REVISITED

The concrete problem we must solve when we want to apply the transformational design methodology of Section 3 to the existing Hilarics HDL is:

- given a structure description in Hilarics
- given a transformation description
- *calculate* a new structure description in Hilarics, which represents the transformed structure.

In this section, we explain how this can be done and identify the subproblems to be solved.

The indices of components and nets in the parameterized structure and transformation descriptions indicate particular structural items. The structure description indicates the static composition of a cell. The transformation description indicates how a cell is recomposed in a functional equivalent cell. This implies that the structure description of the new, functional equivalent cell can be calculated by merging the structure description of the original cell and the transformation description. In the method we propose, this merging is done by applying the transformation description to each individual net description in the original structure description, as indicated schematically in Figure 7.

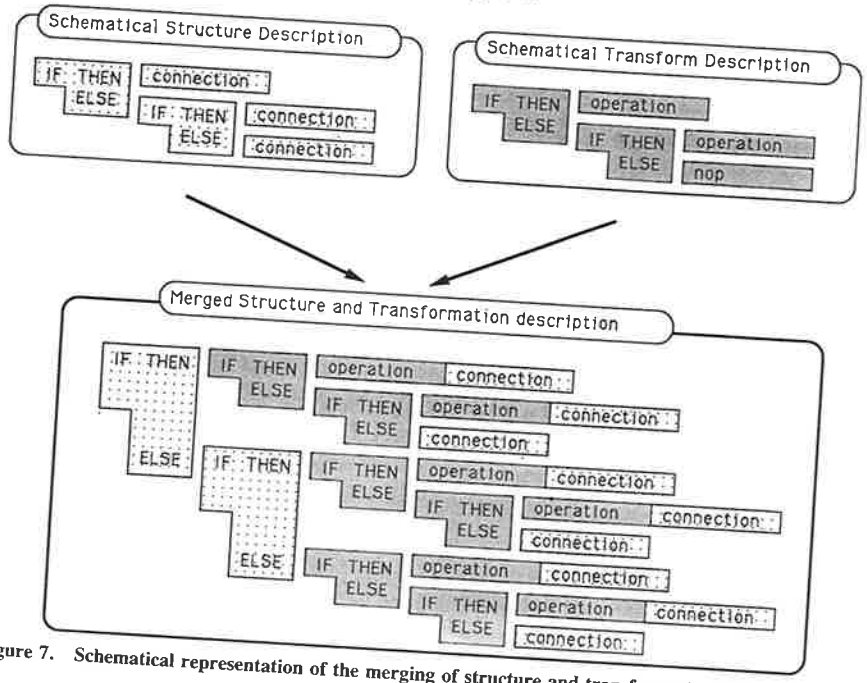


Figure 7. Schematical representation of the merging of structure and transformation description.

Before starting with a concrete example, we will discuss a few procedures that will be used frequently in the remainder of the paper.

5.1 Representation of the Hilarics Description and a Geometrical Interpretation

As a *shorthand notation* of the structure description, we use a binary tree. The translation from a Hilarics description to its tree representation is fairly straightforward. Note that this translation is not a transformation in the sense described in Section 4.2.2. The tree is just an alternative way—better suited for manipulation by a computer program and more concise to reason with—of representing the Hilarics description.

To obtain this tree representation, we first rewrite for_loops as a combination of if_then_else constructs. Once this is done, the tree can be constructed very easily: the nodes of the tree are the tests of the if_then_else constructs, the *left* branch of each node corresponds to the *then* case of the if_then_else expression and the *right* branch corresponds to the *else* case. The leaves of the tree are the net definitions. We call this tree the *structure tree*. The translation of the tree representation back to a Hilarics description is also straightforward. (To get a feeling of what such a tree looks like, the reader can compare the Hilarics structure description of Section 4.1 with its equivalent tree representation depicted on the left-hand side of Figure 12.)

Completely analogous, we translate the transformation description into a binary tree. The nodes and branches of the tree have the same meaning as in the previous case, only now the leaves represent the primitive transformations that must be applied. This tree is called the *transformation tree*.

Such a tree has the following geometrical interpretation. By following a path from the root to a leaf, the tests that are passed delimit a polytope, i.e., a region in the $n + m$-dimensional space of the n indices and the m parameters. For all points with integer coordinates inside this polytope, the net definition in the leaf of the root-leaf path defining the polytope is valid. Here valid means that the nets obtained by instantiating the net definition with the integer coordinates are present in the structure. In the case of the transformation tree, the primitive transformation of the leaf must be executed for all integer points in the polytope.

So, the tests in the root-leaf paths define polytopes in which there are items (components, nets, primitive transformations, etc.) on all integer points inside the polytope.

5.2 Merging of the Trees

If a particular primitive transformation applies to some of the nets defined by a particular net definition, then the two polytopes mentioned earlier will have an intersection.

To determine whether a particular net definition in the structure tree must be transformed by a particular transformation in the transformation tree, we merge the trees: we take the complete transformation tree and place it in the structure

tree just in front of each leaf. In that way, we obtain a new tree, which we call the *merged tree*. This merging operation corresponds to the schematical merging of the structure transformation description depicted in Figure 7.

Now, when following a path from the root to a leaf of the merged tree, the first tests encountered, delimit the region where the net definition is valid. Then the tests delimiting the operation region of the primitive transformation are encountered. Finally, in the leaf itself, we now have the net definition and the primitive equivalence transformation. If there exists an intersection between the polytopes delimited by the tests on a root-leaf path of the merged tree, the nets in that intersection must be transformed.

5.3 Solving the Graph

In fact, this merged tree already describes more or less the transformed structure, be it in a rather complicated way. We now want to make this description more concise by executing the primitive transformations where necessary. Therefore, we need a decision procedure, which we call DECIDE, that can tell us whether the polytopes, defined by the (in)equalities on the root-leaf path, have an intersection. In other words, we need a procedure that can tell us whether a set of (in)equalities, defining the two polytopes, has a feasible solution. To calculate the more concise description of the transformed structure, we examine every root-leaf path of the merged tree. We apply DECIDE to the tests in that root-leaf path. If they do not have a feasible solution, then the net in the leaf is not transformed and it remains as such in the description of the final structure. If the tests have a feasible solution, the nets in the leaf must be transformed by the primitive transformation in the leaf, and after execution of this transformation we obtain the net definition that appears in the final description.

When the tests on a root-leaf path have an intersection, it is possible that this intersection can be described with much fewer tests than actually present in the path. A number of tests will be redundant, meaning that they are always true or always false given the previous tests in the path. For example, in the sequence $i \geq 1$, $i \leq m$, $i \leq m + 1$, the last equation is redundant because given the two previous inequalities it can be decided to be always true. If we would replace the last inequality by $i \geq m + 1$, then it would still be redundant (always false). If we replace it by $i \leq m - 1$, we could not make a decision. The DECIDE procedure should be able to detect redundant (in)equalities.

After complete traversal of the merged tree, we now have a tree that corresponds to the Hilarics description of the transformed structure.

Two points still must be examined:

1. The execution of the specific primitive transformation and the checking of the corresponding conditions. Since this is dependent on the specific primitive transformation applied, we postpone the discussion of this point until we treat the example.
2. The decision procedure DECIDE. which will be discussed in the next section.

6 ALGORITHMS FOR DECIDE

In this section, we discuss the decision procedure. First, we analyze the requirements for the DECIDE procedure, and then examine a possible solution.

6.1 Requirements for DECIDE

From the previous sections, the following requirements for DECIDE can be derived:

1. DECIDE should be able to decide whether a set of equalities and inequalities has an *integer* feasible solution. The feasible solution should be integer, because the variables denote indices and parameters (see Section 5.1).
2. DECIDE should be able to detect if an (in)equality is redundant.
3. To be able to describe arbitrary VLSI designs in a concise way, Hilarics allows nonlinear expressions, such as $i \times j$, $div(i, j)$, $mod(i, j)$, etc. These functions are used, for example, in describing systolic arrays. This implies that DECIDE should be able to handle *nonlinear constraints*.
4. DECIDE is at the heart of the calculation technique. It will be used intensively and, thus, should be as *fast* as reasonably possible.

A possible algorithm satisfying these requirements is discussed in the next section.

6.2 Possible Algorithm for DECIDE

The inequalities we want to decide can be put in the form of *Presburger formulas* [26]. These are built up from integers, integer variables, addition, multiplication by constants, arithmetical relations, and first-order logical connectives. Suppose we want to decide whether two polytopes defined by the following sequence of conditionals have an intersection,

IF $i \leq 2j + 3$ THEN IF $i = 3j + 1$ THEN NET
IF $j \geq 2i - 1$ THEN IF $2i + 3j - 1 \geq 0$ THEN TRANSFORM

then the validity of the following Presburger formula must be examined:

$$(i \leq 2j + 3) \wedge (i = 3j + 1) \wedge (j \geq 2i - 1) \wedge (2i + 3j - 1 \geq 0).$$

The elimination of redundant (in)equalities can also be formulated as a Presburger formula. If we want to know whether the last inequality of the previous formula is redundant and always true, we must check whether the following Presburger formula is true:

$$((i \leq 2j + 3) \wedge (i = 3j + 1) \wedge (j \geq 2i - 1)) \Rightarrow (2i + 3j - 1 \geq 0).$$

If we want to know whether the last inequality of the previous formula is redundant and always false, we must check whether the following Presburger formula is true:

$$((i \leq 2j + 3) \wedge (i = 3j + 1) \wedge (j \geq 2i - 1)) \Rightarrow \neg(2i + 3j - 1 \geq 0).$$

If both of the preceding Presburger formulas are false, then the last inequality is not redundant.

Presburger formulas have been shown to be decidable by Presburger [26]. A decision procedure that can decide about validity as well as invalidity of Presburger formulas has been proposed by Shostak [27]. This procedure consists of two stages. First, the formula to be proven is reduced to a set of integer linear programming problems (ILP), with the property that the formula is valid if and only if none of the ILPs has a solution. In the second stage, each of the ILPs is tested for solvability. In his paper, Shostak describes a method that is an extension of the *Sup-Inf* method [28] for dealing with this second stage. An alternative would be to use a more classical ILP algorithm, such as Gomory's [29].

From the description of Presburger formulas, it is clear that requirement 3 has not been met as yet. However, Shostak also proposed a decision procedure to deal with an extension of Presburger arithmetic in which function symbols [30] occur. This method consists of reducing a formula in which function symbols occur, to a genuine Presburger formula. This genuine Presburger formula can then be decided by using the preceding two-stage approach.

6.3 Complexity Issues

In general, the decision algorithm for Presburger arithmetic extended with function symbols is NP-complete. Each Presburger formula leads to several ILPs, and the ILP is known to be an NP-complete problem [31, p. 350]. This would imply that the technique proposed herein is not applicable to hardware descriptions with a lot of nested IF THEN ELSE constructs and many indices and parameters. However, when we use *Gomory's cutting plane* algorithm [29] to solve the ILPs, this technique is feasible. The reason for that is twofold:

- Gomory's algorithm is based on first solving a relaxed linear programming problem (LP). The relaxed LP is identical to the ILP but without the constraint for an integer solution. If there is a (noninteger) solution for the relaxed LP, then Gomory's algorithm starts iterating to an integer solution, if such a solution exists. To solve the relaxed LP, the classical Simplex algorithm is used, which behaves well in all practical cases.

  Presburger formulas with function symbols give rise to several ILPs, but thse ILPs are related to each other. If the algorithm that solves the ILP can work incrementally, several of these new ILPs can be solved without much additional effort. We decided in favor of the Simplex-based Gomory algorithm because it can work incrementally.
- Experience has shown that the edge points of the intersection of the polytope of the structure root-leaf path and the polytope of the transformation root-leaf path are mostly integer. This gives rise to ILP problems in which all edge points

  are integer. The constraint matrix of such an ILP with integer edge points is called *totally unimodular* (TUM). It is a property of ILPs with a TUM matrix, that the optimal solution obtained after solving the relaxed LP problem will be integer. This is fairly logical, knowing that Simplex finds an optimal solution by iterating from edge point to edge point. Thus, with a TUM matrix, there is no need for the *cutting plane* iteration step of Gomory's algorithm, and we will be able to make a decision after only the first step of the algorithm.

The algorithm described here is thus feasible because:

1. Presburger formulas generate ILPs that can be solved *incrementally* using Gomory's algorithm.
2. Because the ILPs originate from hardware descriptions, they often describe polytopes with integer edge points. These result in ILPs with TUM matrices, which can be solved by applying *Simplex only once* without the need to iterate on the Gomory algorithm.
3. One of the phases in solving the graph (see Section 5.3) consists of simplifying the resulting structure description by eliminating redundant (in)equalities. To do this, we generate a number of Presburger formulas, where every consecutive Presburger formula has one more (in)equality until a redundant one is detected. All of these consecutive Presburger formulas can also be solved *incrementally*.
4. Finally, even the *real* ILPs, which need to be solved by repeatedly iterating on the Gomory algorithm, are still easy to solve because of the small size of the problems [only a small number of (in)equalities and variables].

7 BOOTH MULTIPLIER-ACCUMULATOR MODULE

The Booth multiplier module is one of the executional units used in the silicon compilation system Cathedral II [4]. It is defined in a parameterized way in order to be reusable for a large number of applications.

In the first part of this section, an overview will be given of the Booth module and of the global tranformation of its naive implementation into its final efficient implementation. In the second part, one particular transformation step will be investigated more profoundly.

7.1 Transformation of Booth Multiplier Module

The behavioral specification of the module is quite simple. The module has as inputs the n-bit multiplicand X and the m-bit multiplier Y. The $(n + m)$-bit product is conditionally accumulated with the previous calculated product, depending on the *CACC* flag. The high-level specification, in the register-transfer language Logmos [32], of the module is as follows:

```
CELL Mult_acc
  TERMINALS X[0..N-1] Y[0..M-1] EN CACC BUS[0..N+M-1]
END

CELL Mult_acc
  IN=X[0..N-1] Y[0..M-1] EN CACC
  REG=ACCREG[N+M-1..0]
  SIGNAL=ACCOUT[N+M-1..0] MULT_ACCOUT[N+M-1..0]
  TRI_OUT=BUS[N+M-1..0]

BEGIN
  IF CACC THEN ACCOUT=ACCREG
          ELSE ACCOUT=#0_D[N+M] END
  ACCREG<==MULT_ACCOUT
  MULT_ACCOUT=ADD(MPY(X, Y), ACCOUT)
  IF EN THEN BUS=ACCREG END
END
```

The operations ADD and MPY in the register-transfer-level (RTL) specification are defined in terms of naive implementations. However, in view of speed, area, and other trade-offs, these will not be used in the actual implementation. For reasons of hardware efficiency, the Booth algorithm is chosen as the implementation of the MPY operation, and a straightforward implementation of this algorithm results in the structure of Figure 8. The correctness of the Booth algorithm

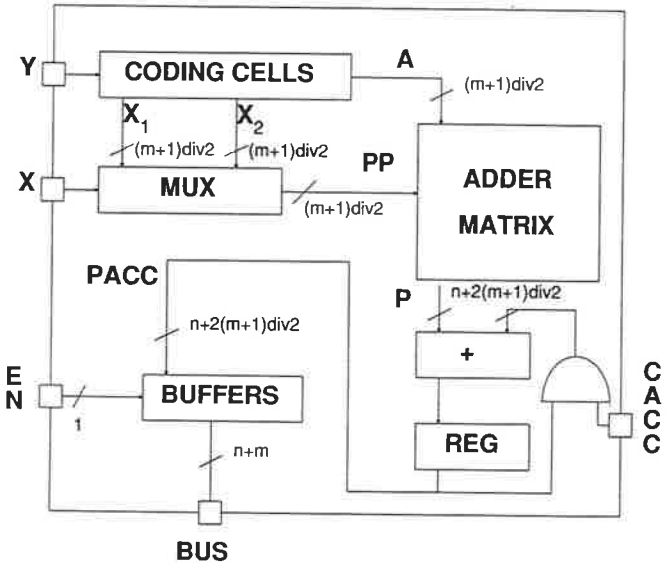


Figure 8. Booth multiplier module.

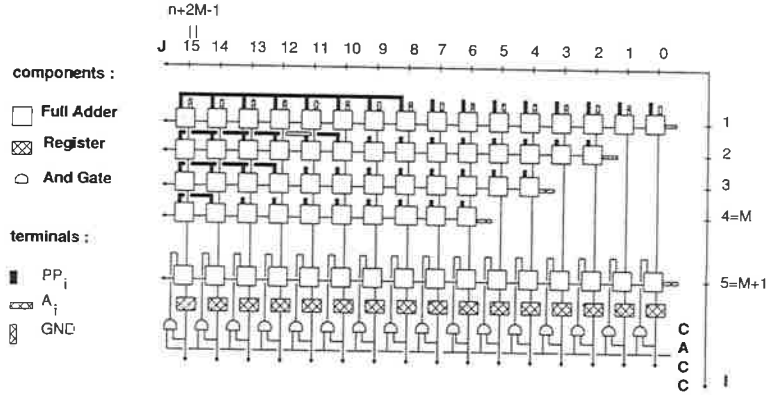


Figure 9. Naive implementation of adder matrix and accumulation structure.

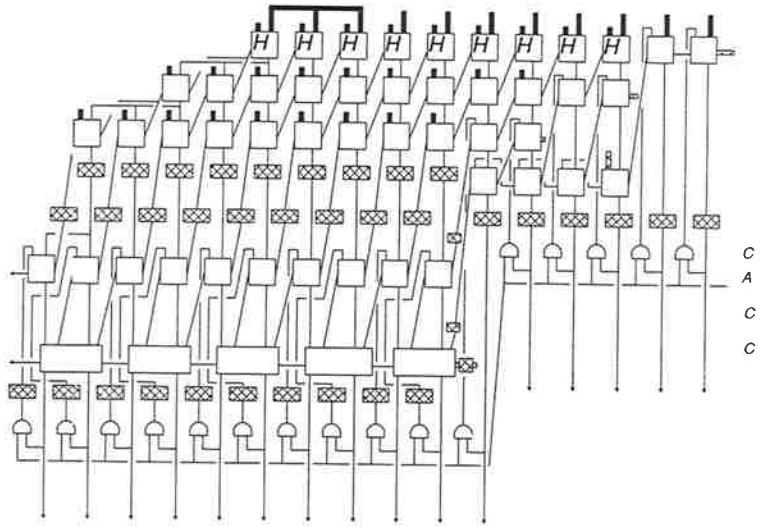


Figure 10. Final refined implementation of the structure.

with respect to the straightforward shift-add implementation of MPY is given in [19].

The naive implementation of the Booth algorithm, however, can still be optimized. The main improvements are situated in the organization of the adder matrix and the accumulator structure, so this is what we will focus on. The naive implementation of these two blocks is given in Figure 9 for an 8×8 instance. By

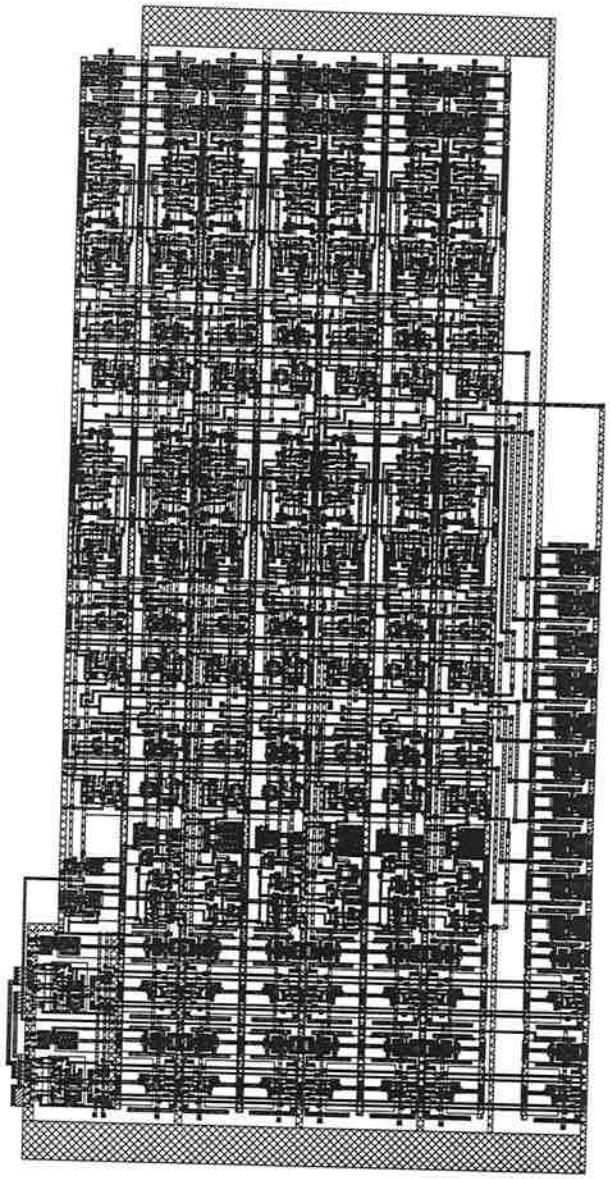


Figure 11. Automatically generated module layout of an 6 × 6 multiplier.

using 15 transformation descriptions, the initial parameterized description is transformed into the final parameterized structure description, which corresponds to the actual implementation shown in Figure 10.

A parameterized layout module generator for the multiplier accumulator, corresponding to the final parameterized structure in Figure 10, has been designed in the Module Generation Environment (MGE) environment [2]. Figure 11 shows the layout of a multiplier instance of 6 × 6 bits.

7.2 Elaboration of a Transformation Step

The transformation step we have chosen to work out in this section is the transformation from a naive carry-propagate structure for the adder matrix into a more efficient carry-save structure. The transformation step is illustrated in Figure 6. The upper part of the figure shows the result of the previous transformation step. The components with shadows are affected by the transformation description in the middle of the figure. The lower part of the figure shows the result of the transformation step.

As far as the structure description is concerned, we are interested only in the part that describes the carry inputs and outputs. This part has already been described in Section 4.1. After translating this description into its equivalent tree representation, we get the structure tree of Figure 12. The transformation tree, corresponding to the transformation description of Figure 6, is also given in Figure 12.

The first operation is the merging of the two trees. We start with the leftmost root-leaf path of the structure tree and insert the transformation graph before the net definition in the leaf. Then we take the next path, and so on until the structure tree has been traversed completely (see Figure 13).

The second operation is the solving of the graph. This operation can be performed during the construction of the merged graph. For each root-leaf path, we apply the DECIDE procedure. If DECIDE finds that there is an intersection, we eliminate as many redundant tests as possible. After this operation has been completed for all the paths of the merged tree, we obtain the tree in Figure 14.

From here on the operations are specific for the *comm* transformation. To start, the conditions for the validity of the *comm* operation must be checked.

1. The components must allow commutativity. This is obvious here: all components mentioned in the transformation description are full adders.
2. The terminals that are exchanged must have the same weight factor and delay potential. As can be seen from the merged tree in Figure 14, the weight factors of the exchanged terminals are indeed all the same (i.e., j). Also all the delay potentials are equal (i.e., to 0).
3. Finally, all terminals that are exchanged must occur once as the left-hand-side argument and once as the right-hand-side argument of *comm*. If this is not the case, there will be unconnected terminals after execution of the transformation. This can be checked in the following way:

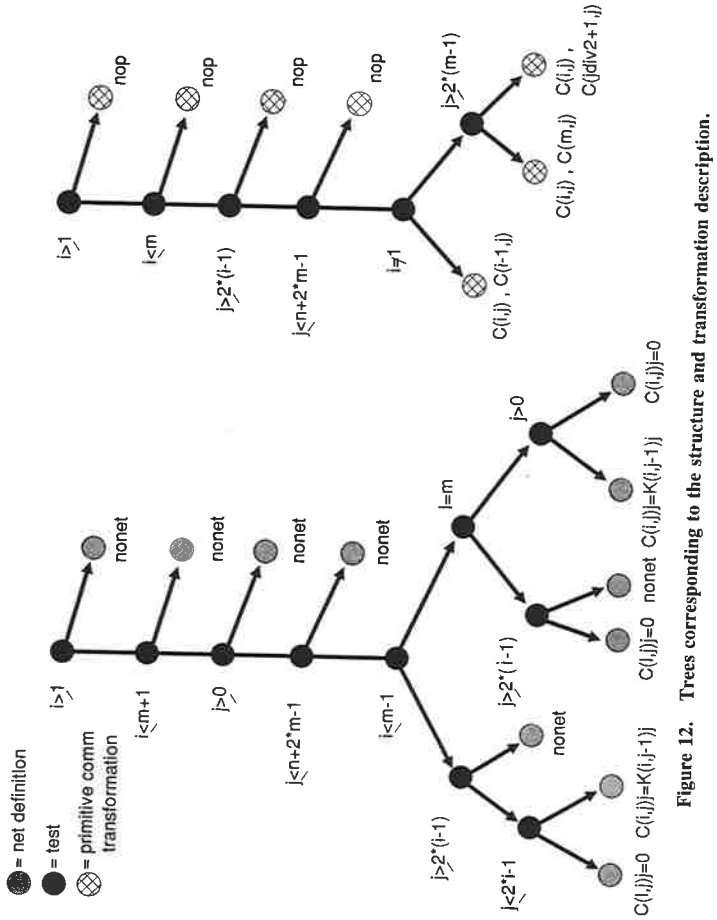


Figure 12. Trees corresponding to the structure and transformation description.

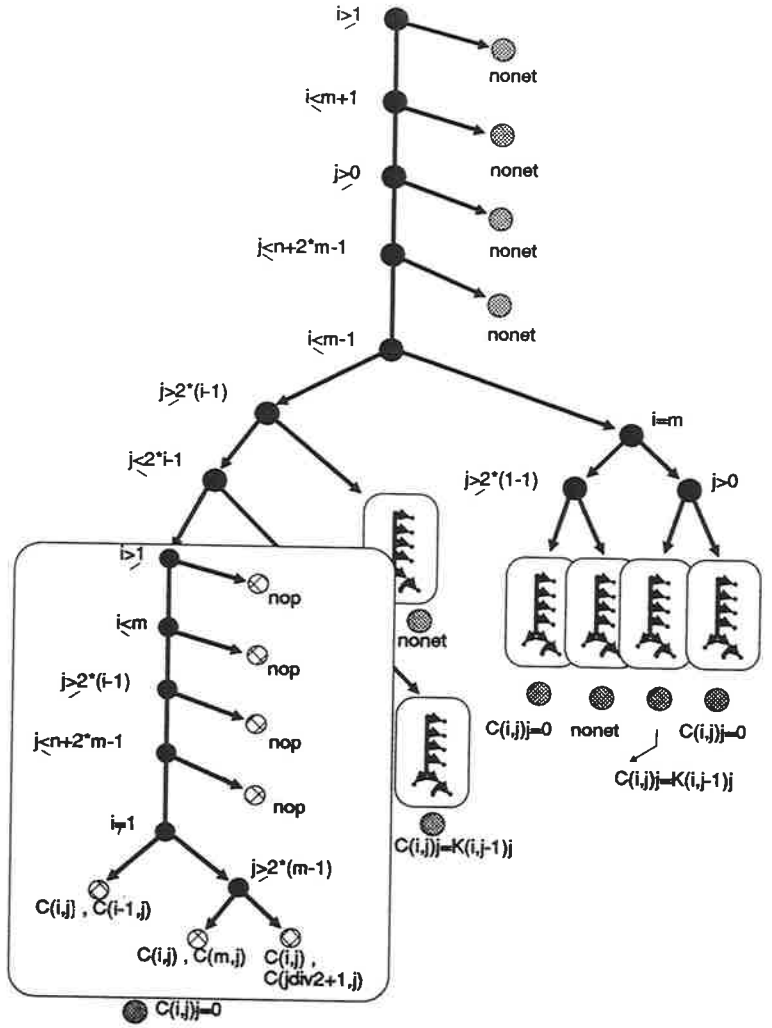


Figure 13. Resulting tree after merging the structure and transformation tree.

- For each path to a leaf of the merged tree.
- Take the left-hand-side argument of *comm* and determine the region in which it is defined. This results in a tree that corresponds to the region where the left-hand-side arguments of *comm* are transformed.

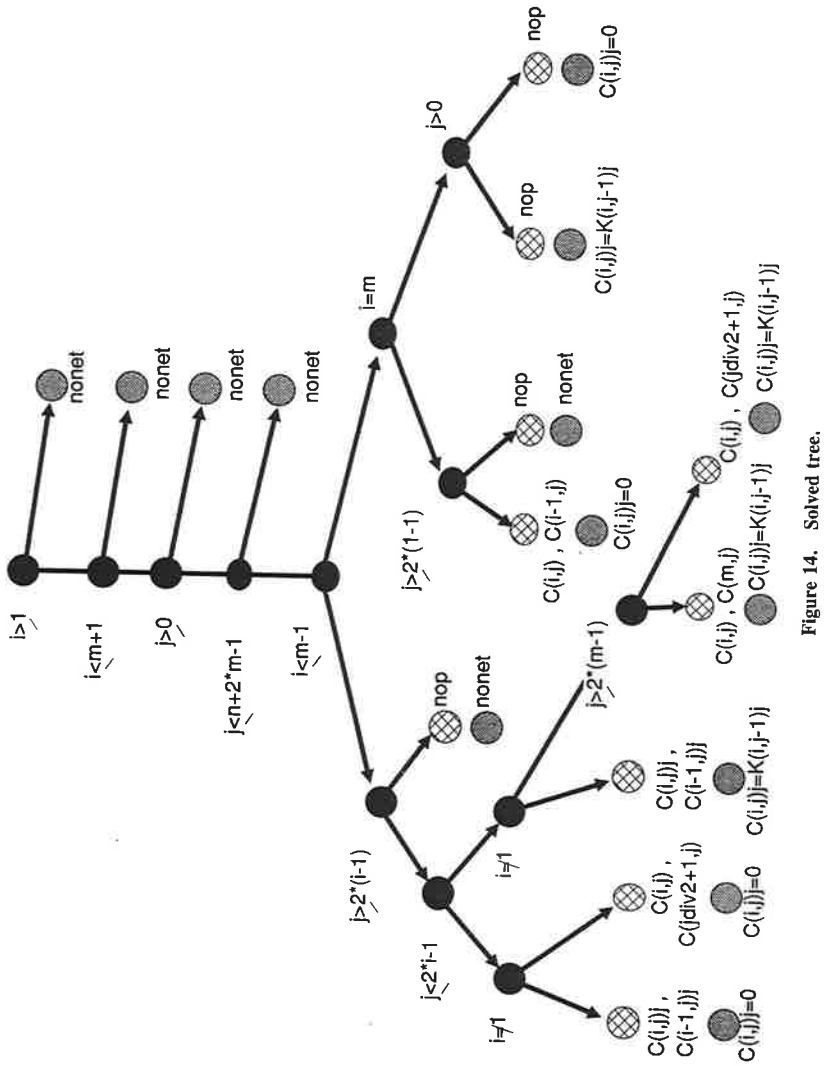


Figure 14. Solved tree.

- Do the same for the right-hand-side argument of *comm*.
- If the two regions defined by the resulting trees are the same, then the condition is fulfilled.

This last condition could also be tested after completion of the transformation step, because then all terminals should be connected if the transformation was valid.

The next step is execution of the primitive transformation. Let us take the leftmost path in the merged tree in Figure 14. We have to make a new net, of which one terminal is $C(i, j)$ and the other terminal is the terminal that is connected to $C(i-1, j)$ in the original structure description. Hereby i and j are determined by the tests along the path:

$$i \geq 1 \rightarrow i \leq m+1 \rightarrow j \geq 0 \rightarrow j \leq n+2*m-1 \rightarrow i \leq m-1 \rightarrow j \geq 2*(i-1) \rightarrow j \leq 2*i-1 \rightarrow i \neq 1.$$

Now we have to look for the new right-hand side of $C(i, j)$, which is the right-hand side of $C(i-1, j)$ in the original structure description. To find this terminal, we merge the original structure description with the tree formed by the tests mentioned earlier, but with i substituted by $i+1$ to take into account the $i-1$ index of $C(i-1, j)$. This gives the tree in Figure 15. After solving this merged graph, we obtain the new right-hand-side terminal of $C(i, j)$.

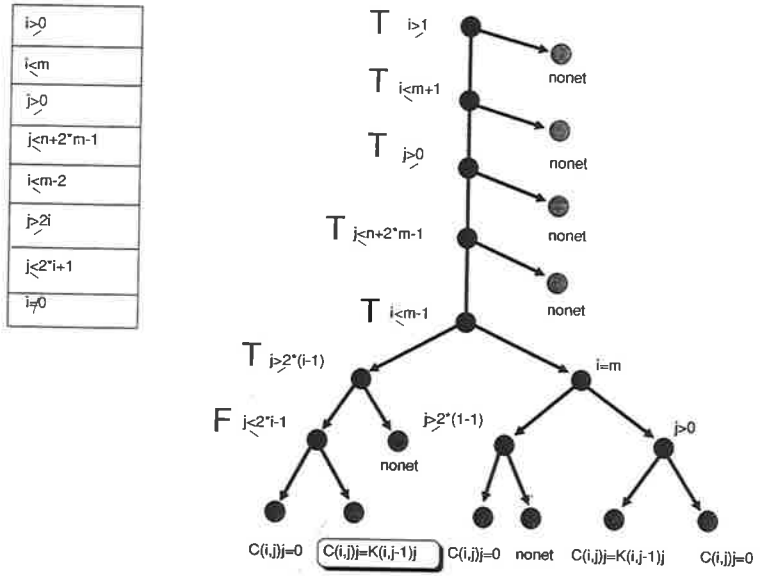


Figure 15. Tree describing the $C(i-1, j)$ terminals.

This procedure is repeated for every leaf of the merged tree of Figure 14, in which the primitive equivalence transformation is mentioned. After traversal of the complete tree, we get the structure tree corresponding to the carry-save structure indicated in Figure 16.

After retranslating this tree into Hilarics, the structure description for the transformed carry inputs is obtained.

```
for i = 1 to m + 1 do
 for j = 0 to n + 2 * m − 1 do
  if i ≤ m − 1 then if j ≥ 2 * (i − 1) then if i ≠ 1 then C(i, j)j = K(i − 1, j − 1)j
                                                  else C(i, j)j = 0
                                            fi
                                       else nonet
                    fi
               else if i = m then if j ≥ 2 * (i − 1) then C(i, j)j = K(i − 1, j − 1)j
                                                     else nonet
                                  fi
                               else if j > 0 then C(i, j)j = K(i, j − 1)j
                                             else C(i, j)j = 0
                                  fi
                    fi
  fi
 rof
rof
```

This is the Hilarics description of the carry inputs and outputs of the structure in the bottom part of Figure 6. This structure description can then be used as the input for the next transformation step, until finally the desired structure is obtained.

8 SYSTOLIC FINITE IMPULSE RESPONSE FILTER

In this last section, we take the opportunity to show a few steps of the transformational design of an FIR filter. We would like to derive a systolic implementation of this filter, starting from a straightforward implementation of its function.

An FIR filter has an output y that is on each time point t equal to the sum of the N previous input signals x; it implements the following function (N is a parameter):

$$y^t = \sum_{i=1}^{N} a_i x^{t-i\Delta}$$

A straightforward implementation of this function is given in Figure 17 for a parameter value of $N = 9$. A systolic implementation of this FIR filter consists of N processing elements in a matrix structure. In between each processing element there is a register. To derive this systolic implementation from the straightforward implementation, a number of retiming transformations must be done. These transformations are shown in Figures 18–21. This last implementation is, in fact, the

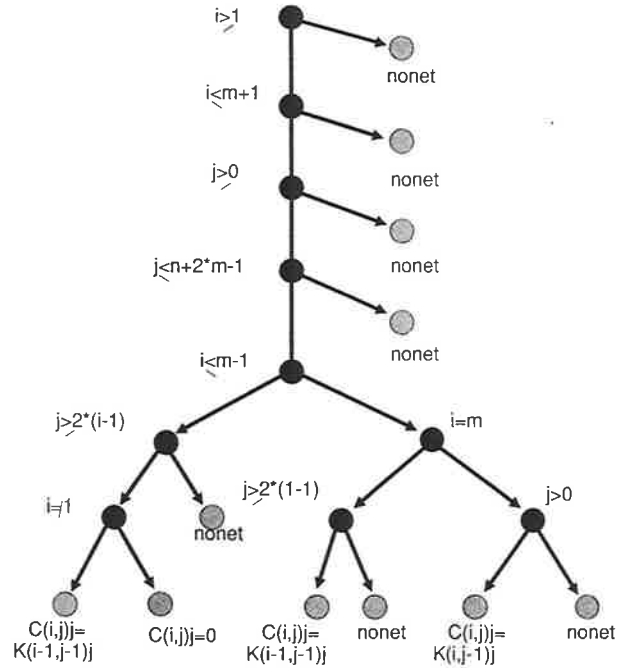


Figure 16. Tree corresponding to the transformed structure description.

systolic implementation we are looking for: after rearranging the blocks (without changing the structure), we get Figure 22.

Based on the technique described in Section 5, the structure description corresponding to Figure 22 can be calculated starting from the structure description of Figure 17, which was, in turn, a straightforward implementation of the function of the filter. The registers that must be moved are indicated by the tests in the transformation descriptions. The net definitions that describe nets going into and out of these registers can be identified in the structure description by looking for the intersections of the polytope in which the net definition is valid and the polytope in which the transformation definition is valid. The net definitions that must be changed, because the registers are being moved into the nets defined by them, can be identified in the same way.

9 CONCLUSIONS

In this paper, we have presented a synthesis and/or verification method that is built on the concept of correctness preserving transformations. The transformations are

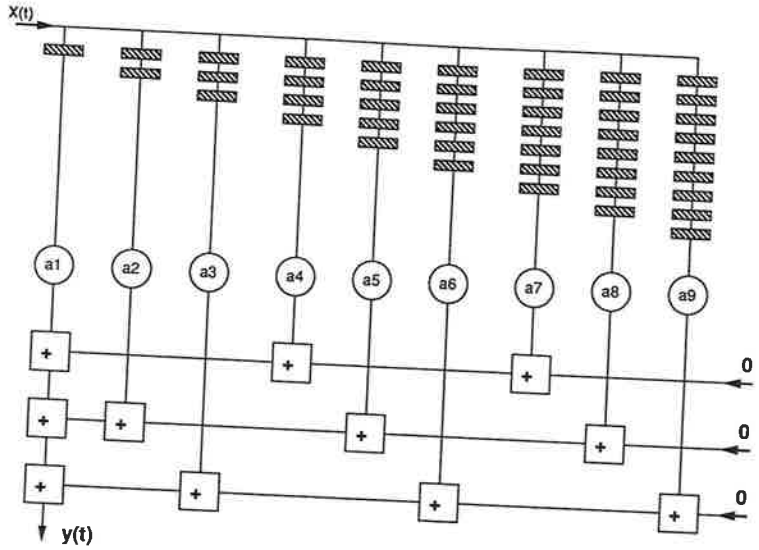


Figure 17. Straightforward implementation of the function of an FIR filter.

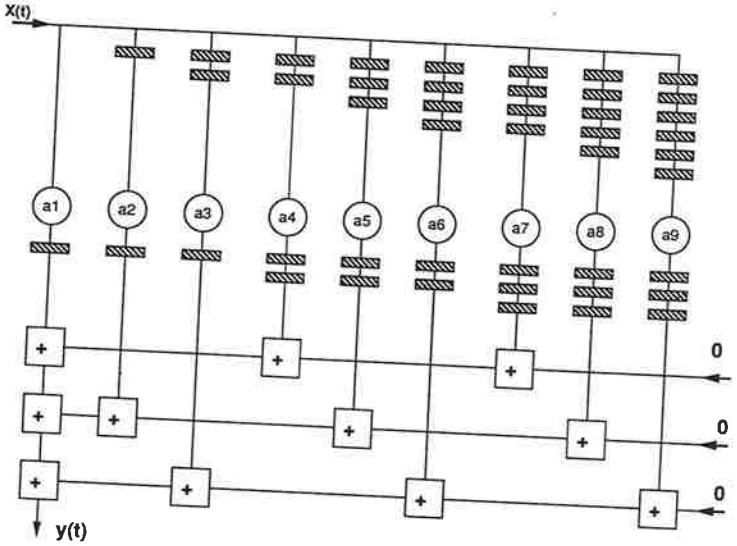


Figure 18. Shifting some registers over the coefficient blocks.

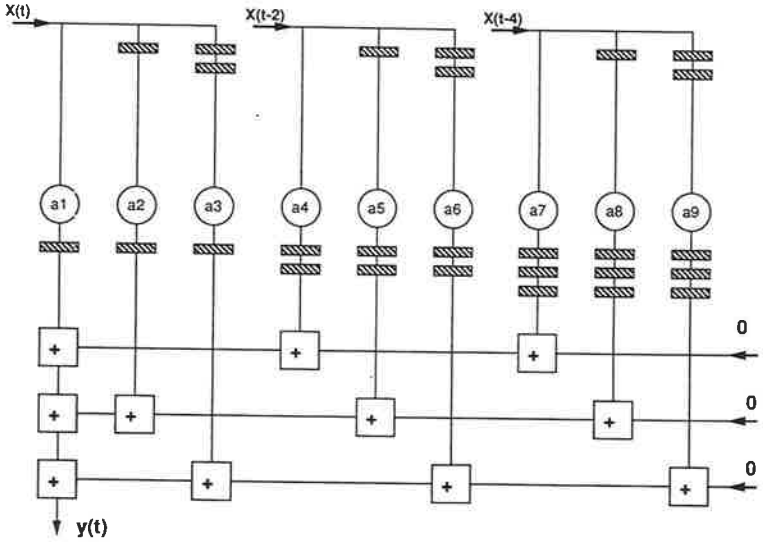


Figure 19. Shifting the input signals in time.

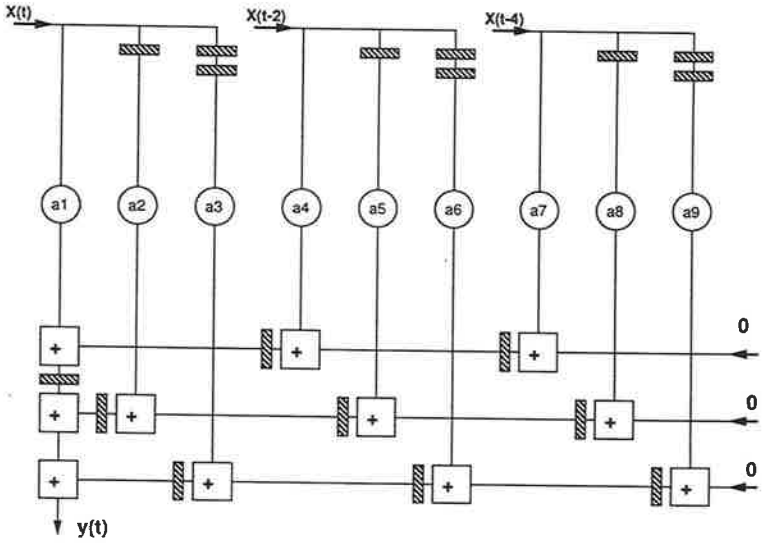


Figure 20. Shifting some registers over the adder blocks.

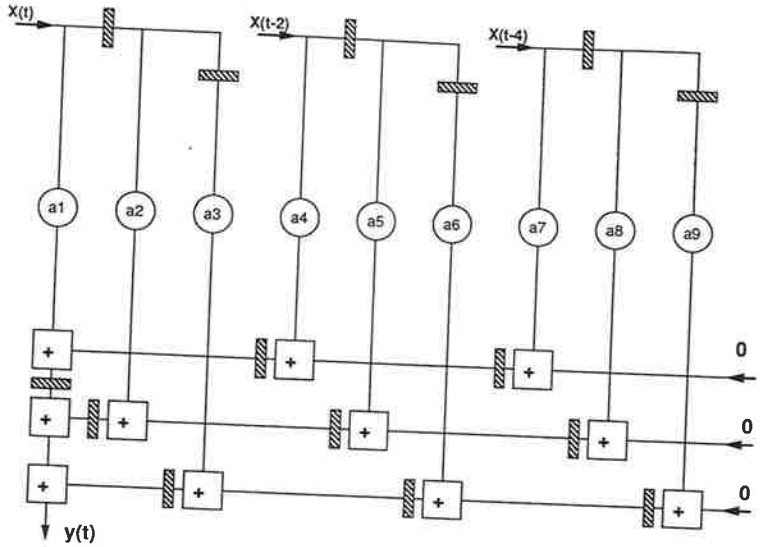


Figure 21. Rearranging the input registers.

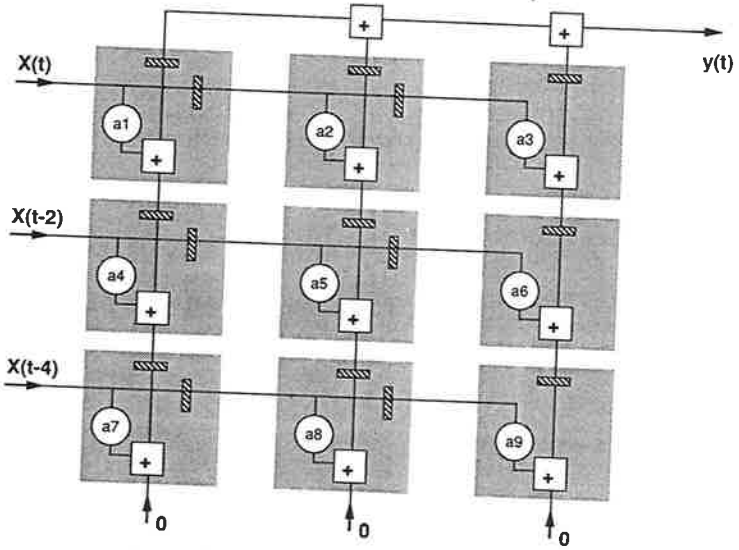


Figure 22. Systolic implementation of the FIR filter.

performed on parameterized structure descriptions in order to come from a specification up to an actual implementation. The transformations are formalized using transformation descriptions, which, in turn, make use of primitive equivalence transformations. Algorithms have been presented that implement this method using an existing hardware description language (HDL), Hilarics. Note, however, that the method could be applied to other HDLs as well, as long as the HDL is applicative and allows IF THEN ELSE and FOR loop statements in the structure description.

The method has been applied for the design of a parameterized Booth multiplication module and a parameterized systolic implementation of a finite impulse response filter.

As proposed herein, the synthesis technique cannot be used for behavior synthesis. Further research will concentrate on applying the same principles as outlined herein to include manipulations on register-transfer-level constructs and on high-level system specifications in the Silage language. In the latter case, it will be very useful for the guided synthesis of high-speed video-type signal processors. In addition, the formal proof of the correctness of the transformation descriptions will be investigated.

REFERENCES

[1] H. De Man, "Evolution of CAD Tools Towards Third Generation Custom VLSI Design," *Revue Phys. Appl.*, Vol. 22, pp. 31–45, January 1988.

[2] P. Six, L. Claesen, J. Rabaey, and H. De Man, "An Intelligent Module Generation Environment," *Proc. of 23rd Design Automation Conf.*, Las Vegas, NV, 1986, pp. 730–735.

[3] R. Jain et al., "Custom Design of a VLSI PCM-FDM Transmultiplexer from System Specifications to Circuit Layout Using a Computer Aided Design System," *IEEE Journal of Solid-State Circuits*, Vol. SC-21, No. 1, pp. 73–85, February 1986.

[4] H. De Man, J. Rabaey, P. Six, and L. Claesen, "Cathedral II: A Silicon Compiler for Digital Signal Processing," *IEEE Design & Test of Computers*, Vol. 3, No. 6, pp. 73–85, December 1986.

[5] P.N. Hilfinger, "A High Level Language and Silicon Compiler for Digital Signal Processing," *Proc. IEEE 1985 Custom Integrated Circuits Conf.*, 1985, pp. 213–216.

[6] P. Camurati, and P. Prinetto, "Formal Verification of Hardware Correctness: An Introduction," *Proc. of IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, April 1987, pp. 225–247.

[7] W.A. Hunt, "FM8501: A Verified Microprocessor," *IFIP WG 10.2 Workshop: from HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, France, September 1986, pp. 85–114.

[8] J.J. Joyce, "Formal Verification and Implementation of a Microprocessor," *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P. Subrahmanyam, Kluwer Academic, Norwell, MA, 1988, pp. 129–157.

[9] M. Sheeran, "μFP, an Algebraic VLSI Design Language," Ph.D. thesis, Programming Research Group, Oxford University, 1983.

[10] M. Sheeran, "Describing and Reasoning about Circuits Using Relations," *Proc. of Leeds Workshop on Theoretical Aspects of VLSI Design*, 1986.

[11] P.A. Subrahmanyam, "The Algebraic Basis of an Expert System for VLSI Design," *Formal Aspects of VLSI Design*, edited by G. L. Milne and P. A. Subrahmanyam, Elsevier, New York, 1986, pp. 59–81.

[12] G.J. Milne, "Contextual Constraints for Design and Verification," *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P. Subrahmanyam, Kluwer Academic, Norwell, MA, 1988, pp. 257–265.

[13] H. Eveking, "Verification, Synthesis and Correctness Preserving Transformations—Cooperative Approaches to Correct Hardware Design," *IFIP WG 10.2 Workshop: From HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, France, September 1986, pp. 229–239.

[14] M. Gordon, "HOL: A Proof Generating System for Higher-Order Logic," *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P. Subrahmanyan, Kluwer Academic, Norwell, MA, 1988, pp. 73–128.

[15] R.E. Bryant, "Boolean Analysis of MOS Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 4, pp. 634–649, July 1987.

[16] H. De Man et al., "DIALOG: an Expert Debugging System for MOS VLSI Design," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-4, No. 3, pp. 301–311, June 1985.

[17] G.D. Hachtel, and R.M. Jacoby, "Verification Algorithms for VLSI Synthesis," *NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design*, SSGRR, l'Aquila, Italy, July 1986.

[18] P. Lammens, L. Claesen, and H. De Man, "Correctness Verification of VLSI Modules Supported by a Very Efficient Boolean Prover," *Proc. of IEEE International Conference on Computer Design*, 1989, pp. 266–269.

[19] L.P. Rubinfield, "A Proof of the Modified Booth Algorithm for Multiplication," *IEEE Transactions on Computers*, Vol. C-24, pp. 1014–1015, October 1975.

[20] L. Claesen, P. Johannes, D. Verkest, and H. De Man, "Guided Synthesis and Formal Verification Techniques for Parameterized Hardware Modules," *Proc. of COMPEURO 88*, 1988, pp. 90–99.

[21] D. Verkest et al., "Formal Techniques for Proving Correctness of Parameterized Hardware Using Correctness Preserving Transformations," *IFIP WG10.2 Workshop: The Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988. pp. 77–98.

[22] E. Vanden Meersch, and R. Severyns, "HILARICS: User's Manual, 2nd edition," Internal Report IMEC MR03-KUL-7-B3-2, January 1986.

[23] C.E. Leiserson, and J.B. Saxe, "Optimizing Synchronous Systems," *22nd Annual Symposium on Foundations of Computer Science*, IEEE, October 1981, pp. 23–36.

[24] R.T. Boute, "System Semantics and Formal Circuit Description," *IEEE Transactions on Circuits and Systems*, Vol. CAS-33, No. 12, pp. 1219–1231, December 1986.

[25] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, Vol. 21, No. 8, pp. 613–641, August 1978.

[26] M. Presburger, "Uber die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zählen in Welchem die Addition als einzig Operation hervortritt," *Sprawozdanie z I Kongresu Matematykow Krajow Slowcanskich Warszawa*, Warsaw, Poland, pp. 92–101, 1929.

[27] R.E. Shostak, "On the Sup-Inf Method for Proving Presburger Formulas," *Journal of the ACM*, Vol. 24, No. 4, pp. 529–534, October 1977.

[28] W.W. Bledsoe, "A New Method for Proving Certain Presburger Formulas," *Advance Papers 4th International Joint Conference on Artificial Intelligence*, Tibilisi, Georgia, USSR, September 1975, pp. 15–21.

[29] R.E. Gomory, "An Algorithm for Integer Solutions to Linear Problems," *Recent Aavances in Mathematical Programming*, edited by R. L. Graves and P. Wolfe. New York: McGraw-Hill, 1963, pp. 269–302.

[30] R.E. Shostak, "A Practical Decision Procedure for Arithmetic with Function Symbols," *Journal of the ACM*, Vol. 26, No. 2, pp. 351–360, April 1979.

[31] C.H. Papadimitriou, and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ. Prentice-Hall, 1982.

[32] R. Severyns, and E. Marien, "LOGMOS User's Guide," Internal Report IMEC, 1986.

Diederik T. M. Verkest received his Electrical Engineering degree from the Katholieke Universiteit Leuven, Belgium in 1987. He joined the IMEC laboratory in 1987 in Heverlee (Belgium) as a research assistant, where he is currently working toward his Ph.D. in the field of computer aided design of integrated systems for digital signal processing. His research interests are in computer aided design, formal design and verification methods and theorem proving systems. He is a student member of IEEE.

Luc J. M. Claesen (S'77–M'85) received his Electrical Engineering degree and his Ph.D. degree from the Katholieke Universiteit Leuven, Belgium in 1979 and 1984, respectively. After graduation in 1979 he joined the ESAT-laboratory, Katholieke Universiteit Leuven, as a research assistant. There he has been working in the field of computer aided design of integrated systems for digital and analog signal processing. Since July 1984 he is with the IMEC laboratory in Heverlee (Belgium), where he is heading research in the design management and verification group within the VLSI systems design methodologies division. Since 1989 he became an associate professor at Katholieke Universiteit Leuven. He received a Best Paper Award at the ICCD'86 conference for work on an analysis and verification system for digital signal processing systems. His research interests are in computer aided design, formal design and verification methods, integrated digital signal processing and image synthesis systems. He is a member of IFIP working group 10.2 "System Description and Design Tools."

Hugo J. De Man received his electrical engineering degree and his Ph.D. degree in Applied Sciences from the Katholieke Universiteit Leuven, Heverlee, Belgium, in 1964 and 1968, respectively. In 1968 he became a member of the staff of the Laboratory for Physics and Electronics of Semiconductors at the University of Leuven, working on device physics and integrated circuit technology. From 1969 to 1971 he was at the Electronic Research Laboratory, University of California, Berkeley, as an ESRO-NASA postdoctoral research fellow, working on computer-aided device and circuit design. In 1971 he returned to the University of Leuven as a research associate of the NFWO (Belgian National Science Foundation). In 1974 he became a professor at the University of Leuven. During the winter quarter of 1974–1975 he was a visiting associate professor at the University of California, Berkeley. He was an associate editor for the IEEE Journal of Solid-State Circuits from 1975–1980 and was european associate editor for the IEEE Transactions on CAD from 1982 to 1985. He received a Best Paper Award at the ISSCC of 1973 on bipolar device simulation and at the 1981 ESSCIRC conference for work on an integrated CAD system. In 1986 he became a fellow of the IEEE. His actual field of research is the design of integrated circuits and computer-aided design. Since 1984 he is vice president of the VLSI systems design group of IMEC (Leuven, Belgium).